Compilation optimisante pour processeurs extensibles

Antoine Floc'H

N° d'ordre : 4538 ANNÉE 2012

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Antoine Floc'h

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires - CAIRN
Composante universitaire : ISTIC

Compilation optimisante pour processeurs extensibles

Thèse soutenue à Rennes le 8 juin 2012

devant le jury composé de :

Tanguy Risset
Professeur INSA Lyon / président

Albert Cohen
Directeur de recherche INRIA / rapporteur

Pierre Boulet
Professeur Université de Lille 1 / rapporteur

Vincent Lorquet
Architecte compilateur STM / examinateur

Krzysztof Kuchcinski
Professeur Université de Lund / examinateur

Erven Rohou
Directeur de recherche INRIA / examinateur

Christophe Wolinski
Professeur Université de Rennes 1 / directeur de thèse

François Charot
Chargé de recherche INRIA / co-directeur de thèse

Remerciements

Je souhaite remercier Christophe Wolinski, professeur et directeur de l'ESIR, ainsi que François Charot, chargé de recherche à l'INRIA, qui ont encadré mes travaux durant ces trois années.

Je tiens également à remercier les différents membres du jury. Merci à Albert Cohen, directeur de recherche à l'INRIA, ainsi qu'à Pierre Boulet, professeur à l'université de Lille 1, d'avoir accepté de rapporter cette thèse. Leurs analyses critiques et leur enthousiasme quant à l'intérêt de ces travaux constituent un réel accomplissement personnel. Je remercie également Vincent Lorquet, architecte compilateur à STMicrolelectronics, Erven Rohou, directeur de recherche à l'INRIA, Krzysztof Kuchcinski, professeur à l'Université de Lünd en Suède pour leur participation au jury. Je remercie tout particulièrement Tanguy Risset, professeur à l'INSA de Lyon, de m'avoir fait l'honneur d'en accepter la présidence.

Cette thèse doit beaucoup à Steven Derrien, professeur à l'ISTIC, qui en plus des multiples discussions en N-dimensions, m'a beaucoup aidé lors de l'élaboration et de la rédaction de la partie basée sur le modèle polyédrique. Une mention spéciale pour être arrivé à peine vingt minutes en retard à ma soutenance après un parcours semé d'embûches et de plus de dix mille kilomètres.

Je remercie évidemment tous les autres membres de l'équipe CAIRN pour les multiples collaborations et discussions scientifiques (ou pas) de ces cinq années passées à l'IRISA. Parmi ces membres, je pense plus particulièrement aux (ex-)doctorants Kevin, Erwann, Antoine, Muhammad Adeel. Je remercie également Nadia, Laurent, Maxime et Jérémie pour leur aide technique, scientifique ou administrative et tout simplement pour leur compagnie.

J'adresse de chaleureux remerciements à ma famille et à mes amis qui m'ont encouragé à commencer puis aidé à achever cette thèse. Merci notamment à David et à ma mère pour la relecture complète du présent manuscrit (qui s'avère être particulièrement long si l'on y comprend pas grand-chose).

Enfin, je remercie particulièrement Stéphanie qui a supporté tous les moments difficiles et la disponibilité particulièrement réduite d'un thésard.

Table des matières

Introduction

Systèmes embarqués à hautes performances

L'omniprésence actuelle des systèmes numériques n'est possible que par leur capacité à embarquer toujours plus de puissance de calcul dans des matériels dont la taille est pourtant de plus en plus réduite. Ainsi, il est aujourd'hui courant de trouver des systèmes d'exploitation multitâches dans des appareils portables capables d'encoder, en temps réel, des flux vidéos en haute définition alors que cette tâche nécessite à elle seule plus de 300 Gigas opérations par seconde.

Cette puissance de calcul embarquée ne serait pas envisageable si les architectures matérielles déployées se contentaient de multiplier les transistors pour obtenir des processeurs plus puissants, capables de répondre aux besoins toujours croissants des usages et des applications. De plus, comme le montre la figure 1, la finesse de gravure des transistors participe de moins en moins aux améliorations des performances des processeurs. En effet, le graphique montre que, depuis 2005, ce sont les autres innovations matérielles qui y contribuent le plus. De manière générale, les contraintes physiques et énergétiques [1] constituent un réel problème transversal à tous les systèmes numériques récents. Ces contraintes sont évidemment d'autant plus fortes pour les systèmes embarqués pour lesquels l'énergie n'est pas qu'un simple objectif d'optimisation mais également une ressource limitée.

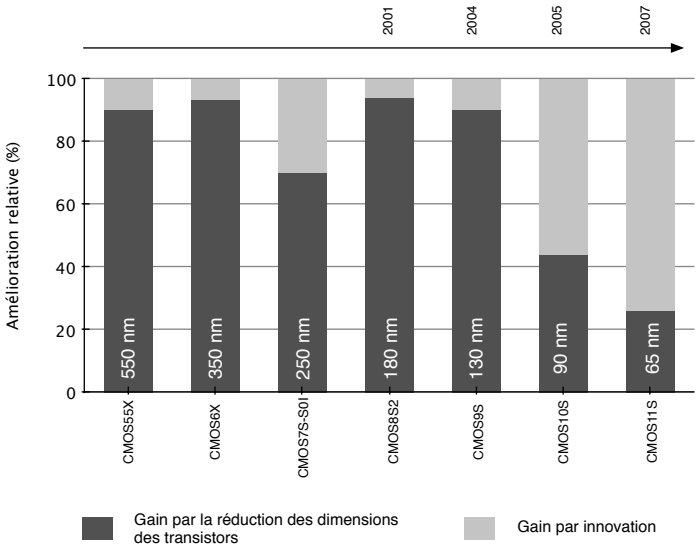


FIGURE 1 – Amélioration des performances des transistors pour les processeurs IBM [175].

Une réponse possible à cette problématique est de concevoir des circuits dédiés à des applications spécifiques (ASIC [2]) qui seront beaucoup plus efficaces qu'un processeur généraliste. Cependant, les coûts de conception et de vérification de tels produits sont généralement trop élevés pour répondre

1. En 2010 les fermes de serveurs représentaient entre 1,1% et 1,5% de la consommation électrique mondiale
2. Application Specific Integrated Circuit

aux fortes pressions du marché où un nouvel appareil est souvent considéré comme étant obsolète en moins d'un an.

La tendance actuelle [97] est plutôt de profiter des progrès en microélectronique pour concevoir des puces qui forment des systèmes numériques hétérogènes complets (SoC [3]) adaptés aux exigences et contraintes d'une gamme de produits. La figure 2 illustre un schéma de conception standard d'un SoC où des processeurs généralistes sont connectés à des accélérateurs matériels embarqués dans le SoC. L'idée sous-jacente aux SoC est de capitaliser des composants matériels (IP [4]) conçus et vérifiés pour une exécution efficace de tâches spécifiques et récurrentes (e.g., encodeur et décodeur H.264). La nature de ces composants est donc très variée (e.g., processeurs généralistes, accélérateurs dédiés, mémoires, etc.) et la conception d'un SoC s'apparente alors à un assemblage des IP les plus adaptés à un type de produit et aux applications qu'il cible. Ainsi, des outils tels que NX builder [141] permettent d'accélérer de 25% la conception d'un SoC et de plus de 75% la conception de ses différentes variations. Un tel outil repose sur une exploration facilitée par la mise à disposition d'IP dans une bibliothèque consultable par le concepteur.

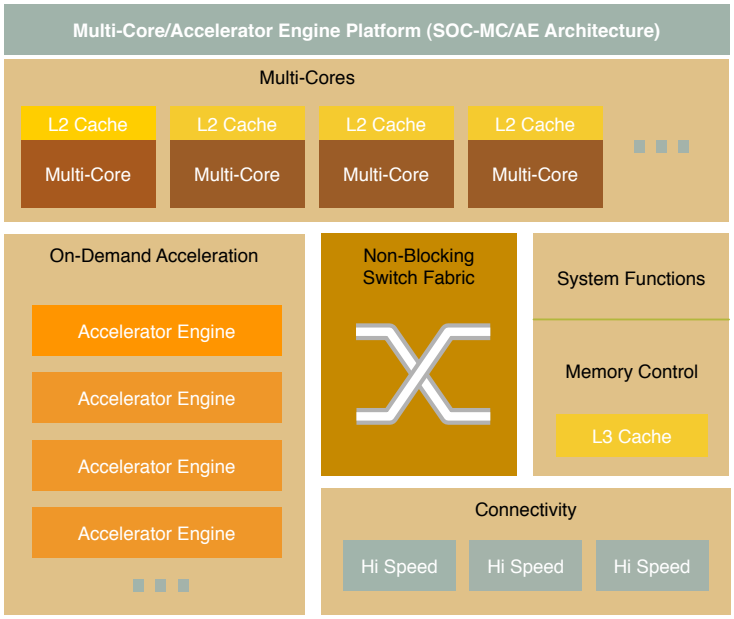


FIGURE 2 – Patron de conception d'un SoC [97].

Dans cette thèse, nous nous sommes intéressé à la conception d'IP qui offrent un compromis entre les performances d'un ASIC et la flexibilité d'un processeur programmable. Ainsi, ces IP ciblent une application ou une famille d'applications et on parle alors de *processeurs spécialisés* ou de *processeurs à jeu d'instructions spécifique* (ASIP [5]). L'architecture matérielle améliorera les performances par rapport à un processeur généraliste tout en limitant sa surface matérielle en étant reconfigurable fonctionnellement.

3. System on Chip
4. Intellectual Property
5. Application Specific Instruction Set Processor

Conception de processeurs spécialisés

Les processeurs spécialisés constituent une famille d'accélérateurs matériels qui s'intègre dans le spectre des architectures entre le processeur généraliste et le chemin de données dédié (cf. figure 3). L'atteinte d'un compromis entre les performances et la flexibilité du processeur repose sur une analyse fine d'une ou de plusieurs applications afin d'en extraire les opérations critiques. L'exécution de ces opérations sur un matériel dédié permettra d'augmenter significativement les performances de l'application tout en réduisant sa consommation énergétique.

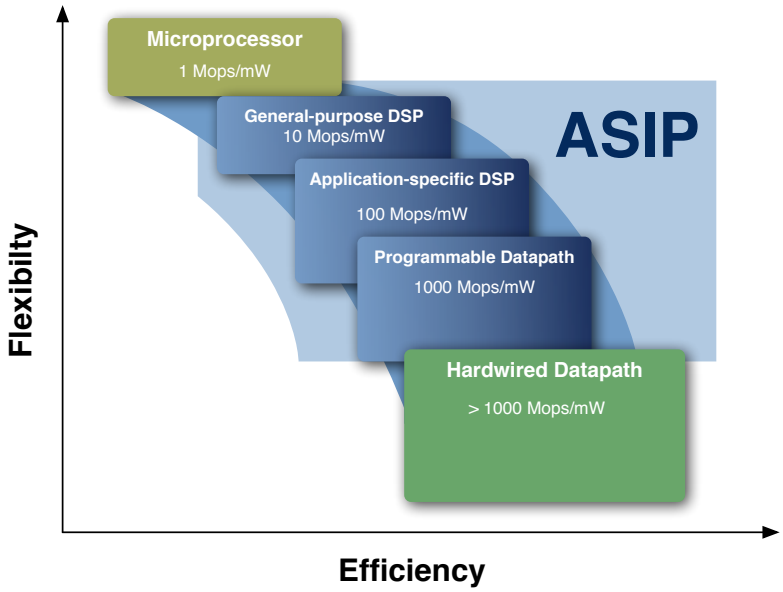


FIGURE 3 – Intégration des ASIP dans le spectre des architectures matérielles [1].

À la différence d'un ASIC, les processeurs spécialisés sont reconfigurables fonctionnellement. Les groupes d'opérations identifiés dans l'application cible constitueront alors autant d'instructions spécifiques au processeur qui coderont chacune des fonctionnalités du processeur. Dès lors, l'espace des possibilités architecturales (e.g., parallélisme, capacité mémoire, etc.) est énorme et la sélection des meilleures instructions spécifiques dépendra fortement des caractéristiques retenues par le concepteur.

D'autre part, pour exploiter un ASIP, le code haut niveau d'une application (e.g., langage C) doit être traduit en une séquence d'instructions compréhensible par le processeur spécialisé. Ce processus présente cependant une différence fondamentale avec les compilateurs usuels : **le jeu d'instructions est à définir lors de la compilation**. En effet, comment décrire des instructions spécifiques pertinentes sans avoir vérifié au préalable qu'elles sont adaptées à l'application ciblée ? Réciproquement, comment sélectionner les instructions lors de la compilation sans les connaître au préalable ? Ces deux questions paradoxales remettent en cause l'étape de génération de code machine d'un flot de compilation habituel.

La conception d'un ASIP se doit donc d'inclure celle d'un compilateur adapté au processeur spécifique. Dès lors, les étapes de conception d'un ASIP peuvent apparaître comme étant aussi difficiles que celles d'un ASIC : exploration architecturale, partitionnement logiciel/matériel, extraction du jeu d'instructions spécifiques, synthèse matérielle de l'architecture, production du compilateur adapté et

enfin vérification. Cependant, la plupart des tâches précédentes peuvent être en grande partie automatisées afin d'être intégrées dans un **processus d'exploration semi-automatique** guidé par les choix du concepteur [1, 47]. À chaque itération de l'exploration, les outils identifieront des instructions spécifiques pertinentes à une application puis génèreront à la fois un compilateur, un simulateur et la description matérielle de l'ASIP. Ces outils permettront d'évaluer la qualité de l'architecture et laisseront la possibilité au concepteur de raffiner les caractéristiques de l'architecture ainsi que de l'application dans l'optique d'une nouvelle itération qui améliorera la qualité de la solution.

Les *processeurs extensibles* sont des ASIP dont la particularité est de coupler une extension matérielle à un processeur généraliste (GPP [6]). La figure 4 illustre un exemple de processeur extensible : l'extension matérielle est fortement couplée au chemin de données de l'ALU [7] du processeur NiosII d'Altera. Ainsi, le processeur spécialisé sera capable d'exécuter n'importe quelle application avec le jeu d'instructions standard du GPP hôte ; il pourra également utiliser la logique disponible dans l'extension pour des instructions spécialisées (ISE [8]) à une ou plusieurs applications. On parle alors de *conception partielle* en opposition à la *conception complète* d'un ASIP qui nécessite de définir l'intégralité du jeu d'instructions du processeur ainsi que sa micro-architecture. Dans le cas d'une conception partielle, l'objectif est donc d'identifier et de sélectionner les instructions les plus pertinentes, on parle alors d'**extension du jeu d'instructions** d'un processeur.

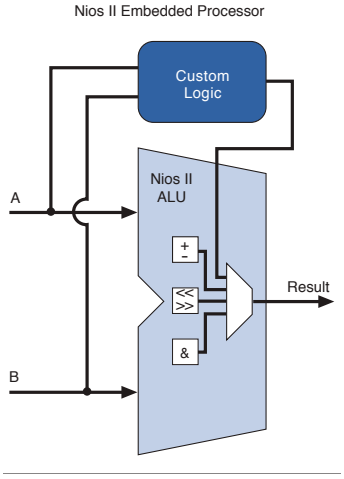


FIGURE 4 – Exemple de processeur extensible : le NiosII d'Altera [7].

Outre leur flexibilité, les processeurs extensibles ont également l'intérêt de simplifier énormément le flot de compilation. En effet, ces processeurs disposent déjà d'un compilateur efficace et vérifié qui peut être utilisé pour compiler les zones de code ne contenant aucune ISE et qui ne seront donc pas exécutées par l'extension matérielle.

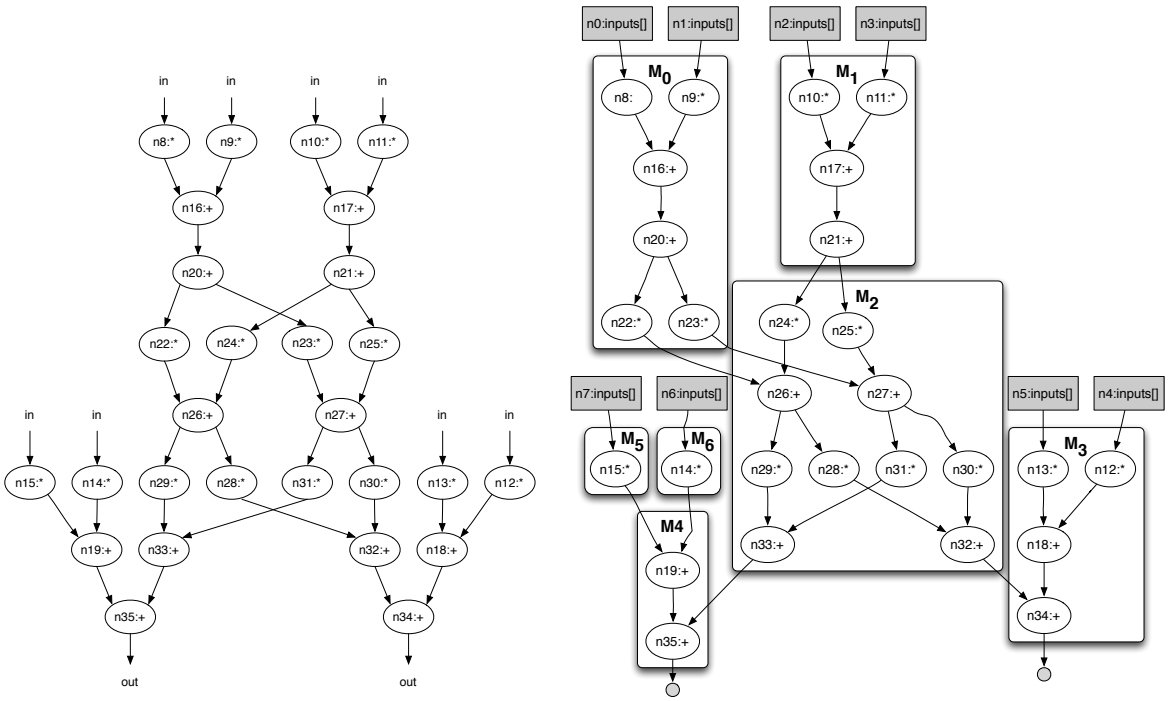
6. General Purpose Processor
7. Arithmetic and Logical Unit
8. Instruction Set Extension

Exemple d'une application exécutée sur un processeur extensible

Afin d'illustrer l'intérêt et le comportement d'un processeur extensible, nous allons étudier une application exécutée sur un NiosII dont le jeu d'instructions a été étendu. L'application est un filtre autorégressif (ARF) issu de la suite MediaBench [113] dont le comportement est illustré par un graphe flot de données acyclique (DFG [9]) dans la figure 5.A. Chacun des 28 nœuds correspond à une opération (multiplication ou addition) et l'exécution de ce comportement sur le NiosII (cf. figure 4) nécessite 28 instructions de son jeu standard. Les instructions spécialisées sélectionnées comportent au plus quatre opérandes et deux résultats.

Les 7 instructions sélectionnées lors de la compilation sont illustrées par la figure 5.B et sont nommées de M_0 à M_6. Les groupements d'opérations qui ne contiennent qu'un seul nœud sont des instructions standard du processeur et seront exécutés par son ALU, les autres correspondent à des instructions spécialisées associées à un chemin de données spécifique mis en œuvre dans l'extension matérielle.

Les durées des instructions spécialisées ont été calculées pour un NiosII cadencé à 150 Mhz, elles durent deux cycles au maximum. L'ordonnancement obtenu pour l'ensemble des instructions sélectionnées, qui est détaillé par la figure 6, accélère l'application d'un facteur 3. En effet, seulement 9 cycles sont nécessaires au lieu des 28 pour une exécution standard (on considère que chaque instruction du GPP ne dure qu'un seul cycle).



A) DFG original B) Graphe couvert par les instructions spécialisées

FIGURE 5 – Sélection d'instructions spécialisées pour l'application ARF.

9. Dataflow Graph

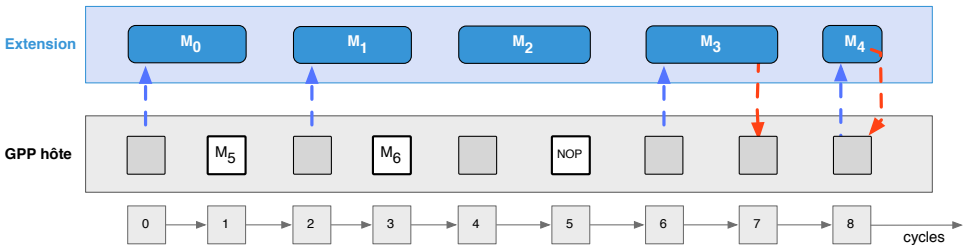


FIGURE 6 – Ordonnancement du graphe couvert de l'application ARF.

Dans l'ordonnancement de la figure 6, les flèches entre le GPP et l'extension matérielle symbolisent des transferts de données (i.e., opérandes ou résultats) et les rectangles indiquent l'occupation de la ressource d'exécution (i.e., GPP ou extension) respective de chaque instruction. On remarque que l'instruction M_2 ne communique pas avec le GPP alors qu'elle possède pourtant quatre opérandes et deux résultats. Ce comportement est issu du choix des caractéristiques de l'extension : les données produites sur l'extension sont conservées dans des registres si elles sont réutilisées plus tard par d'autres instructions spécialisées. D'autre part, on considère ici que le GPP peut exécuter une instruction en parallèle de l'extension matérielle (e.g., aux cycles 1 et 3) si celui-ci n'est pas occupé à lancer l'exécution d'une instruction spécialisée ou encore à envoyer/recevoir des données.

Contributions

Au cours de cette thèse, nous nous sommes intéressé à la compilation pour des processeurs extensibles. Comme évoqué précédemment, celle-ci diffère de la compilation pour processeurs généralistes puisque le jeu d'instructions est à la fois identifié et utilisé par cette étape. Cette différence majeure complique considérablement la compilation qui doit alors répondre à de complexes problèmes d'optimisations et s'intégrer dans une infrastructure logicielle hétérogène du fait de la multiplicité des thématiques abordées.

Sélection et ordonnancement d'instructions spécialisées

La majorité des techniques d'extension de jeu d'instructions dissocie l'étape de sélection de celle d'ordonnancement des instructions spécialisées. Les algorithmes s'appuient alors sur un oracle (e.g., sélection des instructions qui contiennent le plus de nœuds, sélection des instructions dont le comportement sera le plus accéléré par le matériel, etc.) qui prédira les conséquences de la sélection sur l'ordonnancement produit ultérieurement par le compilateur.

Cette approche est adaptée à des instructions spécialisées ne durant qu'un seul cycle car, dans ce cas, la prédiction sera juste. Cependant, pour des instructions multicycles, une sélection qui ne tient pas compte de l'ordonnancement risque de privilégier des instructions qui seront, a posteriori, moins intéressantes que d'autres pouvant notamment s'exécuter en parallèle du GPP (e.g., figure 6).

Nous proposons d'effectuer la **sélection et l'ordonnancement des instructions spécialisées en une unique étape d'optimisation**. Ce problème complexe est NP-complet et nous utilisons la *programmation par contraintes* pour l'énoncer et le résoudre. Cette méthodologie nous permet d'être

particulièrement flexible sur les caractéristiques du processeur extensible et deux modèles d'architectures ont été envisagés. Le premier est un processeur extensible fortement couplé à une extension matérielle disposant de registres internes pour mémoriser les données qui y sont produites. Le deuxième modèle s'appuie sur un modèle d'exécution VLIW [10] où une même instruction spécialisée configure plusieurs unités de traitement s'exécutant en parallèle.

Nos résultats expérimentaux montrent que les approches mises en œuvre sont à même de traiter des graphes de quelques centaines de nœuds en un temps raisonnable [11] (inférieur à une dizaine de secondes pour la majorité des cas). Les accélérations obtenues sont particulièrement intéressantes pour des applications disposant d'un degré de parallélisme d'instructions élevé.

Optimisation de code guidée par la sélection d'instructions spécialisées

Les flots existants de conception et de compilation ASIP ne s'intéressent généralement pas aux interactions entre les optimisations du compilateur et la sélection des instructions spécialisées. Pourtant, cette sélection est fortement dépendante de la structure du code analysé. Ainsi, fusionner deux nids de boucles permettra de faire apparaître plus d'opportunités de regroupements d'opérations et favorisera donc la qualité des instructions spécialisées sélectionnées.

Le peu de travaux qui traitent cette problématique explore itérativement des combinaisons de transformations de code afin de comparer les résultats obtenus après sélection des instructions.

Nous proposons une approche originale qui suit la démarche inverse : c'est **la sélection des instructions spécialisées qui conditionne les transformations effectuées sur le code original**. La technique présentée s'appuie sur l'expressivité du *modèle polyédrique* qui permet de décrire une combinaison complexe de transformations de nids de boucles (e.g., rotation, fusion et fission de boucles, etc.) par un ensemble de fonctions affines [194].

L'objectif de notre approche est de résoudre un problème d'optimisation (exprimé avec la programmation par contraintes) dont la résolution identifiera à la fois les instructions sélectionnées et les transformations affines qui les feront apparaître après modification du code. Ainsi, nous permettons d'identifier des instructions spécialisées en regroupant des opérations se trouvant pourtant à l'origine dans des nids de boucles différents et qui seront alors fusionnés. De plus, l'approche permet d'imposer de multiples contraintes supplémentaires afin de respecter des exigences de performance ou des limitations architecturales. Dans le cadre de cette thèse, nous imposerons que toutes **les instructions spécialisées sélectionnées soient vectorisables**.

Intégration de méthodologies logicielles dans la conception d'outils pour la compilation optimisante

La variété des thèmes abordés par un compilateur extensible favorise le recours à de multiples représentations intermédiaires d'un même programme. Certaines sont adaptées à des optimisations standard d'un compilateur (e.g., propagation de constantes, élimination de code mort, etc.) et d'autres sont spécifiques à l'extension de jeux d'instructions (e.g., synthèse matérielle de l'extension). La multiplicité de ces représentations intermédiaires entraîne un coût de développement et de maintenance

10. Very Long Instruction Word
11. dans le contexte de la compilation/conception pour un ASIP

important associé à des tâches fastidieuses et transversales aux représentations (e.g., génération de code, sérialisation et désérialisation des représentations intermédiaires, utilisation de logiciels externes, etc.).

Au cours de cette thèse, nous nous sommes intéressé à l'**intégration de l'ingénierie dirigée par les modèles (IDM) au cœur d'un compilateur optimisant**. Dès lors, toutes les représentations intermédiaires sont décrites dans un même langage qui offre la possibilité de se connecter à un ensemble d'outils existants et dédiés aux tâches transversales susmentionnées.

Nous montrons que ce paradigme de développement, s'il ne constitue pas en soi une réponse aux problèmes d'optimisation rencontrés dans un compilateur, simplifie la mise en œuvre et l'intégration des multiples composants d'un compilateur optimisant. Ainsi, l'émergence des architectures hétérogènes et spécialisées a pour conséquence de favoriser l'apparition de langages dédiés ou de dialectes pour guider les optimisations du compilateur. L'IDM offre notamment des solutions avancées qui simplifient à la fois la description et l'intégration de ces langages au sein d'une infrastructure existante.

Dans l'optique de faciliter la conception d'optimisations complexes, nous proposons un **environnement de modélisation de problèmes de programmation par contraintes** (ARCAdE [12]) qui s'appuie sur des concepts avancés de génie logiciel et de programmation-objet. En effet, les techniques d'extension de jeu d'instructions présentées dans cette thèse s'appuient sur la modélisation et la résolution de tels problèmes. La programmation par contraintes (de même que l'ILP [13]) dissocie l'énoncé du problème de sa résolution et les différentes composantes d'un problème sont modulaires par nature. Ce constat nous a amené à concevoir un environnement et un langage dédié qui simplifie la formulation de problèmes complexes en composant très simplement des sous-problèmes. La modélisation d'un problème s'apparente alors, intuitivement, à un assemblage de différentes pièces de « puzzle » qui sont indépendantes et réutilisables dans de multiples contextes.

Structure du document

Ce mémoire comporte huit chapitres structurés en trois parties.

Première partie : extension de jeu d'instructions pour processeurs spécialisés

Le premier chapitre détaille la problématique de la compilation pour des processeurs extensibles et résume les principales techniques existantes pour son automatisation.

Le deuxième chapitre présente les principes et concepts de la programmation par contraintes utilisée dans nos techniques de sélection d'instructions.

La première contribution de cette thèse est l'objet du troisième chapitre qui propose une nouvelle approche réalisant à la fois la sélection et l'ordonnancement d'instructions spécialisées pour chaque bloc de base d'une application C. Cette technique est intégrée dans un flot de compilation et de conception semi-automatique s'appuyant sur l'infrastructure de compilation GeCoS [14].

12. Aspect Oriented Constraint Programming Environment
13. Integer Linear Programming
14. http://gecos.inria.gforge.fr

Deuxième partie : sélection d'instructions spécialisées et optimisation de code

Le modèle polyédrique est une abstraction mathématique permettant de transformer les nids de boucles d'un programme par de simples fonctions affines (on parle également d'*ordonnancements affines*). L'objectif du quatrième chapitre est d'en résumer les concepts et différentes applications dans le domaine de la compilation optimisante.

Le cinquième chapitre étudie les possibilités de jonction entre le modèle polyédrique et la programmation par contraintes. L'objectif est de mettre en œuvre une contrainte polyédrique qui permettra de modéliser et de résoudre des problèmes d'ordonnancement affine mêlés à des contraintes non linéaires.

Le sixième chapitre s'intéresse aux problématiques d'optimisation de nids de boucles dans le contexte de l'extension de jeu d'instructions. Nous y détaillons notre seconde contribution principale qui détermine les transformations affines d'un programme permettant de sélectionner des instructions spécialisées vectorisables. L'algorithme proposé s'appuie à la fois sur la programmation par contraintes et sur le modèle polyédrique pour énoncer un problème d'optimisation non linéaire mais qui repose sur les *ordonnancements affines structurés* de Feautrier [57].

Troisième partie : intégration de méthodologies logicielles dans la conception d'outils de compilation optimisante

Le septième chapitre propose une réflexion et un retour d'expérience sur l'intégration de l'ingénierie dirigée par les modèles dans la conception et la mise en œuvre d'un compilateur optimisant. Cette expérience est issue de nos contributions à l'infrastructure de compilation GeCoS et des multiples outils qui ont été développés pour simplifier les tâches transversales aux différentes représentations intermédiaires.

Enfin, le huitième chapitre introduit ARCAdE, un nouvel environnement dédié à la modélisation modulaire de problèmes d'optimisation exprimés avec la programmation par contraintes. La mise en œuvre de cet environnement repose également sur l'IDM et permet de décrire simplement, dans un langage dédié, des sous-problèmes qui peuvent être combinés en de nouveaux problèmes complexes. Cet environnement a été utilisé pour modéliser les différentes techniques de sélection d'instructions spécialisées qui sont alors décrites en quelques centaines de lignes.

Première partie

Extension de jeu d'instructions pour processeurs spécialisés

Chapitre 1

Contexte et état de l'art sur l'extension de jeux d'instructions

Sommaire

1.1 Compilation pour processeurs spécialisés

1.1.1 Processeur spécialisé extensible

Les contraintes matérielles des systèmes embarqués sont parfois trop fortes pour que des applications hautement consommatrices de puissance de calcul puissent être convenablement traitées par des processeurs généralistes (GPP [1]). Dans ce contexte, il est naturel de chercher à concevoir des accélérateurs matériels spécifiques qui amélioreront les performances et la consommation énergétique d'applications ciblées. Les processeurs ASIP [2] accélèrent des applications spécifiques et reposent sur la définition d'un jeu d'instructions dédié à ces applications. Chacune de ces instructions spécifiques correspond alors à la reconfiguration fonctionnelle de l'architecture dont le chemin de données effectuera plus efficacement le traitement d'un sous-ensemble des opérations de l'application analysée.

Les processeurs extensibles sont des cas particuliers de la famille des ASIP. Ils offrent un compromis entre les performances d'un processeur dédié à une unique application et la polyvalence des processeurs généralistes. En effet, un processeur extensible est composé d'un GPP (généralement de type RISC [3]) couplé à une extension matérielle dédiée au traitement des parties critiques d'applications spécifiques. Les instructions supportées par le processeur sont alors celles du GPP hôte ainsi que des **instructions spécialisées** qui sont conçues pour améliorer l'efficacité du traitement des zones critiques de certaines applications. La figure 1.1 illustre le couplage d'une extension matérielle au processeur extensible NiosII [7] ; chaque instruction spécialisée qui y sera exécutée pourra lire deux opérandes dans la file de registres du processeur hôte et y enregistrer un résultat.

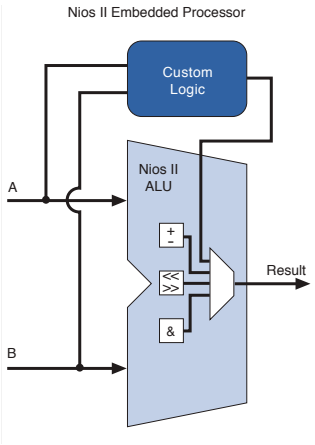


Figure 1.1 – Vue simplifiée de l'architecture du NiosII. Une extension matérielle est fortement couplée à l'UAL du processeur hôte [7].

Dans un contexte de conception de SoC [4], les processeurs extensibles permettent de concevoir rapidement de nouvelles IP [5] en réduisant notamment leurs coûts de vérification. En effet, l'architecture du GPP hôte d'un processeur extensible est déjà vérifiée et dispose généralement d'une chaîne de compilation existante qui simplifie considérablement le portage d'une application. Ces caractéristiques

1. General Purpose Processor
2. Application Specific Instruction Set Processor
3. Reduced Instruction Set Processor
4. System On Chip
5. Intellectual Property

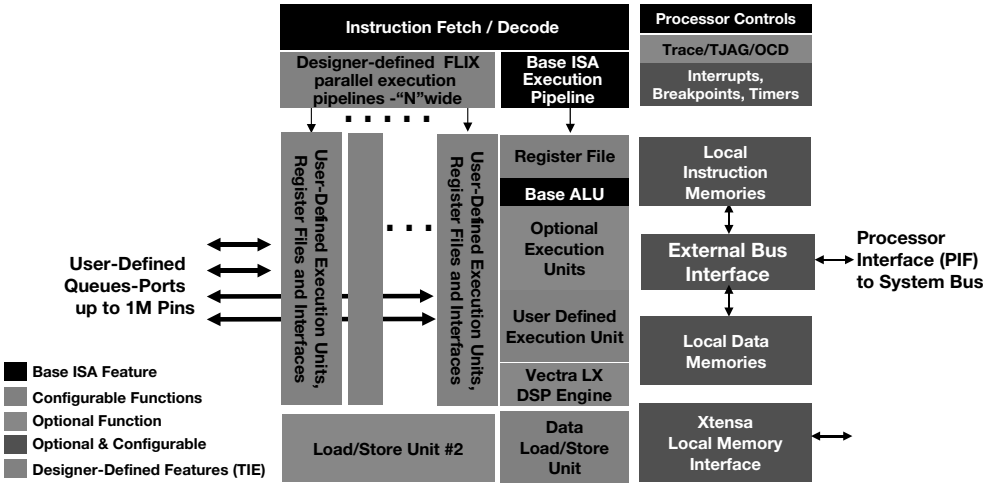


FIGURE 1.2 – Options de configuration du Xtensa LX [178].

s'avèrent particulièrement intéressantes pour la conception de SoC où 60% du temps d'ingénierie est consacré à la seule vérification de l'architecture [92]. De plus, les processeurs extensibles offrent généralement la possibilité d'adapter l'architecture aux besoins du SoC en configurant le GPP hôte : profondeur du pipeline d'exécution, ressources fonctionnelles disponibles (e.g., présence d'un multiplieur, unité de calcul flottant), etc. C'est le cas, par exemple, du processeur Xtensa LX de Tensilica [178] dont les multiples options de configuration sont résumées dans la figure 1.2. Il est notamment possible de faire varier la profondeur du pipeline d'exécution entre 5 et 7 étages, de modifier la taille de la file de registres, d'utiliser une seconde unité de chargement/enregistrement dans la mémoire ou encore d'ajouter des unités fonctionnelles supplémentaires (multiplieurs 16 ou 32 bits, MAC [6], etc.).

Les architectures existantes de processeurs extensibles se différencient notamment par leur technologie de mise en œuvre.

- Processeur synthétisé. Après personnalisation de l'architecture du processeur pour cibler une ou plusieurs applications, celle-ci est synthétisée en un circuit dédié qui peut alors être intégré à un SoC. Dans cette catégorie, on compte le processeur Xtensa de Tensilica [178], ARC de Synopsys [176].
- FPGA [7]. Les processeurs extensibles peuvent également être mis en œuvre en utilisant la logique programmable des FPGA. Ces processeurs *soft-core* permettent un prototypage rapide au prix d'une moindre efficacité que leurs homologues ASIC : il sont généralement 3 à 5 fois moins rapides et peuvent occuper jusqu'à 35 fois plus de surface [31, 112]. Les *soft-core* les plus connus proviennent des sociétés Altera et Xilinx qui proposent respectivement le NIOSII [7] et le Microblaze [199].
- Hybrides. Certains processeurs extensibles s'appuient sur un processeur ASIC couplé à une logique programmable FPGA pour profiter des performances ASIC tout en étant à même de recibler les instructions spécialisées sans avoir à produire un nouveau circuit. Ainsi, le processeur S6000 de Stretch [174] s'appuie sur un processeur Xtensa LX couplé à un module

6. Multiply Accumulate
7. Field Programmable Gate Array

reconfigurable appelé ISEF [8] qui est chargé de l'exécution des instructions spécialisées.

1.1.2 Compilation

L'exécution d'une application sur un processeur nécessite une étape de compilation qui transforme le code de cette application en une séquence d'instructions compréhensible par le processeur. Dans cette sous-section, nous expliciterons les principales étapes d'un flot de compilation en faisant notamment apparaître les spécificités de la compilation pour des processeurs spécialisés.

1.1.2.1 Représentations intermédiaires

Tout flot de compilation se base sur une ou plusieurs représentations intermédiaires pour analyser le code des applications et produire un binaire exploitable par le processeur. Dans ce paragraphe, nous présentons succinctement quelques-unes de ces représentations.

CDFG La représentation intermédiaire la plus commune est le *Control Data Flow Graph* qui exprime le comportement et la structure de contrôle d'un programme à travers un graphe où les nœuds correspondent aux blocs de base d'un programme et les liens à ses différents chemins de contrôle. Chaque bloc de base contient un comportement flot de données où les opérations sont effectuées de manière inconditionnelle et sont indépendantes de la cible matérielle. Les liens de contrôle représentant quant à eux les différentes branches possibles de l'exécution du programme.

DFT Les instructions d'un bloc de base sont souvent représentées sous une forme arborescente (DFT [9]) dans laquelle chaque nœud peut correspondre à une opération ou encore à un symbole du programme. Si un nœud possède des fils, ces derniers indiquent les opérandes de l'opération du nœud. La figure 1.3 montre un exemple de DFT pour l'instruction $d = a + b + c + d$. Les nœuds qui n'ont pas de fils sont des accès aux symboles du programme.

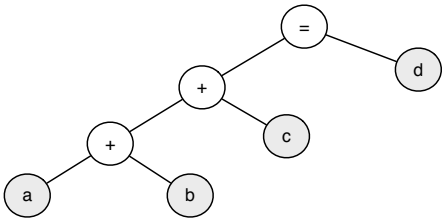


FIGURE 1.3 – Représentation arborescente de l'instruction $d = a + b + c$.

DFG Les calculs effectués dans un bloc de base d'un CDFG peuvent également être représentés par un graphe acyclique dirigé (DAG [10]). La principale différence avec un DFT réside dans le nombre de sorties d'une opération : le résultat d'un même calcul peut être utilisé par différents nœuds et chaque utilisation est symbolisée par un lien. Cette caractéristique est illustrée par la figure 1.4 qui montre le DAG de deux instructions d'un programme C. La seconde instruction utilise le résultat de la première

8. Instruction Set Extension Fabric
9. Dataflow Tree
10. Directed Acyclic Graph

et cette dépendance de données est représentée par le lien entre les deux additions du DAG. La représentation DAG a le mérite de faire apparaître clairement le parallélisme potentiel des opérations : si aucun chemin n'existe entre deux nœuds du DAG, ils peuvent être exécutés simultanément.

```
1 int foo(int a, int b){
2   ...
3   c = a + b;
4   d = a + c;
5   ...
6 }
```

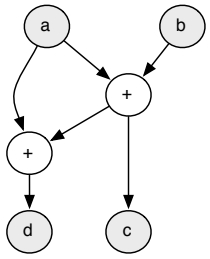


FIGURE 1.4 – Représentation DFG de l'instruction $d = a + b + c$.

HCDG Les graphes hiérarchiques aux dépendances conditionnées (HCDG [11]) offrent une représentation unique des flots de contrôle et de données d'une application. Cette représentation est largement inspirée des programmes synchrones (e.g., Lustre [32], Signal [67], etc.) et en reprend la notion de transition conditionnée par un prédicat logique. Un HCDG diffère d'un CDFG par la hiérarchisation des informations de contrôle facilitant ainsi les opérations liées à la synthèse de haut niveau [109]. Les dépendances de données et de contrôle sont représentées selon une vue typiquement *dataflow* et chaque nœud symbolise soit une donnée, soit une garde qui conditionne l'exécution de ses nœuds fils. Dans un modèle à temps discret, une garde peut être vue comme un ensemble de conditions boo-

```
1  int abs(int a, int b){
2    int c;
3    if(b < a){ //G1
4      c = a - b;
5    }
6    else{ //!G1
7      c = b - a;
8    }
9    return c;
10 }
```

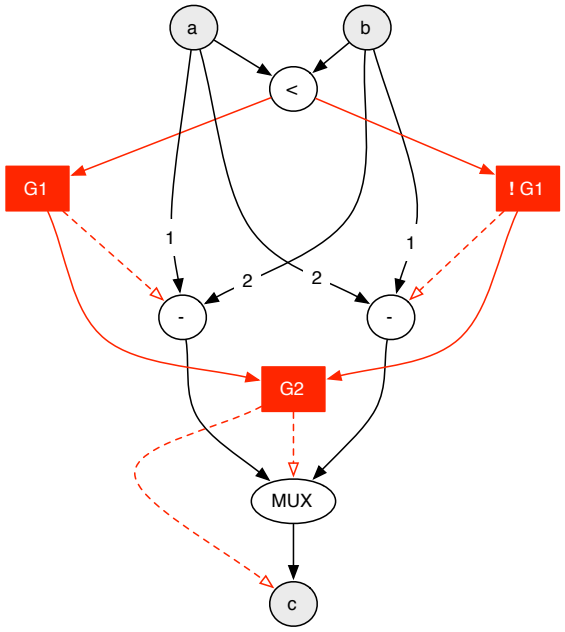


FIGURE 1.5 – Représentation HCDG du calcul de la valeur absolue d'une différence. La garde $G2$ est définie par $G2 = G1 \vee !G1$, elle est donc toujours vraie.

léennes valant VRAI ou FAUX en fonction de l'instant logique. À chaque garde correspond alors une

11. Hierarchical Conditional Dependency Graph

fonction booléenne dont l'évaluation conditionnera l'exécution de ses nœuds fils. Deux gardes (G1 et G2) seront exclusives si leurs ensembles d'instants logiques satisfaisant leurs conditions sont disjoints ($G1 \cap G2 = \emptyset$). Le graphe HCDG utilise le principe SSA (*Static Single Assignement*) et si plusieurs définitions pour un nœud sont possibles, leurs gardes sont alors nécessairement exclusives.

Le calcul d'une valeur absolue est représenté sous forme de HCDG dans la figure 1.5. La garde $G2$ correspond à l'écriture d'une valeur dans la variable c, son expression booléenne ($G1 \vee !G1$) est calculée à partir de celles de $G1$ ($b < a$) et de $!G1$ ($b \geq a$) qui sont mutuellement exclusives. Cette exclusivité est symbolisée par un multiplexeur précédant l'écriture du résultat de la différence dans c, la donnée provient soit de la soustraction conditionnée par $G1$ soit de celle conditionnée par $!G1$.

1.1.2.2 Flot de compilation GPP

La compilation d'un programme pour un processeur généraliste est effectuée en trois étapes illustrées dans la figure 1.6.

1. *Front-end*. Lors de cette étape, le code de l'application est analysé [3] pour construire un AST [12] transformé en une représentation intermédiaire (IR [13]) du programme.
2. Optimisations. Un certain nombre de transformations de l'IR sont opérées afin d'améliorer les performances du programme. Ces optimisations sont généralement indépendantes (e.g., élimination de code mort, propagation de constantes, etc.) de l'architecture cible.
3. Génération du code. Cette dernière étape, qui est chargée de produire un code binaire compatible avec l'architecture, est divisée en trois sous-étapes : 1) Couverture de la représentation intermédiaire par les instructions du jeu du processeur 2) Allocation des registres du processeur utilisés pour mémoriser les données intermédiaires 3) Ordonnancement des instructions sélectionnées.

Un compilateur peut éventuellement être reciblable. Dans ce cas, la description de l'architecture du processeur et celle de son jeu d'instructions sont utilisées lors de l'étape génération de code.

1.1.2.3 Flot de compilation ASIP

Le comportement d'un compilateur pour un ASIP est particulier puisque, par définition, les instructions du processeur sont conçues en fonction des applications qui y seront exécutées. Dans ce cas, à la différence d'un compilateur reciblable, la description de l'architecture est à la fois une entrée et une sortie du flot de compilation. Ainsi, comme illustré par la figure 1.7, les compilateurs ASIP ne se différencient d'un compilateur classique que lors de l'étape de sélection de code : les instructions du processeur GPP sont étendues par de nouvelles instructions spécialisées (ISE [14]) qui détermineront l'architecture de l'extension matérielle couplée au processeur.

Sélection de code et extension du jeu d'instructions d'un processeur L'étape de sélection de code identifie, dans une application, des groupes d'opérations qui seront exécutés sur une extension matérielle. Chacun de ces groupes correspond alors à une ISE et les opérations restantes seront couvertes par le jeu d'instructions standard du processeur.

12. Abstract Syntax Tree
13. Intermediate Representation
14. Instruction Set Extension

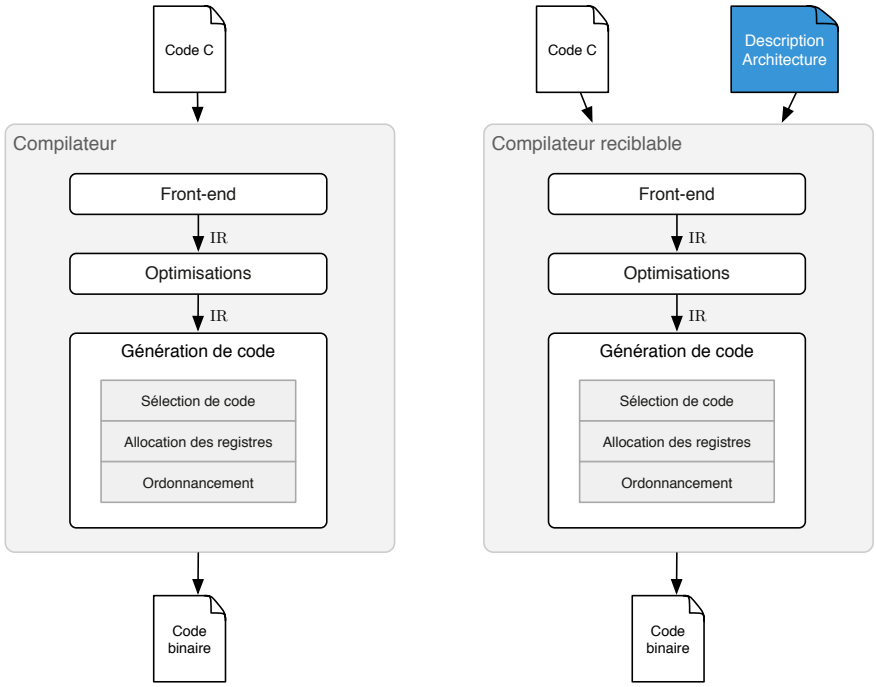


FIGURE 1.6 – Principes des compilateur optimisants.

Le choix de la représentation intermédiaire conditionne les techniques possibles de sélection. Ainsi, pour un GPP, il existe des algorithmes de complexité polynomiale (e.g., Burg [62]) pour couvrir, de manière optimale, les instructions du DFT d'un programme par celles du processeur et qui sont alors exprimées sous forme de motifs arborescents. Cependant, de tels algorithmes ne sont pas adaptés à l'extraction d'instructions spécialisées [119] pour deux raisons. Tout d'abord, les arbres ne permettent pas de décrire des sous expressions communes qui se prêtent pourtant de manière naturelle et efficace à une implémentation matérielle dans un chemin de données. De plus, dans la représentation DFT d'une application, le comportement d'un bloc de base est décomposé en une séquence d'instructions communiquant par des variables temporaires ce qui implique que le parallélisme des opérations ne peut donc être détecté simplement. Il apparaît alors beaucoup plus pertinent de s'intéresser à des représentations intermédiaires de type DAG qui expriment, par construction, la communication d'une même donnée à plusieurs consommateurs et où le parallélisme des opérations est explicite.

Malheureusement, la sélection de code sur un DAG est beaucoup plus difficile que sur un arbre et les algorithmes déterminant une couverture optimale sont NP-complets. Le concept même d'optimalité est loin d'être évident comme dans le cas d'une couverture d'un DFT qui se contente de minimiser la somme des coûts des instructions sélectionnées. En effet, de nombreux critères sont pertinents dans le contexte d'un processeur extensible : la durée d'exécution totale, le nombre d'ISE sélectionnées, la consommation énergétique, la surface occupée par l'extension, etc. On distingue principalement deux familles d'algorithmes pour traiter ce problème.

La première famille d'algorithmes propose des heuristiques adaptant les algorithmes de couverture de DFT à une représentation DAG afin d'en conserver la faible complexité. Ces heuristiques [119, 205, 11] s'appuient sur des règles de coupes d'un DAG qui permettront d'explorer les recompositions alternatives de plusieurs DFT formant alors des instructions DAG.

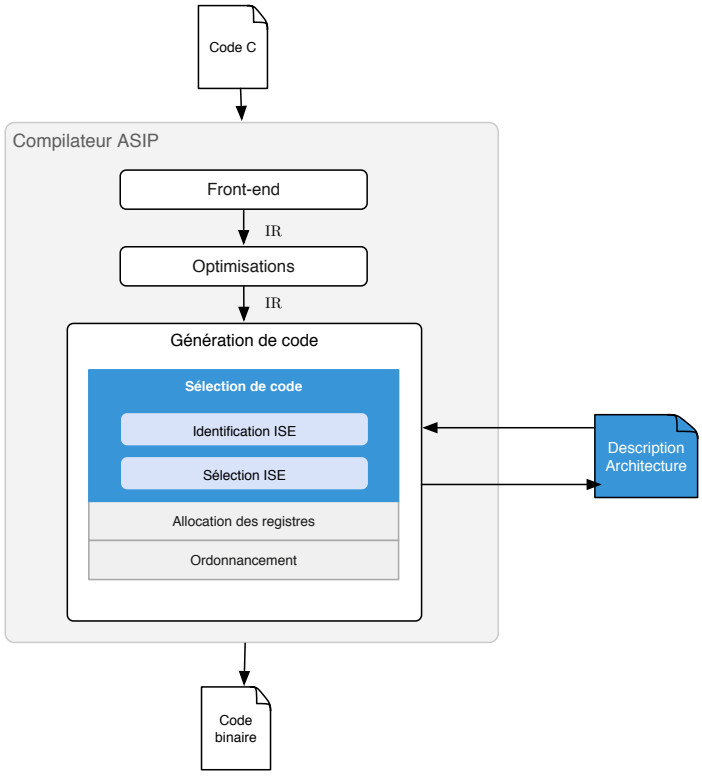


FIGURE 1.7 – Compilation ASIP : l'architecture est à la fois un paramètre et un résultat du flot de compilation.

La seconde famille d'algorithmes repose sur la notion d'**isomorphisme de sous graphes** pour identifier et sélectionner les **occurrences** des instructions du processeur dans le DAG de chaque bloc de base. Chacune des instructions du processeur correspond alors à un **motif** DAG qui peut avoir une ou plusieurs occurrences isomorphes dans un bloc de base. Dans le contexte d'un processeur extensible, deux approches sont possibles pour automatiser ce processus. La première approche repose sur une unique étape qui partitionne chaque DAG en occurrences qui constitueront les instances d'ISE. La seconde approche est divisée en deux étapes successives : identification (i.e., génération ou utilisation d'une bibliothèque existante de motifs) des instances d'ISE potentielles puis sélection de leurs occurrences dans chaque DAG. Ainsi, seuls les motifs des occurrences sélectionnées seront des ISE du processeur extensible. La problématique de la sélection automatisée des ISE d'une application est généralement appelée **extension de jeu d'instructions** et l'état de l'art des algorithmes existants fait l'objet de la section 1.3, page 25.

Quelle que soit la technique utilisée, celle-ci ne tient généralement pas compte de l'ordonnancement et risque de sélectionner des occurrences d'ISE qui ne considèreront pas le parallélisme potentiel (i.e., si l'architecture le permet). En effet, les approches existantes favorisent généralement la taille des ISE en partant du principe que plus celle-ci est élevée plus les opportunités d'accélération sur le matériel dédié seront importantes.

Exploitation des ISE sélectionnées La majorité des approches s'appuie ensuite sur le flot standard de compilation en transformant le DAG couvert de chaque bloc de base en une séquence de DFT.

Pour ce faire, les nœuds contenus dans chacune des occurrences d'ISE sélectionnées sont réduits en un seul nœud codant l'instruction du processeur utilisée. Le compilateur standard du processeur extensible se base ensuite sur cette représentation pour effectuer l'allocation des registres et l'ordonnancement.

Cependant, l'algorithme d'ordonnancement du GPP hôte ne tient généralement pas compte du parallélisme entre l'extension et le GPP hôte. Pour pallier cette insuffisance, il est possible d'utiliser un algorithme d'ordonnancement spécifique généré à partir de la description de l'architecture et du jeu d'instructions du processeur [118, 191, 33] (dans un langage ADL [15] comme LISA [89]) ou encore de n'utiliser l'ordonnancement du compilateur que pour des zones de code où aucune instruction spécialisée n'est utilisée. Dans ce cas, une nouvelle version du code de l'application est générée ou écrite par le concepteur : pour chaque zone optimisée par des ISE, l'ordonnancement des instructions est décrit explicitement par une séquence d'instructions assembleur *en ligne* qui ne sera pas analysée par le compilateur.

1.2 Conception d'une extension matérielle

Concevoir un ASIP consiste, par définition, à analyser une ou plusieurs applications qui correspondront à la cible logicielle du processeur. C'est cette analyse qui permettra d'explorer les multiples possibilités architecturales afin de répondre aux besoins de l'application ainsi qu'aux contraintes matérielles du produit. L'objectif de cette section est de présenter l'intérêt et le principe d'un flot de conception ASIP reposant sur une exploration itérative des nombreux paramètres à évaluer.

1.2.1 Conception par exploration

Quels que soient l'architecture du processeur et le niveau de conception (i.e., complète ou partielle) d'un ASIP, la compilation de l'application sur le processeur constitue le cœur de la problématique. En effet, comme évoqué dans la sous-section 1.1.2, c'est lors de la compilation que sont sélectionnées les instructions du processeur. C'est donc uniquement après cette étape que l'architecture concrète du processeur pourra être définie. Ces informations permettront alors au concepteur d'évaluer la pertinence des instructions sélectionnées et leur adéquation avec les différentes contraintes applicatives et architecturales.

Il est aujourd'hui admis qu'un flot de conception ASIP ne peut pas être entièrement automatisé [92]. En effet, l'espace des possibilités architecturales est trop vaste et complexe pour être entièrement modélisé par des fonctions de coût qui sont pourtant les seuls moyens de quantifier la qualité de choix automatiques. Il s'agit plutôt d'identifier un compromis acceptable, en fonction du contexte applicatif et matériel, entre les critères de performance et les coûts en surface et en consommation. Un tel choix ne peut raisonnablement reposer que sur l'expertise du concepteur et l'objectif des outils est alors d'automatiser la compilation tout en laissant au concepteur la possibilité de définir un cadre architectural en adéquation avec ses exigences et contraintes.

Pour répondre au mieux à cette problématique, les méthodologies de conception d'ASIP s'appuient sur un processus cyclique [77, 99, 47, 74] qui assiste le concepteur à atteindre un compromis satisfaisant par une exploration itérative de l'espace de conception. Les principales étapes de ce processus de

15. Architecture Description Language

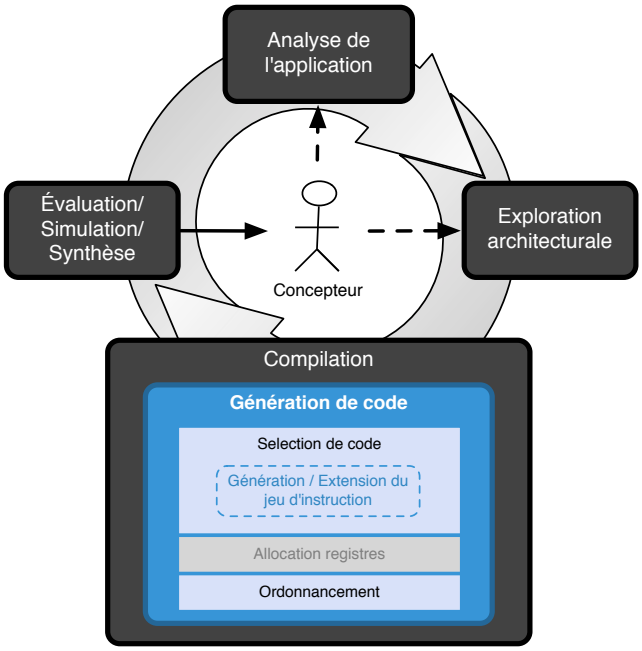


FIGURE 1.8 – Conception assistée d'un ASIP par exploration.

conception peuvent être résumées par la figure 1.8. Tout d'abord, le concepteur analyse l'application pour identifier les parties critiques et en déduire les modèles d'architecture et d'exécution (e.g., SIMD, VLIW, etc.) qui lui semblent les plus adaptés pour le futur processeur spécialisé. Les caractéristiques issues de cette exploration architecturale sont alors utilisées par le compilateur ASIP qui sélectionne les instructions du processeur et produit un code assembleur exploitable par l'architecture. La description matérielle de cette dernière peut être générée à partir du modèle architectural retenu lors de l'étape précédente et des instructions sélectionnées. Les informations obtenues sur les performances de l'application (par simulation [47, 74] ou par évaluation d'une fonction de coût) combinées à celles issues de la synthèse de la description matérielle du processeur permettront au concepteur d'établir si le compromis entre les performances de l'application et les contraintes matérielles est acceptable. Si ce n'est pas le cas, le processus de conception sera raffiné par une nouvelle exploration. Une fois que le concepteur est satisfait des performances et de l'architecture, le processus est terminé.

1.2.2 Couplage de l'extension matérielle au processeur

Dans le cas d'un processeur extensible, la première étape de l'exploration architecturale consiste à choisir un couplage entre l'extension matérielle et le GPP hôte :

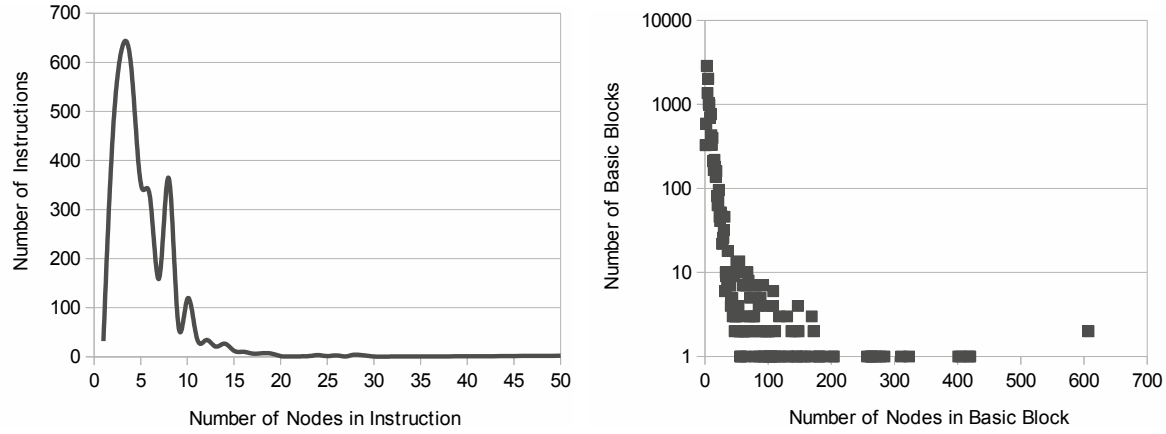
- Couplage fort (e.g., OneChip [192] ou Chimaera [203]). L'extension matérielle est assimilée à une nouvelle unité fonctionnelle connectée au chemin de données du GPP hôte. Elle accède directement à sa file de registres pour lire les opérandes et écrire les résultats des ISE et le coût des communications entre le processeur et l'extension est négligeable.

- Couplage lâche (e.g., Microblaze [199, 22] ou S6000 [174]). L'extension matérielle peut également communiquer avec le processeur hôte par l'intermédiaire de mémoires et éventuellement en utilisant un DMA [16], les ISE sont généralement de taille plus importante que pour un couplage fort. Si une communication utilise une mémoire, son coût n'est plus négligeable et dépend de la technologie utilisée.
- Coprocesseur (e.g., GARP [86] ou ADRES [131]). L'indépendance de l'extension matérielle est élevée : elle dispose généralement de sa propre file de registres ainsi que de mémoires internes qui lui permettent d'exécuter des zones de code pouvant aller jusqu'à des fonctions entières. Ainsi, un coprocesseur peut avoir son propre flot de contrôle et communique très peu avec le processeur hôte.

La liste des architectures citées est loin d'être exhaustive et, pour une étude plus complète, le lecteur est invité à consulter les articles [63, 84, 187] qui référencent et présentent les plus répandues.

1.2.3 Granularité de la spécialisation

La granularité des instructions spécialisées est liée au couplage entre l'extension matérielle et le processeur hôte. Ainsi, des ISE de petite taille sont généralement utilisées dans des extensions matérielles fortement couplées. D'autre part, il est évident que la granularité des ISE joue sur la flexibilité de l'extension : des motifs de taille réduite auront plus de probabilité d'être présents dans de multiples applications en comparaison avec des motifs contenant plus de nœuds et donc plus spécifiques. Réciproquement, des instructions spécialisées dont la taille est importante offrent souvent, mais pas systématiquement (cf. section 1.1.2.3, page 20), plus d'opportunités d'accélération matérielle.



(a) Distribution of size of extension instructions.

(b) Distribution of size of basic blocks (logarithmic scale).

FIGURE 1.9 – Tailles des ISE identifiées par ISEGEN [23] dans les blocs de base de 179 *benchmarks* [137].

Définir la taille des instructions spécialisées constitue donc un autre paramètre supplémentaire de l'espace d'exploration d'un flot de conception ASIP. Lors d'une étude récente, la répartition des ISE en fonction de leurs tailles a été mesurée pour 179 *benchmarks* (télécommunications, multimédia et

16. Direct Memory Access

cryptographie) en utilisant ISEGEN [23], un algorithme glouton qui identifie à chaque itération une ISE qui maximise le profit d'une fonction de mérite. Les résultats de ces expérimentations ont été obtenus pour la représentation intermédiaire de GCC (options de compilation `-O2`) et sont résumés par la figure 1.9. Il y apparaît clairement que le nombre d'ISE de plus de dix nœuds est très faible et que la taille des blocs de bases ne dépasse que rarement 200 nœuds.

1.2.4 Espace d'exploration des architectures de l'extension

Les caractéristiques de l'architecture conditionneront les performances de l'application exécutée sur le processeur spécialisé et de nombreux paramètres sont donc à évaluer par le concepteur lors du processus d'exploration.

- **Parallélisme spatial des opérations**. Les opérations d'une représentation intermédiaire de type DAG peuvent être exécutées en parallèle si elles ne sont pas liées par des chemins de dépendances de données. L'extension comportera alors de multiples ressources matérielles qui occuperont d'autant plus de surface.
- **Modèle d'exécution VLIW**. Un modèle d'exécution VLIW pour les ISE utilisera simultanément plusieurs unités fonctionnelles reconfigurables de l'extension matérielle. Ainsi, le parallélisme spatial est exploité tout en augmentant le degré de réutilisation du matériel.
- **Modèle d'exécution SIMD**. Le matériel de l'extension est dupliqué pour exécuter simultanément plusieurs instances du comportement d'une ISE pour des données différentes. Ce comportement peut être fait à granularité fine (*subword-parallelism*) ou encore entre plusieurs itérations d'un même corps de boucle. On parle également de *vectorisation*.
- **Modèle d'exécution pipelinée**. Des ISE *multicycles* peuvent être pipelinées afin d'augmenter le débit de l'extension matérielle ou encore de réduire l'impact des contraintes issues de son nombre restreint d'entrées et de sorties [150].
- **Approximation des flottants**. Le traitement des flottants peut être simplifié en utilisant une approximation par virgule fixe dont la précision acceptable est dépendante de l'application.
- **Arithmétique des opérations**. Le choix de l'arithmétique utilisée pour mettre en œuvre les opérations de l'application peut augmenter les performances des calculs (par des opérateurs matériels plus efficaces) ou encore améliorer la sécurité du matériel [181].
- **Mémorisation sur l'extension**. L'utilisation de ressources mémoires embarquées sur l'extension réduit la pression sur le processeur hôte en limitant le nombre de communications nécessaires. Ainsi, une donnée intermédiaire produite sur l'extension restera sur l'extension si elle n'est pas utilisée par le processeur.
- **Alimentation de l'extension**. Les techniques de *clock-gating* et de *power-gating* peuvent également être envisagées pour réduire la consommation dynamique et statique de l'extension matérielle si celle-ci n'est pas utilisée lors de l'exécution d'un programme.
- **Réseau d'interconnexions**. La capacité d'un élément de calcul à communiquer avec d'autres ressources de calculs ou de mémorisation conditionne les performances de l'architecture. Ainsi, un réseau de communication très permissif (e.g., *full-crossbar*) facilitera l'allocation des ressources au prix d'une surface matérielle élevée.

1.3 Automatisation de l'extension de jeux d'instructions

Dans cette section, nous nous intéresserons aux principales techniques existantes d'automatisation de la génération de code pour un processeur extensible. Comme il a été évoqué précédemment, cette étape requiert de différencier les opérations qui seront exécutées sur l'extension de celles qui le seront sur le GPP.

1.3.1 Partitionnement d'une application

Le problème d'identification et de sélection d'instructions spécialisées peut être assimilé à un problème de partitionnement logiciel/matériel. Dans ce cas, l'objectif est de choisir pour chaque nœud d'un graphe s'il est exécuté sur le GPP hôte ou bien sur l'extension matérielle. La complexité d'un tel partitionnement pour un graphe de n nœuds est alors de $O(2^n)$ et représente donc un énorme espace d'exploration qui ne peut être traité en un temps raisonnable que pour un faible nombre de nœuds (e.g., pour 20 nœuds cela représente déjà plus d'un million de partitionnements possibles) ou par des heuristiques spécifiques. Pour que les ISE sélectionnées soient ordonnançables, il est impératif de s'assurer de leurs convexités : il ne doit exister aucun chemin à la fois sortant et entrant de chaque ISE.

Une des heuristiques les plus couramment citées est celle proposée dans [14]. Il s'agit d'une approche par séparation et évaluation (BB [17]) qui sélectionne, à chaque itération, l'ISE respectant plusieurs contraintes (i.e., nombre d'entrées et de sorties ainsi que convexité des opérations regroupées) et qui maximise une fonction de mérite. L'heuristique est terminée quand aucun des nœuds ne peut plus être couvert par une nouvelle ISE.

L'approche proposée dans [23] extrait de manière gloutonne les ISE qui maximisent une fonction de mérite et respecte à la fois les contraintes de convexité et celles limitant le nombre d'entrées et de sorties. Elle se base sur l'heuristique de Kernighan-Lin [106] et la construction d'une ISE est itérative : les nœuds du graphe sont ajoutés ou enlevés en fonction de leurs contributions respectives au mérite de l'ISE. Lorsque tous les nœuds ont été évalués et que l'ISE n'évolue plus, elle est considérée comme étant sélectionnée.

D'autres techniques de partitionnement s'appuient sur une formulation dans un problème linéaire en nombre entier (ILP [18]) pour sélectionner de manière gloutonne l'ISE qui maximise/minimise une fonction de mérite/coût. Dans [12], chaque nœud qui n'est pas encore couvert par une ISE est associé à une variable binaire indiquant si ce nœud sera exécuté sur l'extension ou contenu dans une ISE. De plus, des contraintes linéaires limitent le nombre d'entrées et de sorties de l'ISE et garantissent sa convexité dans le graphe. L'objectif est alors de sélectionner, à chaque itération de l'heuristique, l'ISE qui minimise la durée du chemin critique du graphe analysé. Une approche similaire [117] tient compte des coûts de communication entre le processeur hôte et l'extension matérielle : seuls les liens entre le processeur et l'extension ont un coût non négligeable puisque deux nœuds placés sur le matériel communiqueront par un registre interne à l'extension. L'objectif est de sélectionner de manière gloutonne les ISE convexes qui maximiseront la somme des durées logicielles (i.e., opérations exécutées sur le GPP hôte) des nœuds qu'elles contiennent afin de favoriser l'accélération matérielle qu'elles apporteront.

17. Branch and Bound
18. Integer Linear Programming

Les approches de partitionnement profitent généralement d'une dernière étape qui réduit le nombre d'ISE identifiées en analysant leurs isomorphismes. Malgré cette étape supplémentaire, les techniques de partitionnement n'offriront généralement que peu de flexibilité au processeur puisque les ISE auront été générées pour améliorer les performances d'un unique graphe.

1.3.2 Génération d'instructions spécialisées

La génération automatique d'instructions spécialisées vise à construire une bibliothèque d'ISE qui sera utilisée par une étape ultérieure de sélection pour chaque application ciblée. La principale différence de cette méthode par rapport à une sélection par partitionnement est de permettre au concepteur de rechercher un compromis entre les performances des ISE et leurs possibilités de réutilisation dans des familles d'applications. En effet, celui-ci pourra notamment ne sélectionner que des ISE présentes dans plusieurs applications ou encore chercher à minimiser leurs nombres.

Les techniques de génération consistent à énumérer les différents motifs de calcul présents dans un graphe. Ceux-ci peuvent être connexes (i.e., il existe un chemin non orienté entre chaque couple de nœuds du motif) ou non. Dans les deux cas (d'autant plus pour des motifs non connexes), l'espace exhaustif d'énumération est exponentiel et ne peut être raisonnablement parcouru pour des graphes dont la taille dépasse une centaine de nœuds.

1.3.2.1 Motifs sous contraintes d'entrées/sorties

La plupart des approches existantes limitent le nombre d'entrées et de sorties des motifs (i.e., liens entrants et sortants des nœuds d'un motif) afin de restreindre l'espace d'énumération et de le rendre analysable en un temps acceptable (i.e., de l'ordre de quelques minutes pour des graphes de plusieurs centaines de nœuds). Dans ce cas, le nombre maximal de groupements convexes de nœuds d'un graphe $G(V, E)$ nœuds est prouvé [38] comme étant inférieur ou égal à $|V|^{in+out}$ pour des motifs ayant au plus *in* entrées et *out* sorties. Cet espace reste donc, dans le pire cas, trop complexe pour un nombre élevé d'entrées ou de sorties.

Les travaux présentés dans [6] proposent de ne générer que des motifs MISO [19] qui, comme leur nom l'indique, ne comportent qu'une seule sortie. L'algorithme génère, en un temps polynomial, tous les motifs MISO d'un graphe ou encore uniquement ceux dont la taille est maximale (MaxMISO).

Il existe également des heuristiques qui ne génèrent pas exhaustivement tous les motifs existants dans un graphe. La méthode [65] génère des motifs convexes dont le nombre d'entrées et de sorties est limité. Le principe est d'étendre itérativement une ISE qui ne contient au départ qu'un unique nœud graine. Les nœuds de son voisinage sont ensuite ajoutés de manière incrémentale tant qu'il est possible de respecter les différentes contraintes. Dans [28], un algorithme de complexité polynomiale est censé générer l'ensemble exhaustif des motifs convexes également sous contraintes d'entrées/sorties. Cependant cet algorithme est basé sur une restriction qui élimine jusqu'à 25% des regroupements possibles [160].

Les algorithmes [38, 122, 198] exploitent la sémantique de la convexité d'une ISE pour générer, très efficacement, l'ensemble exhaustif des ISE d'un graphe. Le principe est d'énumérer toutes les ISE par une croissance récursive tout en filtrant les nœuds qui ne permettront pas de respecter la convexité.

19. Multiple Inputs Single Output

Les complexités de ces algorithmes ne sont pas évaluées mais les techniques mises en œuvre pour réduire l'espace de recherche permettent généralement de produire l'ensemble des motifs connexes pour des contraintes relativement lâches (6 entrées, 3 sorties) en moins d'une seconde, et ce pour des graphes de plusieurs centaines de nœuds. De plus, ces méthodes peuvent générer des motifs non connexes en un temps raisonnable (ce qu'aucune autre technique ne permet) pour des graphes de plusieurs centaines de nœuds.

La méthode [126] est basée sur la programmation par contraintes (cf. chapitre 2) et modélise un problème de génération exhaustive des motifs respectant des contraintes qui ne sont pas nécessairement linéaires (i.e., différentes d'une forme $a_1x_1 + \ldots + a_kx_k + c \geq 0$). Cette flexibilité se paie par un temps de résolution qui s'avère être de l'ordre de plusieurs minutes pour des graphes de plusieurs centaines de nœuds et avec des contraintes lâches. Cependant, il est intéressant de remarquer que la sémantique de la convexité utilisée dans les approches [38, 122, 198] pourrait être avantageusement modélisée par des contraintes supplémentaires qui permettraient de réduire énormément l'espace de recherche tout en conservant la flexibilité de l'algorithme.

1.3.2.2 Motifs sans contraintes d'entrées/sorties

D'autres approches ne cherchent pas à limiter le nombre d'entrées ou de sorties des ISE identifiées. Ces techniques partent du principe que l'utilisation de *pipelining* dans les ISE réduit l'impact d'un nombre élevé d'entrées (ou de sorties) [150] sur sa durée totale d'exécution. Ainsi, les heuristiques [51] et [64] construisent des ISE en agglomérant, respectivement, des motifs de petite taille ou des MaxMISO. Cette agglomération ne tient pas compte des entrées/sorties et se contente d'assurer la convexité des ISE identifiées.

Les algorithmes [146, 189, 13] n'énumèrent que les ISE convexes, mais pas nécessairement connexes, de taille maximale (MaxMIMO). Les premiers travaux à s'être intéressés aux MaxMIMO se basent sur un graphe de conflit pour les énumérer plus efficacement [146], chaque lien représente deux groupes nœuds qui ne pourront être réunis dans la même ISE sous peine de briser la convexité. Le problème est alors équivalent à l'énumération des plus grands ensembles indépendants du graphe de conflit et la complexité de l'algorithme est en $O(2^{|V_c|})$, où $|V_c|$ correspond au nombre de nœuds du graphe de conflit (et qui est généralement inférieur à celui du graphe). Dans [189] le problème est reformulé sous forme d'une énumération de cliques de poids maximal. Dans [13], il est démontré que le nombre de MaxMIMO d'un graphe G est borné par $2^{|V_f|}$, où V_f est le nombre de nœuds considérés comme étant incompatibles avec une ISE (e.g., accès tableaux) dans G. Ainsi, si aucun nœud n'est interdit dans un graphe, le MaxMIMO est évidemment le graphe lui même. Le problème d'énumération est ensuite résolu avec un algorithme par séparation et évaluation ou par un ILP.

1.3.3 Sélection des instructions spécialisées

La sélection des ISE consiste tout d'abord à identifier dans un graphe toutes les occurrences de motifs provenant d'une bibliothèque (isomorphisme de sous-graphe) et ensuite à choisir celles qui seront effectivement remplacées par des ISE. Il existe de nombreuses techniques de sélection qui se différencient notamment par leurs fonctions respectives d'optimisation et leurs exhaustivités.

1.3.3.1 Sélections optimales à une fonction de coût ou de mérite

Programmation dynamique Les algorithmes [123, 44] expriment le problème sous forme de couverture binaire (NP-difficile) qui peut être résolu efficacement par un algorithme par séparation et évaluation [46]. Pour l'algorithme [123] l'objectif est de maximiser la couverture alors que pour [44] c'est la durée d'exécution qui est minimisée (i.e., somme des durées des ISE sélectionnées). De plus l'approche [44] autorise le recouvrement des occurrences sélectionnées : un nœud peut être couvert par plusieurs ISE et l'opération du nœud est alors dupliquée. Ce comportement requiert une surface plus importante mais peut néanmoins favoriser la sélection d'instructions spécialisées plus intéressantes [4].

Les travaux [41, 197] sélectionnent les occurrences dont l'accélération matérielle est maximale sous contraintes de surface. Dans [41] la résolution utilise un algorithme par séparation et évaluation alors que dans [197] elle correspond à celle du problème dit de « sac à dos ».

L'algorithme [124] a l'originalité de chercher une solution au problème de sélection d'occurrences pouvant s'exécuter en parallèle sur une extension VLIW. En effet, dans ce cas, les algorithmes qui minimisent la somme des durées des ISE sélectionnées ou qui minimisent le nombre d'occurrences sélectionnées ne sont plus adaptés. Il est alors plus pertinent de minimiser le chemin critique du graphe couvert car plusieurs groupements de nœuds pourront s'exécuter en parallèle sur l'extension matérielle. C'est ce qu'optimise [124] par un algorithme de séparation et évaluation. Cependant, cet algorithme ne traite pas les difficultés issues du partage du processeur et du transfert des données vers/de l'extension VLIW.

Modélisation ILP Dans [40, 66] un ILP modélise le problème de sélection d'occurrences et minimise la surface totale nécessaire à la mise en œuvre matérielle des motifs des occurrences sélectionnées. L'approche [206] se différencie des précédentes du fait qu'elle s'intéresse à des applications temps réel et minimise le WCET [20] du graphe. L'algorithme [114] s'appuie également sur l'ILP pour maximiser les gains obtenus par les ISE en comparaison avec leurs durées d'exécution logicielle (i.e., sur le processeur hôte).

Modélisation CP L'algorithme [196] utilise la programmation par contraintes (cf. chapitre 2) pour modéliser le problème de sélection et d'ordonnancement (pour un nombre limité de ressources d'exécution) d'ISE. Deux stratégies de résolutions sont proposées : minimisation de la durée d'exécution sous contraintes de ressource et minimisation du nombre de ressources sous contraintes temporelles. Cependant, la modélisation temporelle des occurrences s'appuie sur des artefacts (appelés *dummy nodes*) qui compliquent considérablement la résolution du problème. De plus, les transferts de données entre le processeur et l'extension ne sont pas pris en compte, ils risquent donc de dégrader nettement la qualité des solutions identifiées.

L'approche [127] tient compte des transferts de données lors de l'évaluation de la durée d'une ISE, elle n'est cependant pas adaptée à un ordonnancement parallèle car son modèle temporel est conçu pour minimiser la somme des durées d'exécution des ISE sélectionnées.

20. Worst Case Execution Time

1.3.3.2 Heuristiques de sélection

Afin de favoriser la rapidité de l'étape complexe de sélection des ISE, il est parfois préférable d'utiliser des heuristiques qui sont spécifiques au problème et qui permettront de trouver une solution acceptable en un minimum de temps.

Les travaux [29] proposent trois algorithmes pour la sélection d'ISE. Le premier est un algorithme glouton qui détermine à chaque itération le motif qui maximise une fonction de mérite (e.g., accélération) dans le graphe restant à couvrir en tenant compte du nombre d'occurrences de chaque motif sélectionné. Le deuxième est également itératif mais est néanmoins prouvé comme étant optimal dans son contexte. Le dernier est une heuristique analysant un sous-ensemble des solutions explorées dans l'algorithme optimal mais qui donne des résultats qui en sont proches.

Il existe également des algorithmes qui se basent sur un graphe de conflit : les nœuds correspondent aux ISE candidates et les liens référencent les incompatibilités de sélection entre ISE. En effet, si un nœud est présent dans plusieurs ISE, une seule pourra être sélectionnée puisque le recouvrement n'est pas autorisé. Les algorithmes gloutons [81, 103] sélectionnent, à chaque itération, le plus grand sous-ensemble d'ISE qui ne sont pas incompatibles. La technique de sélection [102] est présentée par les mêmes auteurs que [103] et s'appuie sur un algorithme glouton qui maximise le nombre de nœuds couverts d'un graphe $G(V, E)$ et dont la complexité de chaque itération est $O(|V|^{2.5})$.

1.4 Résumé

La compilation d'une application pour un processeur extensible est particulièrement complexe puisque le jeu d'instructions du processeur n'est pas connu à l'avance. Ainsi, l'exploration architecturale constitue une dimension supplémentaire aux multiples problématiques d'optimisation du compilateur. Cette complexité s'est traduite en pratique par l'émergence de flots de conception qui aident l'expert à choisir le compromis le plus adapté à un type de produit. Le compilateur est au centre de ce processus exploratoire car il automatise la majorité des étapes qui permettront de valider la pertinence des choix effectués en amont.

L'extension de jeu d'instructions est un thème actif de recherche et de nombreux algorithmes ont été proposés pour répondre notamment aux défis de l'identification et de la sélection des opérations qui seront accélérées par une extension matérielle. Au cours des dix dernières années, c'est le thème de la génération d'ISE qui a été le plus étudié et l'on dispose actuellement d'algorithmes efficaces permettant de traiter plusieurs centaines de nœuds en un temps raisonnable. La problématique de la sélection reste un sujet beaucoup plus ouvert et les algorithmes existants effectuent généralement cette sélection en se basant sur des prédictions inadaptées à un ordonnancement parallèle.

Cette thèse apporte deux contributions pour l'exploitation du parallélisme dans un processeur extensible. La première repose sur la modélisation de la sélection et de l'ordonnancement d'ISE dans un unique problème d'optimisation dont l'objectif est de minimiser la durée d'exécution totale sous contraintes de ressources éventuellement parallèles. La seconde contribution à ce sujet permet de transformer automatiquement les nids de boucles d'un programme afin de faire apparaître des ISE qui seront vectorisables.

Ces contributions sont toutes deux basées sur la programmation par contraintes qui fait l'objet du prochain chapitre.

Chapitre 2

Optimisation par programmation par contraintes

La programmation par contraintes (CP [1]) est le formalisme que nous utiliserons pour modéliser et résoudre les différents problèmes d'optimisation NP-complets présentés dans les prochains chapitres de cette thèse. L'objectif de ce chapitre est de présenter succinctement la modélisation et la résolution d'un problème de satisfaction de contraintes. Ces bases sont en effet nécessaires à la compréhension de nos principales contributions pour l'extension de jeu d'instructions.

Sommaire

1. Constraint Programming

2.1 Introduction

La programmation par contraintes est un formalisme pour la modélisation et la résolution de problèmes difficiles (la plupart du temps NP-complets). Elle s'appuie sur une séparation nette entre le problème à résoudre et la manière de le résoudre.

Un problème est décrit par des variables (aux domaines finis) ainsi que par des liens logiques (appelées **contraintes**) entre ces variables. Ces contraintes énoncent les conditions dans lesquelles des valeurs choisies pour chacune des variables constituent une réponse valide au problème.

À la différence d'un problème d'optimisation linéaire en nombre entier (ILP [2]), la programmation par contraintes privilégie la lisibilité d'un problème modélisé en utilisant de multiples contraintes (dites globales) qui sont proches d'une réflexion naturelle. Par exemple, plutôt que de modéliser un problème d'ordonnancement sous contraintes de ressources en discrétisant le temps par des variables intermédiaires, on énoncera une unique contrainte globale qui se charge d'assurer le partage des ressources pour un ensemble de tâches.

Ce principe constitue l'essence même de la programmation par contraintes : l'algorithme de résolution est générique à n'importe quel problème et ce sont des **contraintes spécifiques** qui tireront parti des caractéristiques du problème pour améliorer l'efficacité de la recherche.

La première section de ce chapitre présente le formalisme de la programmation par contraintes. La section suivante introduit le mécanisme qui guide la recherche d'une solution par filtrage des valeurs qui ne satisfont pas les contraintes du problème. La troisième section présente quelques contraintes qui seront notamment utilisées dans les prochains chapitres de la thèse. Enfin, nous nous intéresserons à l'algorithme générique de résolution en présentant les différentes paramètres et techniques qui améliorent l'efficacité de la recherche.

2.2 Problème de satisfaction de contraintes

Un problème de satisfaction de contraintes, ou CSP [3], porte sur un ensemble de variables dont les valeurs possibles sont définies dans leurs domaines respectifs. Ces variables sont impliquées dans des contraintes qui peuvent limiter les combinaisons possibles de leurs valeurs. Dans le cadre de cette thèse, nous nous limiterons au formalisme de Montanari [133] où chaque domaine est fini et constitué de valeurs entières.

Résoudre un CSP consiste alors à choisir, pour chaque variable, une valeur issue de son domaine de définition et qui satisfasse l'ensemble des contraintes du problème (i.e., instanciation complète et consistante, cf. définitions 5 et 6). Si au moins une des contraintes n'est pas satisfaite, le problème est dit dans un état *inconsistant*.

2. Integer Linear Programming
3. Constraint Satisfaction Problem

Définition 1 (Domaine) *Un domaine est constitué d'un ensemble de valeurs entières. On distingue deux types de domaines :*

- *continu : les valeurs d'un domaine continu d_i sont comprises entre une borne basse (lb) et une borne haute (ub). On note $d_i :: [lb..ub]$*
- *disjoint : les valeurs d'un domaine disjoint d_i constituent une liste de k entiers. On note $d_i :: [v_1, v_2, \ldots, v_k]$*

Définition 2 (Variable) *Une variable x_i d'un CSP est associée à un ensemble de k domaines $D_{x_i} = d_1 \cup d_2 \cup \ldots \cup d_k$ qui indiquent les valeurs possibles de cette variable.*

Définition 3 (Contrainte) *Une contrainte c_i porte sur un ensemble de k variables $X_{c_i} = \{x_1, \ldots, x_k\}$ et restreint les valeurs possibles des combinaisons de valeurs pouvant être prises par X_{c_i} au sous-ensemble $R_{c_i} \subseteq \prod_{x_i \in X_{c_i}} D_{x_i}$. Une contrainte n'est satisfaite que si les valeurs de X_{c_i} se trouvent dans un des tuples de R_{c_i}. Si R_{c_i} est vide, alors la contrainte ne pourra pas être satisfaite.*

Définition 4 (CSP) *Une instance de CSP est un triplet $P = (X, D, C)$ où $X = \{x_1, x_2, ..., x_n\}$ désigne un ensemble de variables, D l'ensemble de leurs domaines respectifs et $C = \{c_1, c_2, ..., c_m\}$ les contraintes qui limiteront les valeurs légales de X.*

Définition 5 (Instanciation) *Une instanciation de $P = (X, D, C)$ est un tuple I de valeurs prises dans le produit cartésien $\prod_{y_i \in Y} D_{y_i}$ des domaines d'une liste de variables $Y \subseteq X$. Si $Y = X$, l'instanciation est dite complète et dans le cas le contraire, elle est partielle.*

Définition 6 (Instanciation consistante) *Une instanciation de $P = (X, D, C)$ est consistante si chaque contrainte $c_i \in C$ est satisfaite. Dans le cas contraire, l'instanciation est inconsistante.*

2.3 Propagation des contraintes

La recherche d'une solution d'un CSP est basée sur une exploration de l'espace des valeurs possibles de chaque variable. Cependant, une des caractéristiques de la programmation par contraintes est de réduire dynamiquement cet espace de recherche en filtrant, en fonction de l'état courant des variables du problème, les valeurs qui ne permettent pas de satisfaire une ou plusieurs contraintes. Cette technique de filtrage permet donc d'élaguer l'espace de recherche en coupant les branches qui ne mèneront pas à une solution valide.

La figure 2.1 illustre l'effet du filtrage sur l'espace de recherche de toutes les solutions au problème $P = (\{x, y\}, \{[1..3], [2..4]\}, x = y)$. Une exploration exhaustive de l'espace de recherche (figure 2.1.c) est faite en douze évaluations dont seulement deux constituent des solutions légales (i.e., qui satisfont $x = y$). La consistance de la contrainte $x = y$ permet, avant même d'explorer l'espace de recherche, de supprimer les valeurs incohérentes des domaines de x et y (figure 2.1.b). Puis, après évaluation de la variable x, la dernière valeur incohérente de y est également filtrée et quatre évaluations (figure 2.1.d) suffisent à construire l'ensemble des solutions valides du problème.

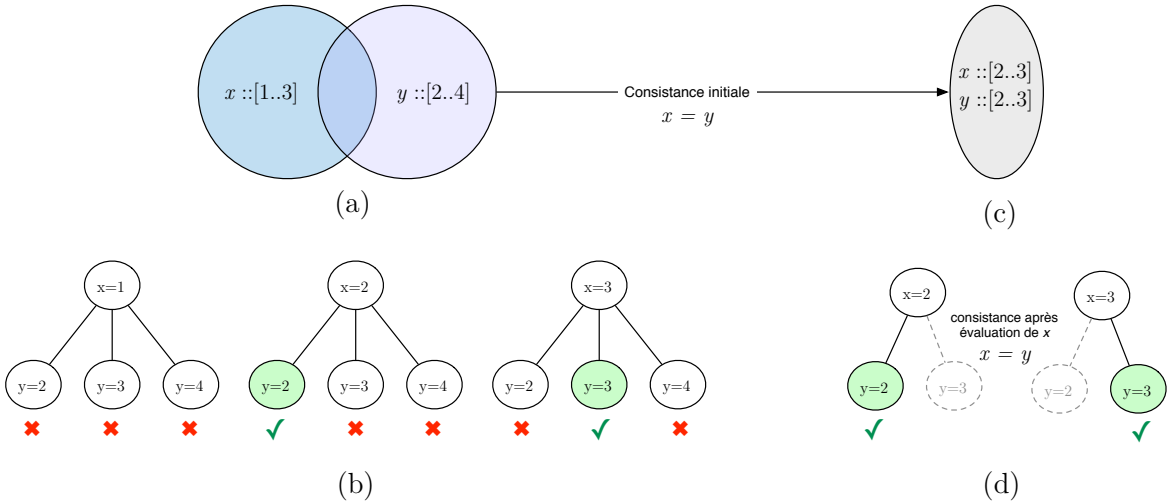


FIGURE 2.1 – Impact du filtrage sur l'espace de recherche pour la contrainte $x = y$. (a) Domaines initiaux des variables x et y. (b) Espace de recherche sans filtrage des valeurs. (c) Domaine commun à x et y après que leurs domaines respectifs soient filtrés pour ne contenir que leurs valeurs communes (consistance de la contrainte $x = y$). (d) Espace de recherche avec filtrage.

Le filtrage des valeurs est appliqué à chaque contrainte du problème. Sa mise en œuvre est faite en deux étapes successives qui seront effectuées après chaque évaluation de variable et tant que les domaines des variables sont modifiés :

1. suppression, dans les domaines, des valeurs qui provoquent une inconsistance de la contrainte.
2. propagation des changements aux autres contraintes du problème.

L'efficacité de la résolution d'un problème est donc significativement influencée par la capacité de chaque contrainte à détecter, le plus tôt possible, les valeurs incohérentes.

2.3.1 Consistances locales

Un problème CSP peut être représenté par un hypergraphe où chaque contrainte est un hyperlien reliant toutes les variables qui y sont impliquées. La figure 2.2.a illustre l'hypergraphe d'un problème contenant trois variables x, y et z et deux contraintes ($x > y$ et $x + y = z$).

D'autre part, si toutes les contraintes d'un CSP sont *binaires* (i.e., qui impliquent deux variables) ce graphe est appelé *réseau de contraintes* et ne contient aucun hyperlien car chaque contrainte n'utilise alors que deux variables au maximum.

Il est toujours possible de construire une forme binaire d'un CSP quelconque. La figure 2.2 illustre un exemple de *binarisation* d'un problème qui contient une contrainte ternaire ($x + y = z$). Un ensemble de tuples U « encapsule » la contrainte $x + y = z$ correspondant au produit cartésien des domaines des variables x, y et z qui satisfont la contrainte. Ces tuples sont utilisés pour exprimer des contraintes binaires liant chaque variable à U : e.g., z étant la dernière variable d'un tuple, on ajoute une contrainte binaire $z = U[2]$. Dans l'exemple, il n'y a que trois tuples possibles dans U $\{(1, 4, 5), (2, 3, 5), (2, 4, 6)\}$ le sous-ensemble de valeurs pour la contrainte $z = U[2]$ est donc $R_{(z=U[2])} = \{((1, 4, 5), 5), ((2, 3, 5), 5), ((2, 4, 6), 6)\}$.

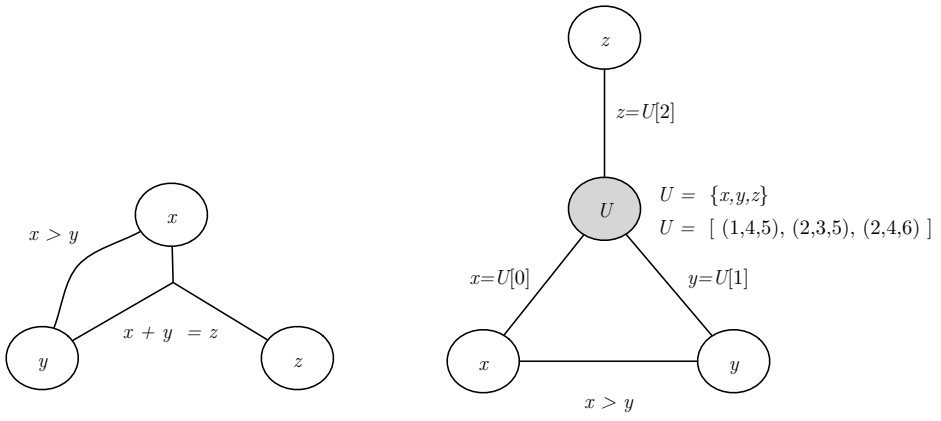


FIGURE 2.2 – Binarisation de $P = (\{x, y, z\}, \{[1..2], [3..4], [5..6]\}, \{x+y = z, x > y\})$. (a) Hypergraphe des contraintes (b) Graphe des contraintes binaires obtenues après binarisation.

La représentation binaire d'un CSP est utilisée dans des techniques de filtrage, appelées *consistances d'arcs*, qui sont applicables à n'importe quel réseau de contraintes. Pour une description détaillée des algorithmes de consistances d'arcs, le lecteur est invité à consulter le livre [164] à partir de la page 37.

Si ces techniques de filtrage sont particulièrement efficaces pour un faible nombre de variables, leur généralisation à des chemins de taille K (on parle alors de *K-consistance*) est souvent trop coûteuse (i.e., la complexité est exponentielle) pour réduire efficacement les domaines d'un CSP complexe. On préfère alors utiliser des contraintes *n-aires* dont la sémantique spécifique est associée à des algorithmes de filtrage dédiés et beaucoup plus efficaces.

2.3.2 Consistances de contraintes globales

Les contraintes globales impliquent un ensemble de variables dont la taille est généralement un paramètre de la contrainte. À chaque contrainte globale correspond une méthode de propagation spécifique qui tire parti de la sémantique de la contrainte pour filtrer efficacement les valeurs incohérentes des variables.

Les algorithmes utilisés (on parle de consistance globale) sont souvent issus de la recherche opérationnelle et s'avèrent plus efficaces et plus rapides que des consistances d'arcs sur un problème binarisé. Les contraintes globales permettent donc de capitaliser les méthodes de résolution de problèmes difficiles (e.g., problème du sac à dos, ordonnancement de tâches sous contraintes de ressources, etc.) pour être réutilisables dans de multiples contextes.

Une autre qualité des contraintes globales est de simplifier considérablement la modélisation de problèmes complexes : elles permettent d'abstraire des sous-problèmes présents dans de nombreuses applications réelles.

2.4 Quelques contraintes

Dans cette section, nous présentons succinctement les contraintes utilisées dans les prochains chapitres, pour modéliser les problèmes de sélection d'instructions spécialisées. Un catalogue de contraintes en ligne [4] référence plus exhaustivement les nombreuses contraintes existantes.

2.4.1 Contraintes arithmétiques, logiques et conditionnelles

Une contrainte arithmétique exprime une comparaison $(=, \neq, >, <, \geq, \leq)$ entre deux expressions arithmétiques quelconques (e.g., linéaires, polynomiales, etc.). Une contrainte arithmétique est généralement décomposée en contraintes binaires sur lesquelles sont appliquées des consistances d'arcs. Par exemple : $x_1 + x_2 * x_3 > x_4$.

Les contraintes logiques expriment des liens logiques (i.e., conjonction, disjonction, équivalence, etc.) entre différentes contraintes arithmétiques. Par exemple : $x_1 > 0 \wedge x_2 < 0$.

Les contraintes conditionnelles expriment des causalités entre contraintes arithmétiques. Par exemple : $if(x_1 > 0)\ then\ (x_2 = x_3 + x_4)\ else\ (x_2 > x_5)$.

2.4.2 AllDifferent (contrainte globale)

La contrainte *AllDifferent* impose, comme son nom l'indique, que les valeurs d'un ensemble de variables soient toutes différentes.

$$AllDifferent(ListV)$$

où $ListV$ est un ensemble de variables.

2.4.3 Element (contrainte globale)

La contrainte *Element* symbolise l'affectation de la valeur d'une variable res à celle d'une autre variable positionnée dans une liste par une variable d'indice appelée I.

$$Element(I, ListV, res)$$

où $ListV$ est une liste de variables.

De manière plus intuitive, on peut également noter cette contrainte selon la syntaxe d'un accès tableau :

$$res = ListV[I]$$

où $ListV$ est une liste de variables.

Par exemple, pour une variable $I :: [0..2]$ et une liste de variables $L = \{x_1, x_2, x_3\}$ telles que $x_1 :: [1..2], x_2 :: [3..4], x_3 :: [5..6]$, la contrainte $Element(I, L, res)$ qui lie une variable $res :: [3..6]$ implique, après consistance, que le domaine de I est réduit à $[1..2]$. De même, si $I = 1$ alors le domaine de res est réduit à $[3..4]$

4. http://www.emn.fr/z-info/sdemasse/gccat/

2.4.4 Diff2 (contrainte globale)

La contrainte *Diff2* impose le non-recouvrement de rectangles bidimensionnels. Chaque rectangle $Rect_k$ est défini par les variables x_k, y_k, w_k et h_k dont les valeurs correspondent respectivement aux coordonnées de l'origine, à la largeur et à la hauteur de $Rect_k$. La figure 2.3.a montre le cas où une contrainte *Diff2* n'est pas satisfaite puisque deux rectangles (*Rect*2 et *Rect*6) se chevauchent. Pour que la contrainte soit respectée, il suffit de décaler l'abscisse du rectangle *Rect*6.

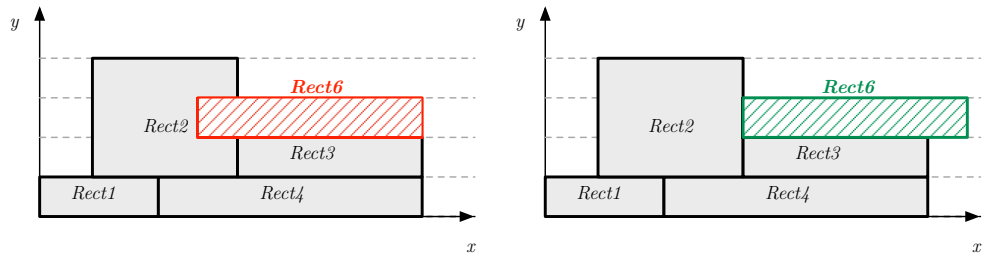


FIGURE 2.3 – Contrainte différentielle à deux dimensions. (a) Le rectangle *rect*6 recouvre *rect*2, la contrainte n'est pas satisfaite. (b) Les rectangles *rect*6 et *rect*2 ne se chevauchent plus, la contrainte est satisfaite.

On note la contrainte *Diff2* selon la syntaxe suivante :

$$Diff2(ListRect)$$

où $ListRect$ est une liste de rectangles.

La contrainte *Diff2* est particulièrement adaptée au problème d'ordonnancement sous contraintes de ressources. L'axe des abscisses indique alors le temps et celui des ordonnées correspond aux ressources utilisées. Une tâche est représentée sous forme de rectangle à partir de sa date de début, de sa durée et de la ressource occupée. Imposer une telle contrainte garantit qu'à n'importe quel instant, des tâches distinctes ne seront pas affectées à la même ressource.

Modélisation	Variables	Contraintes	Durée pour trouver la solution optimale	Durée pour prouver l'optimalité	Mauvaises décisions
(C1) + (C2)	76	1110	1,9s	14,9s	19323
(C1) + (C3)	76	53	1,3s	1,3s	0

TABLE 2.1 – Comparaison des modélisations pour un problème d'ordonnancement sous contraintes de ressources [111].

L'expérience réalisée dans [111] compare deux approches de modélisation d'un problème d'ordonnancement simple. Un nombre limité de ressources doit exécuter les opérations d'un programme en un minimum de temps. Deux types de contraintes apparaissent : le respect des dépendances des données entre les opérations et le partage des ressources disponibles. La première approche consiste à modéliser le problème selon un programme linéaire classique : un ensemble de contraintes (C1) modélise les contraintes portant sur la dépendance des données et un autre ensemble (C2) celles portant sur le partage des ressources. La deuxième approche utilise, quant à elle, une contrainte *Diff2* (C3) pour

modéliser le partage des ressources. Le tableau 2.1 résume les résultats obtenus : le problème est plus simple à modéliser et l'algorithme de propagation dédié permet de ne prendre aucune mauvaise décision lors de la recherche de la solution optimale.

2.4.5 Cardinalité (contrainte globale)

Les contraintes de cardinalité analysent les différentes valeurs possibles dans les domaines d'un ensemble de variables. Les différents problèmes modélisés dans cette thèse utilisent deux types de contraintes de cardinalité.

La contrainte *Count* compte le nombre de fois où un entier c est une valeur possible dans un ensemble de variables.

$$Count(ListV, var, c)$$

où $ListV$ est un ensemble de variables, var une variable et c une constante.

Exemple : la contrainte $Count(\{x_1 :: [1..10], x_2 :: [5]\}, var :: [0..10], 5)$ implique que le domaine de var_1 soit réduit à $[1..2]$.

La contrainte $Values$ compte le nombre de valeurs différentes dans un ensemble de variables.

$$Values(ListV, var)$$

où $ListV$ est un ensemble de variables et var est une variable.

Exemple : la contrainte $Values(\{x_1 :: [1..2], x_2 :: [1..3]\}, var :: [2])$ implique que le domaine de x_2 soit réduit à $[1..2]$.

2.5 Recherche de solution(s)

2.5.1 Algorithme de recherche en profondeur

La recherche d'une ou de toutes les solutions d'un CSP repose sur une exploration des combinaisons de valeurs possibles des variables qui satisfont toutes les contraintes du problème. Il s'agit d'un algorithme de *parcours en profondeur* (DFS [5]) dans un espace de recherche arborescent où chaque nœud correspond à l'évaluation d'une valeur possible pour une variable.

Le nombre de variables à évaluer pour un problème détermine la profondeur de l'arbre à explorer. Il est parfois possible de réduire la profondeur de l'arbre en ne considérant que les *variables de décision* d'un problème. En effet, les contraintes associées à certaines variables sont parfois si fortes qu'elles ne laissent aucune ambiguïté quant à ses valeurs possibles. Par exemple, si b_1 et b_2 sont deux variables booléennes liées par la contrainte $b_1 + b_2 = 1$, il est évident que l'évaluation de b_1 ou de b_2 suffira à déterminer la valeur de l'autre et une seule de ces variables sera considérée comme une variable de décision.

Chaque choix explicite d'une valeur est notifié aux contraintes du problème. Celles-ci utiliseront leurs techniques de propagation respectives pour réduire les domaines de leurs variables et, le cas

5. Depth First Search

échéant, constater que la valeur choisie ne permet pas de satisfaire toutes les contraintes (i.e., l'instanciation du problème est inconsistante).

Si l'évaluation d'une variable conduit à un état inconsistant, il est inutile de parcourir le reste de la branche. Pour continuer la recherche, il est alors nécessaire de revenir en arrière dans l'arbre et de choisir une autre valeur pour la variable, on parle de *backtracking*. À la fin de l'algorithme, une solution au problème correspond à un chemin valide entre les nœuds de la première et de la dernière variable évaluée. Si aucun chemin n'existe, le problème n'a pas de solution.

2.5.2 Optimisation d'une fonction de coût

Un CSP s'intéresse à la satisfiabilité d'un problème et à la recherche d'une ou de toutes les solutions qui y répondent. Néanmoins, la programmation par contraintes peut facilement être étendue aux problèmes d'optimisation, on parle alors de COP [6].

Le principe est de définir une variable contrainte par une fonction de coût. Une fois qu'une solution satisfaisant toutes les contraintes du problème est identifiée dans l'espace de recherche, la variable de coût est alors associée à une valeur. Cette valeur correspond au coût de la solution courante et chercher à optimiser (i.e., minimisation ou maximisation) le problème consiste alors à ajouter une contrainte qui borne le coût, puis de poursuivre l'exploration de l'espace de recherche. Le comportement de l'algorithme est donc similaire à un algorithme standard par séparation et évaluation (*Branch&Bound*) : si la contrainte qui borne le coût ne peut être satisfaite par une branche, celle-ci ne sera pas parcourue.

À la fin du parcours de l'espace de recherche, la dernière solution identifiée est celle qui minimise (ou maximise) la fonction de coût et, si l'exploration est complète, cette solution est garantie comme étant optimale.

Il est important de noter que la solution optimale peut parfois être trouvée très rapidement mais que la preuve de cette optimalité requiert d'explorer exhaustivement l'espace de recherche. Les techniques de filtrage des contraintes du problème (y compris celles de la fonction de coût) réduisent la taille de l'arbre à parcourir mais elles ne suffisent pas toujours à le réduire suffisamment pour prouver l'optimalité en un temps raisonnable. Dans ce cas, on peut fixer une limite temporelle sur le temps de résolution ou encore un seuil quant au nombre total de décisions prises. Une fois cette limite atteinte, le résultat est la dernière solution identifiée qui n'est donc pas prouvée comme étant optimale.

2.5.3 Ordre d'évaluation des variables et des valeurs

Les seuls paramètres du parcours de l'arbre de recherche sont l'ordre d'évaluation des variables et le choix des valeurs évaluées dans leurs domaines respectifs. Ces deux paramètres ont cependant un impact décisif sur l'efficacité de la recherche : il est évident que choisir les valeurs maximales de chaque variable lors de l'évaluation risque de compliquer considérablement la recherche d'une solution minimisant un ordonnancement temporel.

Le choix de l'ordre d'évaluation des variables est un problème complexe dont le comportement est parfois difficilement prédictible. Le choix d'une variable avant une autre peut en effet faire apparaître plus rapidement des branches inconsistantes ou encore réduire plus efficacement les domaines des

6. Constraint Optimization Problem

variables. De nombreuses stratégies ont été proposées pour guider l'ordre d'évaluation et se divisent en deux catégories principales.

- Ordres statiques. Ces stratégies utilisent l'instance initiale du problème pour ordonner les variables (e.g., variables qui sont associées au plus de contraintes, etc.).
- Ordres dynamiques. Ces stratégies s'adaptent à l'état courant du problème lors de la résolution (e.g., variables dont les domaines courants sont les plus larges, les plus petits, etc.).

La sélection d'une valeur pour une variable est généralement effectuée selon quelques stratégies : valeur minimale, maximale ou encore médiane du domaine. Cependant, il est parfois préférable d'utiliser des stratégies personnalisées qui seront en adéquation avec le problème à résoudre. Celles-ci s'appuieront sur une connaissance dynamique de l'état des variables pour sélectionner, par exemple, la valeur qui profitera le plus à une fonction de coût.

L'ordre d'évaluation de variables et le choix des valeurs constituent des paramètres aisément modifiables qui guident le parcours de l'arbre de recherche. Ce parcours peut également bénéficier d'améliorations qui pallient aux problèmes de l'algorithme de *backtracking*.

2.5.4 Améliorations de l'algorithme

Lorsqu'une instanciation est inconsistante, l'algorithme DFS retourne en arrière pour évaluer de nouvelles branches. Cependant, l'algorithme original de *backtracking* opère un retour en arrière chronologique : il se contente de remonter au nœud père de l'échec pour essayer une autre valeur. Dans le cas où aucune des valeurs de ce nœud père ne permet d'obtenir une solution valide, on procède de la même manière avec l'ancêtre précédent. Ce comportement, qui a le mérite d'être générique, pose deux types de problèmes.

- *trashing* : cas où un choix antérieur ne permettra pas, quelle que soit la branche fille, de trouver une solution valide. L'algorithme chronologique remonte alors dans l'arbre mais seulement après avoir testé inutilement toutes les branches filles.
- *redondance* : reproduction des mêmes choix alors qu'ils conduisent à une instanciation inconsistante. Une combinaison de valeurs suffit parfois à montrer qu'aucune sélection n'existe sur la branche. Si cette combinaison est répétée plusieurs fois, l'algorithme chronologique les évaluera inutilement.

Pour pallier à ces problèmes, il existe des méthodes dites de *backtracking intelligents* qui analysent les raisons de l'échec pour éviter de reproduire les mêmes erreurs.

Les approches dites de *backjumping* [68, 152] consistent à retourner à l'ancêtre le plus ancien du nœud inconsistant mais sans pour autant oublier des solutions. Si le *backjumping* réduit le phénomène de *trashing*, ces approches ne permettent pas de traiter les redondances.

Les approches par *apprentissage* [172, 49, 69] réduisent les redondances en mémorisant les combinaisons de valeurs qui mènent à une inconsistance de l'instanciation partielle ou complète. La figure 2.4 illustre le phénomène de redondance dans un arbre de recherche, chaque nœud est une évaluation annotée par l'état de l'instanciation courante. Par exemple, l'annotation $(a, *, *, *)$ correspond au choix de la valeur a pour la première variable, les variables qui n'ont pas encore été évaluées sont notées $*$. Dans la figure, la branche de gauche qui part de $a1$ ne possède aucune solution valide. Si l'on considère que le conflit provient de la combinaison $(*, b, c, d)$, il est possible d'éviter le parcours des

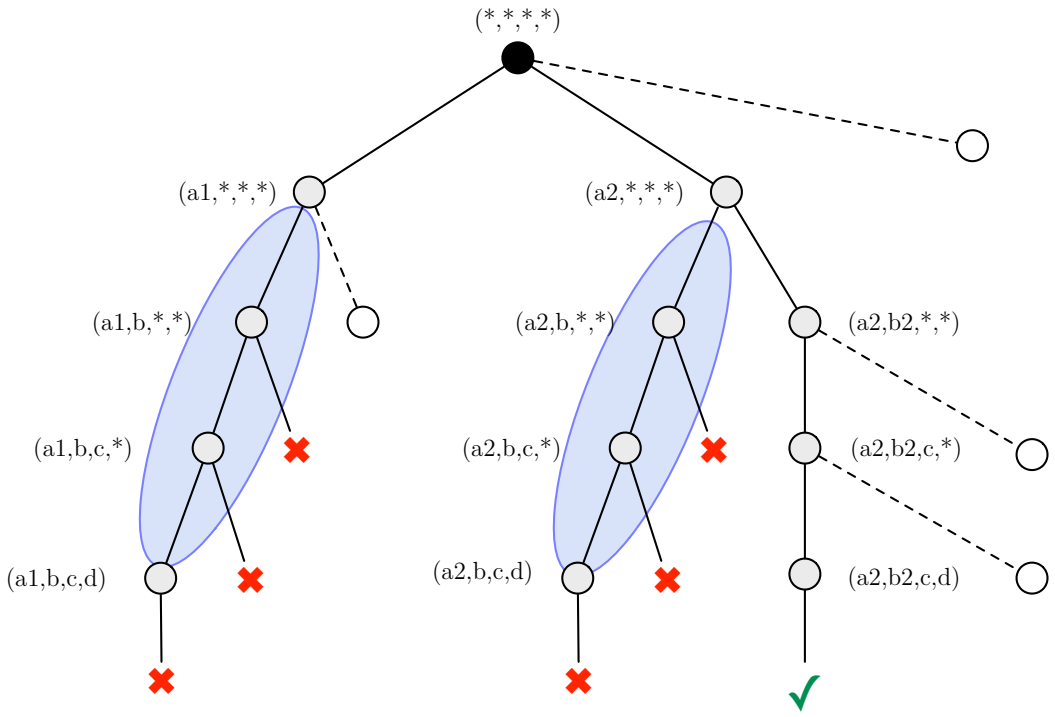


FIGURE 2.4 – Exemple de redondance dans un arbre de recherche. Le choix des valeurs (b, c, d) conduisent à deux instanciations inconsistantes pour les mêmes raisons.

mêmes choix pour la branche $(a2, *, *, *)$ en bénéficiant de l'expérience acquise lors du parcours de la branche $(a1, *, *, *)$. L'apprentissage est mis en œuvre par des contraintes appelées *nogood* qui sont ajoutées au problème et dont les consistances élagueront l'arbre en utilisant l'information acquise lors des parcours précédents. Ces contraintes peuvent néanmoins poser un problème en terme d'espace mémoire et du temps requis pour vérifier leurs satisfiabilités. Une utilisation efficace cherchera donc un compromis entre le gain apporté par l'élagage de branches de l'arbre et le surcoût associé à la gestion des *nogood*.

2.6 Conclusion

La programmation par contraintes peut être vue comme un puissant outil de capitalisation de connaissances sur la résolution de problèmes difficiles. Les techniques de filtrage bénéficient des résultats avancés de la recherche opérationnelle pour prévenir, au plus tôt, le choix de valeurs qui ne permettront pas de satisfaire les contraintes du problème.

D'autre part, on peut considérer la programmation par contraintes comme un support générique de mise en œuvre d'algorithmes *Branch&Bound*. Adapter l'algorithme au problème s'envisage alors selon ces trois axes principaux : 1) utilisation de contraintes spécifiques qui élagueront intelligemment l'espace de recherche 2) sélection d'un ordre d'évaluation des variables adapté au problème 3) sélection des valeurs en adéquation avec la fonction de coût.

Un autre point particulièrement intéressant de la programmation par contraintes réside dans la lisibilité des modèles de contraintes : les contraintes spécifiques capturent des sous-problèmes récurrents dans de nombreux problèmes d'optimisation. Cependant, cette lisibilité est souvent masquée par les interfaces entre l'utilisateur et les solveurs de contraintes qui s'appuient généralement sur un langage de programmation généraliste pour décrire et résoudre un CSP. Ce constat a motivé l'apparition de langages dédiés et le chapitre 8 propose un nouvel environnement de modélisation modulaire de CSP qui simplifie leur description et leur capitalisation.

Chapitre 3

Sélection et ordonnancement simultané d'instructions pour processeurs spécialisés

Dans le premier chapitre, nous avons notamment fait apparaître que les approches existantes pour l'extension de jeu d'instructions ne sont généralement pas adaptées à traiter le parallélisme potentiel de l'architecture. Dans ce chapitre nous proposons une nouvelle technique, basée sur la programmation par contraintes, qui sélectionne et ordonnance les ISE de manière à minimiser la durée d'exécution totale de l'application finale (i.e., utilisant les ISE).

Sommaire

3.1 Introduction

L'extension du jeu d'instructions d'un processeur existant est une solution qui simplifie significativement la mise en place d'une chaîne de compilation complète pour un processeur spécialisé. En effet, contrairement à la conception complète d'un ASIP, l'architecture (i.e., processeur couplé à une extension matérielle) dispose du jeu d'instructions standard du processeur extensible et permet donc d'exécuter n'importe quelle application en utilisant le compilateur natif du processeur. Le problème peut alors être résumé en les questions suivantes : *Quelles sont, parmi les opérations de l'application analysée, celles qui utiliseront le matériel dédié ? Comment et quand seront-elles exécutées ?*

Dans ce chapitre, nous proposons une méthodologie basée sur la programmation par contraintes pour exprimer, dans un unique problème d'optimisation (CSP [1], cf. chapitre 2), la sélection et l'ordonnancement des groupes d'opérations exécutés sur l'extension matérielle d'un processeur extensible. Ces groupes d'opérations sont alors assimilés à des **motifs** de calculs, ils sont associés à un matériel dédié qui réduira les durées d'exécution de leurs **occurrences** dans une application. Ces occurrences correspondent alors à des instructions spécialisées qui étendent le jeu d'instructions standard du processeur extensible.

L'intérêt principal de notre approche est de faciliter l'exploration de différents compromis (e.g, performance, surface, consommation énergétique, etc.) en proposant un cadre unifié dans lequel pourront être ajoutées des contraintes supplémentaires sans remettre en cause le reste du modèle. Ainsi, la figure 3.1 illustre les deux types d'architecture que nous considèreront. La première (figure 3.1.a) est constituée d'un unique bloc matériel dont la reconfiguration fonctionnelle permettra d'accélérer le traitement d'un groupement d'opérations sélectionné dans l'ensemble des motifs supportés. La seconde extension matérielle (figure 3.1.b) utilise plusieurs de ces mêmes blocs comme autant d'unités fonctionnelles parallèles qui permettront de mettre en œuvre des instructions spécialisées VLIW [2].

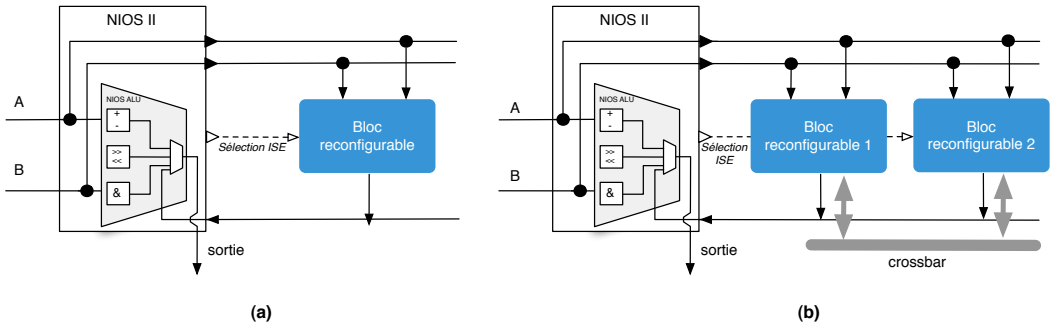


FIGURE 3.1 – Modèles d'architecture supportés pour un processeur extensible NIOSII. (a) extension composée d'un unique bloc reconfigurable fonctionnellement (b) extension composée d'un ensemble de blocs reconfigurables pouvant s'exécuter en parallèle.

Cette méthode s'intègre dans un flot de conception ASIP (cf. figure 3.2) qui assiste l'utilisateur dans l'identification et la sélection de nouvelles instructions spécialisées en analysant une application écrite dans un langage de haut niveau (i.e., langage C) afin de produire un code C compilable par la chaîne de compilation du processeur extensible, ainsi qu'une description matérielle synthétisable de l'extension matérielle.

1. Constraint Satisfaction Problem
2. Very Long Instruction Word

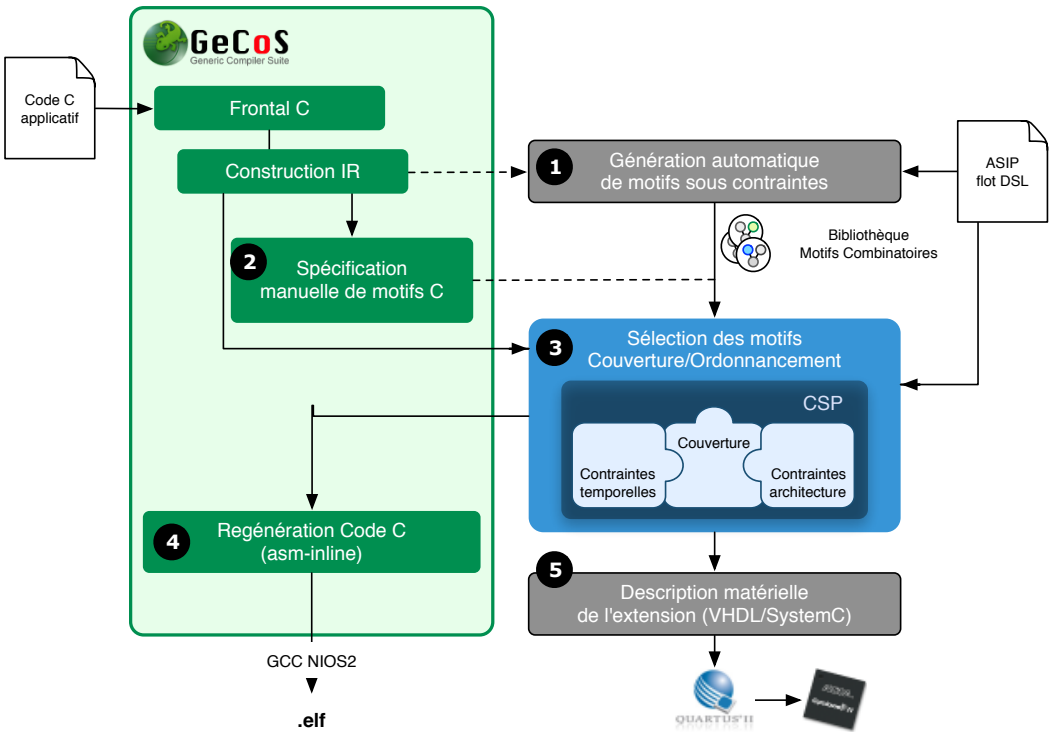


FIGURE 3.2 – Flot de conception ASIP

Les principales contributions de ce chapitre sont les suivantes :

- Une technique de couverture de graphe extensible, basée sur la programmation par contraintes, pour minimiser la durée d'exécution parallèle, sans contraintes de ressources, de groupes de nœuds sélectionnés et ordonnancés.
- La prise en compte des contraintes architecturales fines (au niveau cycle) d'une extension séquentielle pour sélectionner et ordonnancer conjointement les groupes d'opérations à exécuter sur l'extension.
- La prise en compte des contraintes architecturales fines (au niveau cycle) d'une extension parallèle pour sélectionner et ordonnancer des instructions spécialisées de type VLIW dans une application.

Le chapitre est organisé comme suit. La première section présente les différentes étapes du flot de conception ASIP (conçu en collaboration avec Kevin Martin [125]) en faisant notamment apparaître les multiples contributions logicielles apportées à l'infrastructure de compilation source à source GeCoS [3] (utilisée dans la mise en œuvre du flot). La deuxième section détaille une nouvelle technique de couverture de graphe qui a été étendue (cf. sections trois et quatre) pour sélectionner et ordonnancer des instructions spécialisées dans le cas, respectivement, d'une extension séquentielle et d'une extension comportant différents blocs pouvant s'exécuter en parallèle.

3. http://gecos.gforge.inria.fr

3.2 Présentation du flot de conception ASIP

Le flot de conception proposé (cf. figure 3.2) est composé de cinq tâches principales qui sont numérotées de 1 à 5 dans la figure. Les deux premières consistent à identifier, par génération ou par spécification explicite de la part du concepteur, les différents groupes d'opérations qui seront succeptibles d'être accélérés par l'extension matérielle. La troisième tâche sélectionne ceux qui seront effectivement utilisés et qui minimiseront l'ordonnancement de l'application. Les deux dernières tâches produisent les fichiers qui permettront à la chaîne de compilation du processeur extensible de tirer parti des instructions spécialisées sélectionnées. Dans cette section, nous présentons chacune de ces tâches en nous attardant tout d'abord sur l'infrastructure de compilation source à source GeCoS utilisée en *front-end* et en *back-end* du flot.

3.2.1 Infrastructure de compilation GeCoS

3.2.1.1 Présentation de l'infrastructure

GeCoS est une infrastructure de compilation, pour le langage C, dont une des particularités est d'être intégrée à l'environnement Eclipse [4]. Cette intégration permet de profiter de l'importante extensibilité de l'infrastructure et offre notamment la possibilité de définir et d'utiliser des éditeurs textuels dédiés à des langages spécifiques (DSL [5]) afin de guider les différentes étapes d'optimisation.

Dans sa version actuelle, GeCoS est notamment utilisé comme une infrastructure de compilation reciblable pour la synthèse d'ASIP et/ou l'extension de processeurs programmables [121, 142, 127]. GeCoS sert également de support à des transformations source à source (C vers C) basées sur le modèle polyédrique [134] ou encore comme un outil de vérification statique de programmes OpenMP [19] intégré à l'éditeur C d'Eclipse.

Dans notre flot de conception, nous utilisons principalement deux éléments de GeCoS.

1. Le *front-end* C, à partir duquel nous construisons une représentation intermédiaire dans laquelle les opérations des blocs de base et leurs dépendances (contrôle et données) sont représentées sous la forme d'un graphe acyclique flot de données. Cette représentation est ensuite utilisée pour extraire des motifs spécifiés par l'utilisateur (à l'aide de fonctions annotées par la directive `GCS_PATTERN` qui représentent les motifs à sélectionner), et qui seront utilisés pendant le processus de couverture (tâche 3 du flot).
2. Le *back-end* source à source utilisé lors de la phase de régénération d'un code C exploitant les extensions du jeu d'instructions et qui est destiné à être recompilé par la chaîne de *cross-compilation* du processeur cible (NIOS2-gcc). Dans ce code C, l'appel aux extensions définies par l'utilisateur se fait par l'utilisation d'instructions assembleur *en-ligne*.

3.2.1.2 Script de compilation

Un flot de compilation dans GeCoS est contrôlé par un *script* qui énonce une séquence de passes de transformation et d'optimisation de code. Chacune de ces passes constitue un greffon de GeCoS. Le mécanisme de *points d'extension* d'Eclipse se charge ensuite de réaliser, à l'exécution, l'édition

4. http://www.eclipse.org
5. Domain Specific Language

```
1  ps = CDTFrontEnd("example.c");
2  for proc in ps do
3    SSAAnalyser ( proc ) ;
4    do
5      ConstantPropagator ( proc ) ;
6      ConstantEvaluator ( proc ) ;
7    while changing ;
8    RemovePhiNode( proc ) ;
9    AlgebraicSimplifier(proc);
10 done;
11 DAGBuilder(ps);
12 AsipFlow("nios2.asipflow",ps);
13 CRegenerator(ps);
```

FIGURE 3.3 – Flot de compilation ASIP décrit dans un script GeCoS.

de lien entre le cœur de l'infrastructure et les greffons qui lui sont associés. Le langage de script de GeCoS est non typé et les variables sont déclarées implicitement.

La figure 3.3 montre le script GeCoS correspondant à un flot de compilation ASIP pour le processeur extensible NIOSII. La passe `CDTFrontEnd` (ligne 1) utilise le *front-end* de l'environnement de développement C/C++ d'Eclipse (CDT [6]) pour construire la représentation intermédiaire de GeCoS dont la racine est un ensemble de procédures. Cette représentation est un CDFG [7] hiérarchique préservant la structure originale du code C, les blocs de base y contiennent des instructions dans une forme arborescente.

Chaque procédure est ensuite transformée en une forme à assignation unique (SSA [8]) qui permet d'appeler une passe de propagation de constante (ligne 5) et une passe d'évaluation de constante (ligne 6) tant que celles-ci provoquent un changement dans la représentation intermédiaire de la procédure courante.

Une fois ces optimisations effectuées, la forme SSA n'est plus nécessaire ; la représentation intermédiaire est transformée en une forme standard (ligne 8) et une passe de simplification algébrique (ligne 9) élimine les opérations neutres (e.g., multiplication par 1, etc.).

Les blocs de base du CDFG sont ensuite transformés (ligne 11) dans une représentation intermédiaire analysable par nos outils d'extension de jeux d'instructions. Dans cet exemple, on se base sur un graphe acyclique (DAG [9]) pour représenter chaque instruction primitive sous forme d'un nœud dont les prédécesseurs constituent les opérandes.

La passe suivante (ligne 12) correspond à l'appel du flot d'extension de jeu d'instructions (cf. figure 3.3) sur l'ensemble des procédures du fichier C. Les paramètres du flot sont décrits par un fichier externe dans un langage spécifique.

6. C Development Tooling : http://www.eclipse.org/cdt/
7. Control Data Flow Graph
8. Single Static Assignement
9. Directed Acyclic Graph

3.2.1.3 Contributions logicielles dans GeCoS

Au cours de cette thèse, l'infrastructure GeCoS nous a servi de support pour mettre en œuvre plusieurs approches d'extension de jeu d'instructions. Cette infrastructure a également beaucoup évolué en réponse à nos besoins spécifiques. En effet, un flot ASIP se distingue d'un compilateur pour processeur généraliste du fait qu'il explore à la fois l'espace des transformations de code et celui des architectures matérielles dédiées. Cette particularité favorise la diversité des représentations intermédiaires, des modèles d'optimisation ainsi que des interactions avec le concepteur de l'extension matérielle.

Dans ce contexte, nous avons contribué à l'infrastructure GeCoS principalement autour des axes suivants.

- Utilisation de l'ingénierie dirigée par les modèles (IDM). La multiplication des représentations intermédiaires et des transformations a fait apparaître un réel besoin d'outils transversaux aux domaines métiers d'un compilateur. C'est dans cette optique que cette thèse a contribué à l'intégration des méthodologies de l'IDM dans l'infrastructure GeCoS. Chacune des représentations intermédiaires manipulées dans GeCoS est dorénavant exprimée dans une structure commune appelée métamétamodèle, celle-ci permet notamment d'utiliser les multiples outils génériques issus de la communauté IDM. L'expérience acquise dans ce domaine appliqué à la conception d'un compilateur optimisant fait d'ailleurs l'objet du chapitre 7.

- Extensibilité des passes de transformation. La majorité des passes de transformations de code dans GeCoS s'appuie sur le patron de conception *visiteur* qui isole le traitement associé à chaque type d'instruction. Nous avons mis en œuvre un système d'extension de ces visiteurs (basé sur l'infrastructure d'Eclipse) qui permet d'étendre dynamiquement le comportement d'une transformation à n'importe quelle représentation intermédiaire spécifique non supportée nativement par GeCoS . C'est ce système qui a notamment été utilisé pour la regénération du code spécifique au NiosII.

- Les travaux de cette thèse ont également largement contribué à l'intégration du modèle polyédrique (cf. chapitre 4) au sein de GeCoS : interface Java pour la manipulation de polyèdres, extraction des zones analysables par cette abstraction (SCoP [10]) et mise en œuvre d'algorithmes d'ordonnancement affine.

3.2.2 Extraction et description d'instructions spécialisées

Les tâches 1 et 2 de notre flot de conception ont pour objectif d'identifier des regroupements d'opérations qui seront éventuellement déportés sur une extension matérielle, sous forme d'instructions spécialisées, pour en accélérer l'exécution. Chaque groupe d'opérations correspond alors à un motif de calcul qui peut avoir une ou plusieurs occurrences isomorphes (définition 7) pour une application donnée.

10. Static Control Part

Définition 7 (Occurrence de motif) *Soit $G(N, E)$ un graphe contenant N nœuds et E liens, une occurrence $m(N_m, E_m)$ d'un motif P dans G est un sous-graphe de G qui est isomorphe à celui de $P(N_p, E_p)$.*

Les correspondances entre les nœuds du motif et ceux du graphe sont définies par les bijections : $N_p \rightarrow N_m$ et $E_p \rightarrow E_m$ avec $N_m \subseteq N$ et $E_m \subseteq E$.

3.2.2.1 Génération de motifs sous contraintes

La génération de motifs (tâche 1 du flot de la figure 3.3) consiste à identifier automatiquement de nouveaux regroupements d'opérations dans la représentation intermédiaire d'une application cible. L'objectif est d'énumérer les candidats potentiels à une exécution déportée sur le matériel dédié d'une extension au processeur. Cette énumération est faite sous certaines contraintes afin d'éviter d'identifier des instructions spécialisées inadaptées à la cible architecturale.

L'algorithme de génération de motifs que nous utilisons a été développé par Kevin Martin [126]. Il énonce le problème sous forme d'un CSP pour faciliter la prise en compte de contraintes spécifiques supplémentaires. Les contraintes supportées actuellement portent sur le nombre d'entrées et de sorties, la connexité, le nombre de nœuds ainsi que le chemin critique d'un motif. Cet algorithme ne fait pas partie du cadre de cette thèse et, pour plus de détails, le lecteur est invité à consulter l'article [126].

3.2.2.2 Extraction de motifs définis à haut niveau

Il est admis que les approches à base d'extraction automatique de motifs pour la synthèse de jeu d'instructions spécialisées sont des outils incontournables, en particulier lorsqu'il s'agit d'offrir une solution « presse-bouton » aux concepteurs de la plate-forme matérielle.

Toutefois, le concepteur de la plate-forme est parfois également le développeur de l'application. Celui-ci dispose donc d'une connaissance précise de l'algorithme et/ou de l'application à porter sur la plate-forme. Il peut donc souhaiter contrôler de manière très fine le nombre ainsi que le type des motifs d'instructions spécialisées extraits par l'outil afin, par exemple, de déterminer lui-même le meilleur compromis performance/surface.

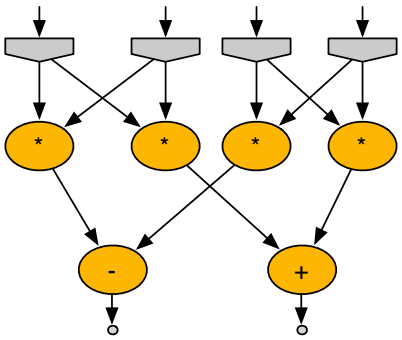
Dans ce type de scénario, le concepteur est généralement amené à utiliser des approches à base de langages de description d'architecture (LISA [88], Tensilica [73]) pour spécifier le nombre et la nature des extensions qu'il souhaite ajouter à son coeur de processeur. Cette étape s'avère cependant rédhibitoire pour de nombreux concepteurs qui sont très réticents à utiliser des langages dédiés qu'ils considèrent souvent comme étant trop complexes et/ou trop bas niveau. Dans ce contexte, la possibilité pour le concepteur de spécifier, directement dans le code source de l'application, les motifs de calcul susceptibles d'être utilisés pour la mise en œuvre de l'extension de jeu d'instructions est donc particulièrement pertinente.

Nous offrons la possibilité (tâche 2 du flot de la figure 3.3) de spécifier, directement dans le code source de l'application, le motif de calcul dont on souhaite déporter l'exécution sur l'extension. Cette fonctionnalité se fait très simplement en créant une fonction C qui réalise le calcul demandé, puis en y associant la directive de compilation `#pragma GCS_PATTERN` illustrée par la figure 3.4.

```c
#pragma GCS_PATTERN
void butterfly(int a, int b,int c,
      int d,int *out1, int *out2){

    *out1 = a*b + c*d;
    *out2 = a*b - c*d;
}
```

(a) Description C du motif



(b) Motif extrait

FIGURE 3.4 – Un exemple de motif (« papillon » FFT) directement spécifié dans le code source de l'application, et sa représentation sous forme de graphe flot de données.

Seul un sous-ensemble du C peut-être utilisé pour décrire des motifs. En particulier, il n'est pas possible d'utiliser des structures de contrôle (boucles, conditionnelles), ni d'accéder à des tableaux ou de manipuler des pointeurs. Les paramètres de la fonction correspondent aux entrées/sorties du motif, les types d'entrées autorisés se limitent à des scalaires, les types de sorties sont nécessairement des pointeurs sur des scalaires (on suppose que les valeurs des données pointées ne sont qu'écrites et jamais lues).

3.2.2.3 Utilisation d'une bibliothèque existante de motifs

Nous offrons également la possibilité d'utiliser une bibliothèque existante de motifs. Celle-ci est décrite dans un langage dédié dont l'éditeur est intégré à Eclipse. La représentation intermédiaire des motifs est basée sur un graphe dirigé où les nœuds sont typés et où chaque lien correspond à une connexion entre un port de sortie du nœud source et un port d'entrée du nœud destination.

La figure 3.5 montre la description d'un motif dans ce langage dédié. Les types des nœuds proviennent d'un fichier externe (e.g., `gecos.types`) utilisable pour différentes bibliothèques de motifs. Un motif contient un graphe (i.e., un ensemble de nœuds et de liens dirigés) et référence les ports d'entrée et de sortie du motif. Ceux-ci sont identifiés par un simple numéro qui précise leurs positions respectives dans le nœud. Par exemple, le port `n1@1` correspond au deuxième port d'entrée ou de sortie du nœud $n1$. Si ce port est utilisé comme une source d'un lien, il s'agit alors d'un port de sortie et réciproquement. De plus, il est possible d'ajouter une information sur la latence matérielle (en nano secondes) d'un motif qui donnera une information plus fine à l'algorithme d'ordonnancement qu'une latence calculée à partir du chemin critique des opérateurs du motif.

Cette représentation intermédiaire découple les tâches d'extraction et de génération des motifs de celles de sélection et d'ordonnancement de leurs occurrences dans une application : les motifs extraits ou générés sont enregistrés dans un fichier utilisé ensuite en entrée de la troisième tâche du flot. La représentation intermédiaire, ainsi que son langage dédié, ont été créés avec Xtext [11] (cf. sous-section 7.3.2 du chapitre 7) afin d'offrir un environnement d'édition intégré à Eclipse (autocomplétion contextuelle, coloration syntaxique, etc.) qui simplifie son utilisation. Cet éditeur (cf. figure 3.6) permet, par exemple, de notifier l'utilisateur de la présence d'isomorphismes potentiels entre deux

11. http://www.eclipse.org/Xtext/

```
import "gecos.types"; //Import available types for nodes

pattern butterfly {
  directed graph {
    n0(mul_INT32); //Create a node of mul_INT32 type
    n1(mul_INT32);
    n2(mul_INT32);
    n3(mul_INT32);
    n4(sub_INT32);
    n5(add_INT32);

    n0@0 -> n4@0; //Edge from the first output port of n0 to the first input port of n4
    n1@0 -> n5@0;
    n2@0 -> n4@1;
    n3@0 -> n5@1;
  }
  latency : 11346; //hardware latency of the pattern (in ns)
  inputs : in0(n0@0,n1@0), in1(n0@1,n1@1), in2(n2@0,n3@0), in3(n2@1,n3@1);
  outputs : out0(n4@0), out1(n5@0);
}
```

FIGURE 3.5 – Description du « papillon » FFT dans un langage dédié.

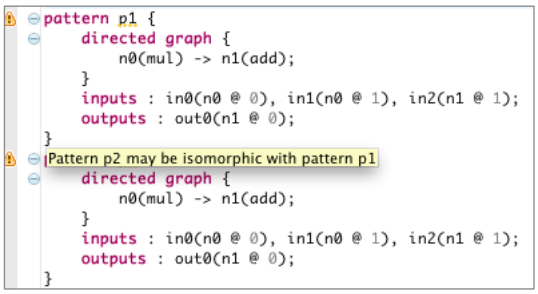


FIGURE 3.6 – Capture d'écran de l'éditeur du langage dédié à la description de motifs : les motifs $p1$ et $p2$ sont isomorphes, une alerte en notifie l'utilisateur.

motifs. L'algorithme de détection utilisé se contente d'analyser le nombre de nœuds, leurs types et les labels des liens (i.e. le type de l'opération source et celui de la destination), pour éviter d'être trop coûteux.

3.2.2.4 Instructions spécialisées pour le NiosII

Les motifs identifiés constituent autant d'instructions spécialisées potentielles et la sélection de leurs occurrences dans une application impliquera de concevoir une extension matérielle chargée de leur exécution. Le contrôle de cette extension est alors spécifique au processeur extensible ciblé.

Le processeur extensible NIOSII de la société Altera est une cible possible de notre flot d'extension de jeu d'instructions. Il s'agit d'un processeur RISC [12] 32 bits, configurable et synthétisable sur un FPGA, il dispose d'une file de 32 registres de 32 bits et son pipeline d'exécution peut contenir un, cinq ou six étages. Ce processeur sera utilisé dans le reste de cette thèse comme la cible matérielle

12. Reduced Instruction Set Computer

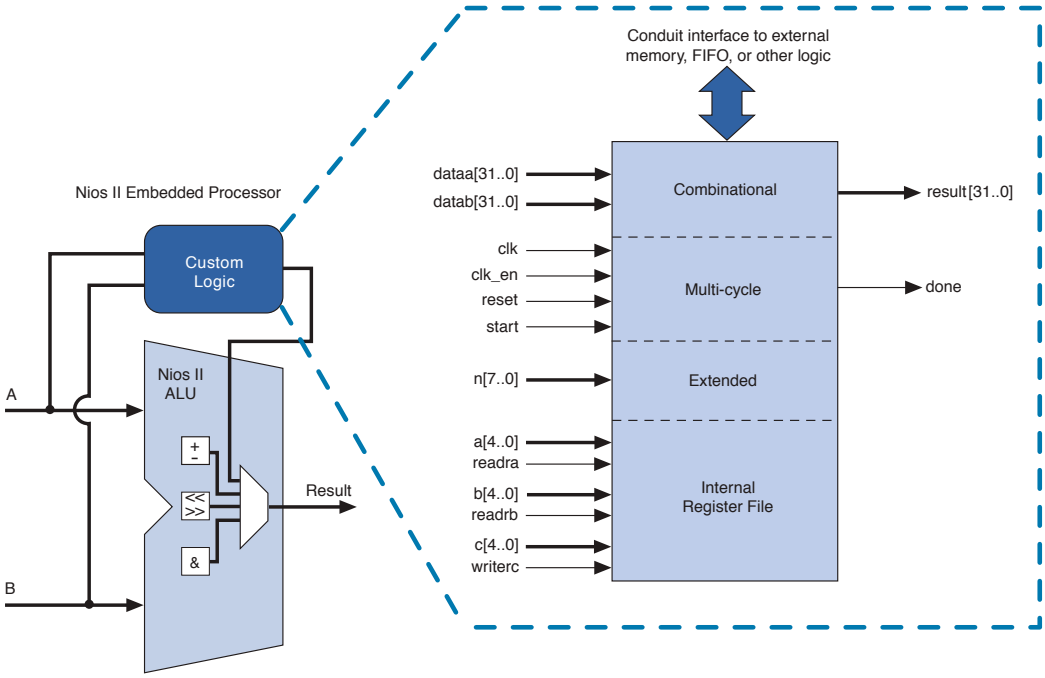


FIGURE 3.7 – Extension matérielle du NIOSII [7].

des différentes approches proposées.

Il est possible de coupler fortement le NIOSII à une extension matérielle (cf. figure 3.7) dans l'optique d'étendre son jeu d'instructions par de nouvelles instructions spécialisées exécutées sur l'extension. L'interface entre l'extension et le processeur varie en fonction du type d'instruction spécialisée :

- *Combinatoire.* Une instruction spécialisée combinatoire s'exécute en un seul cycle d'horloge, les deux opérandes 32 bits sont fournis par les signaux *dataa* et *datab*, le résultat correspond au signal *result* en sortie du bloc logique.
- *Multicycle.* Une instruction spécialisée multicycle gèle le pipeline du processeur jusqu'à la fin de son exécution dont la durée est fixe (indiquée lors de la conception du processeur) ou variable. Lorsque la durée est variable, les signaux *start* et *done* indiquent respectivement le lancement de l'exécution et la fin de l'instruction spécialisée : quand le signal *done* est envoyé par l'extension, le résultat peut être lu par le processeur.
- *Étendue.* Une instruction étendue indique que l'extension matérielle supporte plusieurs opérations spécifiques qui sont identifiées par un numéro unique (codé sur 8 bits). Une instruction étendue peut être combinatoire ou multicycle.
- *Registres internes.* Ce type d'instruction spécialisée s'applique quand le bloc logique contient sa propre file de registres internes. Il est alors possible de lire une donnée dans la file de registres du processeur ou encore dans celle de l'extension. Les signaux a, b et c identifient les registres des opérandes et les signaux *readra*, *readrb* et *writerc* précisent si c'est la file de registres du processeur (état haut) ou celle du bloc logique (état bas) qui est utilisée.

Le format d'une instruction spécialisée pour le NIOSII est détaillé dans la figure 3.8. Une instruction spécialisée est codée sur 32 bits et contient les zones relatives à chaque type d'instruction spécialisée

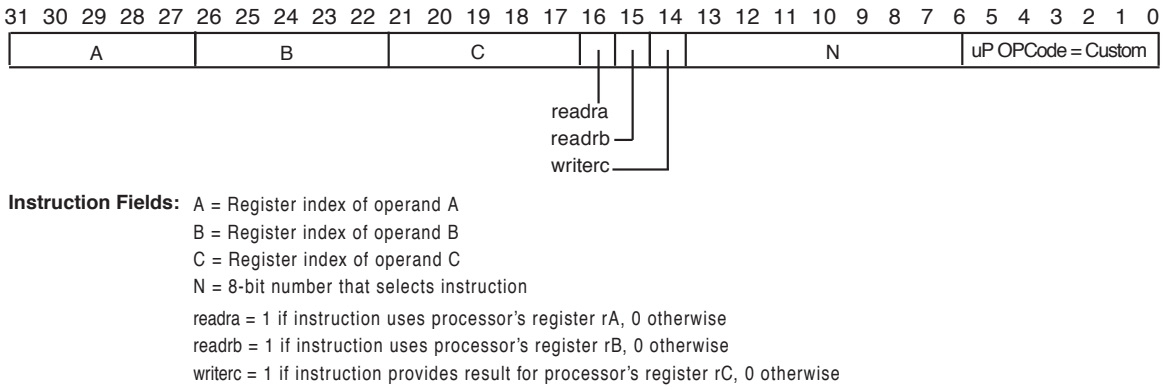


FIGURE 3.8 – Format d'une instruction spécialisée pour le NIOSII [7].

évoqué précédemment. Toutes les instructions spécialisées partagent le même *opcode* (6 premiers bits de l'instruction) et le choix de l'opération est codé par N (256 opérations différentes au maximum). Les bits restants codent les registres utilisés pour les opérandes d'entrée et de sortie de l'instruction.

En fonction des contraintes architecturales de la cible, il est possible qu'un même motif soit décomposé en plusieurs instructions spécialisées. Par exemple, dans le cas du NIOSII qui ne dispose que de deux bus d'opérandes en entrée, un motif opérant sur quatre données nécessitera deux instructions : la première transmet les deux premières données du processeur vers l'extension, la seconde transmet les données restantes et configure le matériel pour effectuer le calcul du motif. Ces contraintes architecturales compliquent considérablement la tâche de sélection des occurrences : un motif dont la taille est la plus importante n'est pas systématiquement celui qui accélère le plus l'application.

3.2.3 Sélection et ordonnancement d'instructions pour un processeur extensible

La selection des occurrences de motifs à exécuter sur une extension (tâche 3 du flot) est un problème d'optimisation pouvant tenir compte de nombreux critères [63] : durée d'execution, nombre de motifs sélectionnés, nombre d'occurrences sélectionnées, nombre de nœuds couverts, surface matérielle utilisée, etc. La pertinence de cette sélection est fortement liée aux contraintes materielles de l'architecture (partage des ressources de calcul, de mémoire et de communication) qui limitent les possibilités d'ordonnancement des occurrences de motifs pour une application donnée.

La majorité des algorithmes de sélection d'instructions spécialisées ne cherchent pas à déterminer un ordonnancement de l'application analysée. Ils se contentent généralement d'évaluer le gain potentiel qu'apporte la sélection de chacune des instructions spécialisées [12, 29, 104] ou encore de minimiser le nombre total d'instructions requises pour exécuter l'application analysée [44, 81]. Le risque de telles approches est que l'ordonnancement ultérieur des instructions spécialisées sélectionnées sera finalement moins performant qu'avec une sélection plus adaptée aux contraintes architecturales.

Les techniques de sélection que nous proposons réalisent l'ordonnancement et la sélection des instructions spécialisées en une seule étape d'optimisation. Le principe est de formuler un problème de satisfaction de contraintes (CSP [13], cf. chapitre 2) modélisant de manière modulaire et extensible, les différents aspects du problème.

- Contraintes de couverture de graphe : sélection d'une occurrence de motif pour chaque nœud du graphe de l'application (sous-section 3.3.1), un nœud ne pouvant être couvert que par une unique occurrence.
- Contraintes temporelles : ordonnancement de chaque occurrence en respectant les dépendances de données du graphe. L'objectif est alors de minimiser la durée totale d'une exécution parallèle sans contraintes de ressources (sous-section 3.3.2).
- Contraintes architecturales : une occurrence sélectionnée peut être exécutée soit sur le processeur soit sur l'extension. Dans les deux cas, ces ressources ne peuvent traiter plus d'une occurrence à la fois, elles doivent donc être partagées dans le temps. Nous nous intéresserons tout d'abord à une architecture composée d'un NiosII couplé à une extension séquentielle (section 3.4). Nous étudierons ensuite une autre architecture où l'extension dispose cette fois d'unités de traitement parallèles (section 3.5).

Après résolution du problème avec un solveur de contraintes, chaque nœud de l'application est couvert par une occurrence ordonnancée et affectée au processeur ou à une ressource d'exécution de l'extension. Il est ensuite possible, à partir de ces informations, de générer une nouvelle version du code source qui profitera de l'extension matérielle en utilisant les instructions spécialisées identifiées.

3.2.4 Génération du code et synthèse de l'extension matérielle

3.2.4.1 Modèles d'architecture de l'extension pour le NiosII

Deux modèles d'architecture sont actuellement supportés par notre flot. Ils sont illustrés par la figure 3.1. La figure 3.1.a décrit une architecture composée d'un processeur extensible (e.g., NiosII) et d'un *bloc reconfigurable* contenant le matériel nécessaire à l'exécution des occurrences de motifs sélectionnées dans le graphe d'une application. L'architecture 3.1.b illustre, quant à elle, le cas où l'extension matérielle contient plusieurs blocs reconfigurables pouvant s'exécuter en parallèle. Ces blocs sont interconnectés par un réseau de communication *full-crossbar* et contiennent potentiellement des mémoires qui sont utilisées pour stocker les données externes (i.e., liens entrants ou sortants du graphe) sans solliciter le processeur pour accéder à la mémoire principale. D'autre part, les données transférées du processeur et celles qui sont produites sur l'extension (sans intervention du processeur) peuvent être mémorisées dans une file de registres interne à l'extension pour une utilisation ultérieure sur l'un des blocs.

Si un bloc de l'extension supporte plusieurs instructions spécialisées, le chemin de données qui met en œuvre l'ensemble de ces opérations dispose alors de multiplexeurs qui seront (re)configurés à l'exécution, en interprétant l'identifiant de l'instruction spécialisée transmis par le processeur. De plus, les blocs reconfigurables peuvent être hétérogènes. Chaque bloc supporte alors un ensemble d'opérations complexes qui lui est propre. Cependant, il est important de noter que la multiplication des blocs ainsi que du nombre d'opérations supportées par chacun d'entre eux augmentera le nombre

13. Constraint Satisfaction Problem

```
1  void custom_butterfly(int a, int b, int c,int d, int *out1, int *out2){
2    asm volatile("
3      custom 1, %0, %1, zero;
4      custom 2, %2, %3, %4;
5      custom 3, zero, zero, %5;"
6      :"r"(a)
7      :"r"(b)
8      :"r"(c)
9      :"r"(d)
10     :"=r"(*out1)
11     :"=r"(*out2)
12   );
13 }
```

FIGURE 3.9 – Fonction C qui exécute le motif de la figure 3.4 sur l'extension, trois instructions spécialisées sont nécessaires : 1) envoi des deux premiers opérandes 2) envoi des autres opérandes, calcul et récupération du résultat 3) récupération du deuxième résultat.

d'instructions spécialisées à supporter dans l'architecture. Dans le cas du NIOSII, le nombre total d'instructions spécialisées différentes ne pourra excéder 256 (cf. paragraphe 3.2.2.4, page 51).

3.2.4.2 Compilation NiosII

À chaque motif peut correspondre une ou plusieurs occurrence(s) sélectionnée(s) dont les contextes d'exécution (i.e., connexions entre les registres et les opérateurs matériels) peuvent varier d'une occurrence à l'autre. Il est donc possible que pour un même bloc reconfigurable, les configurations des multiplexeurs soient différentes pour plusieurs des occurrences d'un même motif. Or, ce sont les instructions spécialisées qui identifient les différentes configurations. Puisque leur nombre est limité par les capacités du processeur cible, il est important de minimiser le nombre de configurations en optimisant l'allocation des registres pour chaque bloc. De plus, la surface occupée par l'extension matérielle sera d'autant plus faible que le nombre de configurations différentes par bloc (et donc de multiplexeurs) sera réduit. Ce problème d'optimisation n'entre pas dans le cadre de cette thèse et pour plus de détails, le lecteur est invité à consulter la thèse de Kevin Martin [125] ou l'article [128].

Une fois que les différentes configurations d'un même motif ont été identifiées, chacune de ses occurrences sélectionnées lors de la couverture est remplacée, dans la représentation intermédiaire (e.g., DAG, HCDG), par un nouveau nœud symbolisant l'ensemble des instructions spécialisées associées. Une nouvelle version du code C qui exploite ces instructions (e.g., par des instructions assembleur *en ligne*) est ensuite regénérée (tâche 4 du flot, Figure 3.2). Cette version sera ensuite compilée par le compilateur natif du NIOSII pour produire un binaire exécutable sur l'architecture.

La Figure 3.9 montre le code C regénéré à partir du motif de la Figure 3.4. La zone de code spécifique est délimitée par la construction `asm volatile` de GCC qui définit une séquence d'instructions assembleur ne pouvant être optimisées par le compilateur. Dans cette zone, chaque instruction spécialisée est écrite dans une syntaxe assembleur :

```
custom <id>, <r1>,<r2>,<r3> ;
```

dont les paramètres (`%0,...,%5`) sont des registres du processeur. Leur allocation est cependant laissée au soin du compilateur : il suffit d'indiquer explicitement l'association entre chacun des paramètres et

des symboles du code C (lignes 6-11). L'utilisation du registre d'une variable est spécifiée par `:"r"(a)` (lignes 6-9) et l'écriture par `:"=r"(a)` (lignes 10-11).

L'exemple de la figure 3.9 illustre la nécessité d'instructions supplémentaires pour la transmission et la réception des données entre le processeur et l'extension (cf. paragraphe 3.2.2.4, page 53) : une instruction supplémentaire (ligne 3) est nécessaire pour récupérer les deux premières données du motif. L'instruction suivante (ligne 4) charge les deux opérandes restants et retourne le premier résultat produit sur le matériel dédié. De même que pour les opérandes d'entrée, une instruction supplémentaire (ligne 5) est requise pour rapatrier le dernier résultat.

Mis à part le code spécifique aux zones exécutées sur l'extension matérielle, le reste du programme exploitant les instructions spécialisées respecte la syntaxe haut niveau du C. Afin de limiter l'effort de développement dans la mise en œuvre du regénérateur de code, celui-ci exploite un des points forts de l'infrastructure GeCoS, à savoir son extensibilité. Il est en effet possible d'étendre la représentation intermédiaire de base utilisée par GeCoS, en y ajoutant de nouvelles constructions, ou en spécialisant des constructions existantes. Les passes d'analyse et d'optimisation existantes peuvent alors être étendues pour supporter ces extensions (sans nécessiter de modification de leur code source) à l'aide d'un système de *points d'extension* qui se charge de réaliser, à l'exécution, l'édition de liens entre le cœur de l'infrastructure et les greffons qui lui sont associés. Le générateur du code spécifique au NiosII utilise cette propriété et se contente d'ajouter les fonctions nécessaires au traitement des instructions spécifiques qui n'existent pas dans la représentation intermédiaire standard de GeCoS.

3.2.4.3 Synthèse de l'extension

L'extension à synthétiser est composée d'un ensemble de blocs de calcul correspondant aux motifs sélectionnés. Un langage dédié permet de décrire simplement les blocs de l'extension et les différents contextes de configurations supportés. Chaque contexte correspond à une instruction spécialisée indiquant les correspondances entre les ports d'entrée/sortie des motifs et l'interface et/ou des registres de l'extension. Ce fichier de description de l'extension est généré à partir des informations obtenues à la fin de l'étape précédente et est utilisé pour générer du VHDL synthétisable.

De même que pour la génération des motifs et l'identification des configurations de l'extension, la synthèse de l'extension n'entre pas dans le cadre de cette thèse et pour plus de détails, le lecteur est invité à consulter la thèse [125]. D'autre part, la génération automatisée de la description matérielle n'est pas, à ce jour, entièrement fonctionnelle et explique l'absence de résultats de synthèse dans nos différentes expérimentations.

3.3 Sélection et ordonnancement sans contraintes de ressources

Le flot de conception présenté dans la section précédente s'appuie sur une étape de sélection et d'ordonnancement des instructions spécialisées. Pour chaque graphe G de la représentation intermédiaire de l'application, l'algorithme de sélection détermine une couverture de G à partir d'une bibliothèque de motifs (PS [14]) identifiés lors des tâches 1 et 2 du flot (cf. figure 3.2, page 45).

L'algorithme présenté dans cette section modélise le problème sous forme d'un CSP dont la résolution sélectionnera, pour chaque nœud $n \in G$, une unique occurrence de motif le contenant. Des

14. Pattern Set

contraintes supplémentaires sont ensuite ajoutées afin de déterminer un ordonnancement des occurrences sélectionnées dont la durée totale est minimale. Dans cette section, on ne considère pas les contraintes issues du partage de ressources, toutes les occurrences peuvent donc s'exécuter en parallèle tant qu'elles respectent les dépendances de données des nœuds.

3.3.1 Couverture d'un graphe par une bibliothèque de motifs

Pour modéliser le problème de couverture, il est nécessaire de connaître toutes les occurrences (M) de chaque motif $P \in PS$ dans G. Si un nœud n se trouve dans une occurrence m alors $n \in N_m$. Dans la suite de ce chapitre nous noterons $n \in m$ par souci de lisibilité et si cela ne prête pas à confusion.

L'ensemble des occurrences de PS dans un graphe G est construit par l'Algorithme 1 qui identifie les occurrences de chaque motif $p \in PS$ en utilisant un algorithme (problème NP-complet) d'isomorphisme de sous-graphe (e.g., Ullman [183], VF2[45]), ou encore une approche basée sur la programmation par contraintes [195, 208, 171]. À la fin de l'algorithme, chaque nœud $n \in N$ est associé à un ensemble $matches_n \subseteq M$ qui contient toutes les occurrences de motifs qui sont susceptibles de couvrir n.

Algorithme 1 Recherche de toutes les occurrences de motifs dans un graphe.

```
1  pour ∀p ∈ PS faire
2     M_p ← Occurrences(G, p)
3     M ← M ∪ M_p
4  fin pour
5  pour ∀m ∈ M faire
6     pour ∀n ∈ m faire
7        matches_n ← matches_n ∪ {m}
8     fin pour
9  fin pour
```

D'autre part, on définit une fonction $size(m)$ qui retourne le nombre de nœuds se trouvant dans une occurrence m. Si $size(m) = 1$ l'occurrence m est alors dite de taille unitaire et est notée m^1.

Pour modéliser la sélection des occurrences dans un CSP, on identifie, de manière unique, chacune des occurrences présentes dans le graphe G. Ainsi, l'occurrence $m_i \in M$ désigne la i-ème occurrence de M et chaque nœud est associé à une variable $match_n$ dont la valeur code l'identifiant de l'occurrence sélectionnée pour couvrir n. Résoudre le problème de couverture consiste alors à déterminer, pour chaque nœud $n \in N$, la valeur de la variable $match_n$. Ainsi, le nombre de variables de décision est proportionnel au nombre de nœuds, contrairement à l'approche de Martin et al. [127] où ce nombre était proportionnel au nombre d'occurrences [15]. Le tableau 3.1 montre les domaines, avant et après sélection, des variables de sélection ($match_n$) de chacun des nœuds présents dans le graphe couvert de la figure 3.10.

Notre méthode de couverture du graphe n'autorise pas le chevauchement des occurrences : un nœud ne peut être couvert que par une unique occurrence (variable de décision $match_n$). Cette restriction implique que si une occurrence est sélectionnée, alors tous les nœuds qui la composent sont couverts par cette même occurrence. Afin de garantir le respect de cette propriété, on impose la contrainte suivante pour chaque occurrence candidate à la sélection :

15. Le nombre d'occurrences est souvent bien supérieur au nombre de nœuds.

Variables	Domaine initial	Domaine après sélection
$match_{n_1}$	$\{0, 7\}$	$\{0\}$
$match_{n_2}$	$\{1, 7\}$	$\{1\}$
$match_{n_3}$	$\{2, 6, 8\}$	$\{2\}$
$match_{n_4}$	$\{4, 8\}$	$\{4\}$
$match_{n_5}$	$\{3\}$	$\{3\}$
$match_{n_6}$	$\{2, 5, 7\}$	$\{2\}$

TABLE 3.1 – Les domaines des variables $match_n$ pour tous les nœuds après initialisation et sélection des occurrences pour la couverture de la figure 3.10.

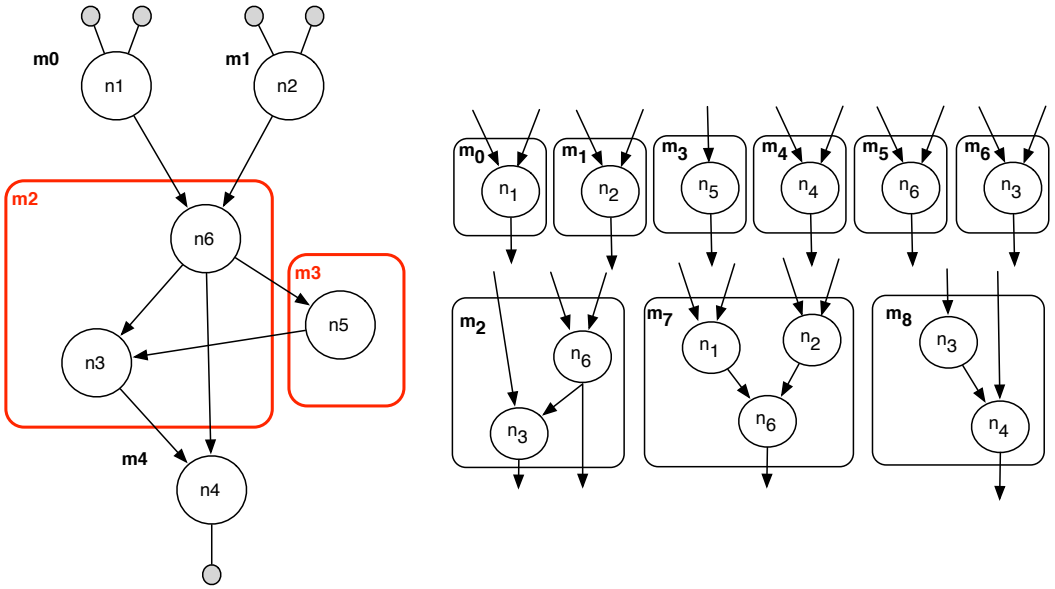


FIGURE 3.10 – Exemple de couverture avec des occurrences non-convexes (m_2,m_3) : invalide dans le cadre d'un ordonnancement.

Contrainte 1 (Sélection d'une occurrence) *Une occurrence $m_i \in M$ n'est sélectionnée que si elle couvre tous ses nœuds et réciproquement :*

$$\forall m_i \in M : Count(i, list_{m_i}, count_{m_i}) \tag{3.1}$$

$$\forall m_i \in M : count_{m_i} > 0 \Leftrightarrow sel_{m_i} \tag{3.2}$$

La contrainte globale `Count` utilisée dans (3.1) détermine si une occurrence m_i est sélectionnée ou non. Elle compte le nombre de fois ($count_{m_i}$) où l'identifiant de m_i est une valeur possible pour chacune des variables de $list_{m_i}$: $\{\forall n \in m_i, match_n\}$. Le domaine de la variable $count_{m_i}$ ne contient que deux valeurs : zéro ou $size(m_i)$, les nœuds de m_i sont donc soit tous couverts par m_i ($match_n = i$) soit couverts par une autre occurrence ($match_n \neq i$). La contrainte (3.2) définit la valeur de la variable sel_{m_i} en utilisant une contrainte « réifiée » (*reified*). Cette variable vaut un si l'occurrence

est sélectionnée, et zéro sinon. Réciproquement, si une occurrence est sélectionnée ($sel_{m_i} = 1$), la variable $count_{m_i}$ n'est pas nulle et la contrainte (3.1) implique alors que $\forall n \in m_i, match_n = i$.

L'annexe A.1 détaille la modélisation ARCAdE (cf. chapitre 8) du problème de couverture de graphe et propose deux stratégies de résolution : recherche d'une couverture et minimisation du nombre d'occurrences sélectionnées.

3.3.2 Ordonnancement temporel et couverture simultanée

Le problème de couverture formulé précédemment est maintenant étendu à un ordonnancement qui respecte les dépendances de données du graphe et qui minimise la durée d'exécution totale du graphe couvert. La qualité de cet ordonnancement dépend des occurrences sélectionnées puisque les durées d'exécution des occurrences varient éventuellement d'une occurrence à l'autre et seront, logiquement, inférieures à la somme des durées des nœuds qu'elles couvrent.

3.3.2.1 Variables temporelles

Pour énoncer les contraintes d'un ordonnancement, on déclare de nouvelles variables pour chaque occurrence $m_i \in M$:

- $start_{m_i} :: [0..\infty]$: début de l'exécution d'une occurrence [16].
- $delay_{m_i}$: durée d'une occurrence (constante ou variable).

De même, à chaque nœud $n \in N$, on associe les variables :

- $start_n :: [0..\infty]$: début de l'exécution de l'occurrence qui couvre n.
- $delay_n :: [0..\infty]$: durée de l'occurrence qui couvre n.

3.3.2.2 Contraintes d'ordonnancement

Afin d'éviter d'exprimer les dépendances de données pour chaque combinaison d'occurrences candidates pour les nœuds source et destination d'une dépendance, notre modèle d'ordonnancement se base sur les variables temporelles des nœuds et non sur celles des occurrences. La difficulté consiste alors à modéliser le caractère dynamique de la durée d'un nœud n, celle-ci variera en effet selon l'occurrence sélectionnée pour couvrir le nœud n.

Contrainte 2 (Couverture d'un nœud) *Pour chaque nœud $n \in G$, le début de l'exécution et la durée de n correspondent aux variables d'ordonnancement respectives de l'occurrence sélectionnée par $match_n$:*

$$\forall_{n \in N}\ Element(match_n, List_{start}, start_n), \tag{3.3}$$

$$\forall_{n \in N}\ Element(match_n, List_{delay}, delay_n) \tag{3.4}$$

$$avec\ List_{start} = [start_m \,|\, m \in M]\ et\ List_{delay} = [delay_m \,|\, m \in M]$$

16. ∞ est un entier représentant l'infini dans le problème.

La contrainte 2 lie dynamiquement (c'est-à-dire en fonction de l'état des variables au cours du processus de résolution du CSP) les variables temporelles d'un nœud à celles de l'occurrence sélectionnée. La contrainte *Element* impose en effet une relation entre une variable d'indice (I), un vecteur de variables (L) et un résultat (R) selon la formule :

$$R = L[I]$$

qui contraint, ici, les valeurs des variables temporelles des nœuds à celles de l'occurrence sélectionnée en utilisant $match_n$ comme un indice. Les contraintes (3.1) et (3.2), en conjonction avec les contraintes (3.3) et (3.4), impliquent que les nœuds d'une occurrence sélectionnée commencent tous au même moment et ont tous la même durée.

Le respect des dépendances de données est assuré par la contrainte (3.5) qui n'est appliquée que pour les liens du graphe connectant au moins deux occurrences différentes. Dans le cas contraire, les nœuds source et destination sont nécessairement dans la même occurrence, aucune contrainte n'est donc requise.

Contrainte 3 (Respect d'une dépendance de données) *Soit s et d, les nœuds respectivement source et destination d'un lien $e \in E$. Si les nœuds s et d ne sont pas couverts par la même occurrence, le nœud d ne peut alors débuter qu'après la fin de l'exécution de s.*

$$\forall_{(s,d)\in E} \quad IF\ match_s \neq match_d\ \ THEN\ start_d \geq start_s + delay_s \tag{3.5}$$

L'identification des occurrences de motifs peut faire apparaître des occurrences non convexes qui, si elles sont sélectionnées, introduisent des cycles dans le graphe couvert et celui-ci sera donc impossible à ordonnancer. Les contraintes temporelles (3.3, 3.4 et 3.5) suffisent à garantir la légalité de l'ordonnancement et donc la convexité des occurrences sélectionnées. Par exemple, si l'occurrence m_2 de la figure 3.10 est sélectionnée, alors :

$$(start_{m_2} = t_{n_6} = t_{n_3}) \wedge (start_{m_3} = t_{n_5}) \wedge (t_{n_6} + d_{n_6} \leq t_{n_5}) \wedge (t_{n_5} + d_{n_5} \leq t_{n_3})$$

ce qui est impossible car $(d_{n_5} = delay_{m_3} \neq 0) \wedge (d_{n_6} = d_{n_3} = delay_{m_2} \neq 0)$.

Il est toujours possible de vérifier, en amont de la couverture, qu'une occurrence est convexe ou non. Par exemple, l'occurrence m_2 dans la figure 3.10 étant clairement non convexe, elle ne pourra pas être sélectionnée (elle n'est présentée ici que pour illustrer la problématique). Lors de l'identification des occurrences, celles qui ne sont pas convexes sont rejetées par un algorithme de filtrage dont le coût est négligeable puisque les motifs ne contiennent généralement qu'un nombre relativement faible de nœuds. Cependant, cette étape ne suffit pas. Il peut en effet arriver qu'un cycle apparaisse en sélectionnant deux occurrences qui sont pourtant convexes.

Ce phénomène est illustré par la figure 3.11 pour un exemple de « papillon » FFT. Si l'on dispose des motifs de la figure 3.11(b), leurs seules occurrences (figure 3.11(c)) dans le graphe sont convexes et pourtant, les sélectionner introduit un cycle qui apparait clairement dans la figure 3.11(d).

Les contraintes (3.5) issues des dépendances de données n'autoriseront pas la sélection de telles occurrences. Elles sont donc nécessaires à toute étape de sélection et ceci y compris dans le cas où le graphe couvert serait ordonnancé pas un autre algorithme exécuté après la couverture du graphe.

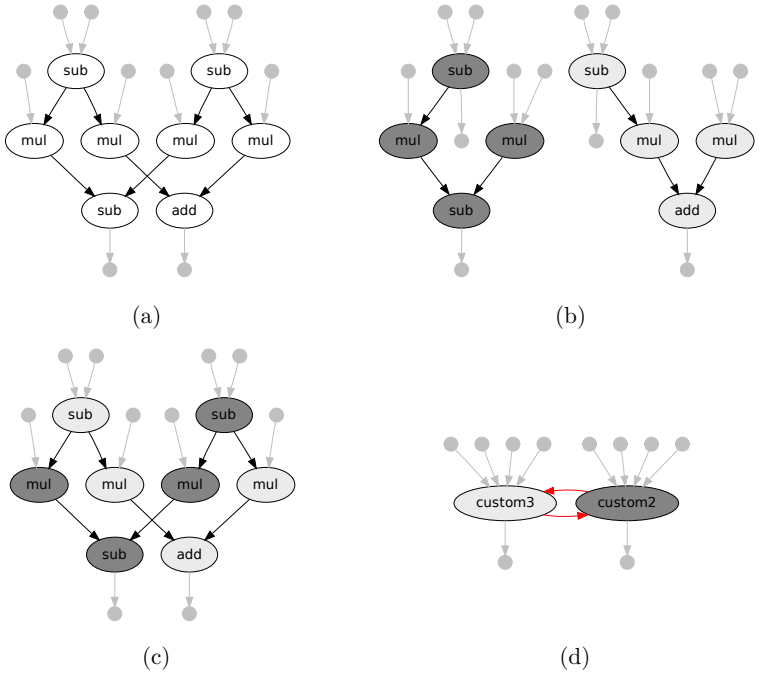


FIGURE 3.11 – Exemple de deux occurrences convexes et pourtant impossibles à ordonnancer.

3.3.2.3 Minimisation de la durée d'exécution

L'ordonnancement recherché minimise la durée d'exécution de la totalité du graphe d'application. Il s'agit donc de minimiser la date de fin du dernier nœud exécuté :

$$cost(G) = \texttt{Max}([t_n + d_n | \forall n \in N]) \tag{3.6}$$

L'annexe A.3 détaille la modélisation ARCAdE (cf. chapitre 8) du problème de couverture de graphe et d'ordonnancement. Deux stratégies de résolution sont modélisées : celle qui minimise la durée d'exécution parallèle et une autre qui minimise la durée d'exécution séquentielle (somme des durées des occurrences sélectionnées).

3.3.3 Résultats expérimentaux

Pour évaluer les performances de l'algorithme de couverture sans contraintes de ressources, on compare les résultats obtenus avec l'approche de Martin et. al [127] qui modélise les contraintes temporelles, issues des dépendances de données, sur les occurrences de motifs et non sur les nœuds, comme c'est le cas dans notre approche. Les mesures ont été réalisées en utilisant le solveur JaCoP [17] sur une machine disposant d'un processeur Intel core2 duo à 2,4 GHz.

Les applications analysées, écrites en C, sont des applications issues de *benchmarks* dont les thèmes applicatifs sont récurrents dans les systèmes embarqués : multimédia, télécommunications, sécurité, etc. Les applications ont été sélectionnées dans *MediaBench* [113], *MiBench* [83] et *MCrypt* [130] ainsi que *PolarSSL* [144] pour leurs caractéristiques variées, résumées dans le tableau 3.2. La représentation

17. \http://jacop.osolpro.com/

	Régularité	Parallélisme	Chemin critique	Connectivité	Accès mémoire
MCRYPT cast128	moyen	moyen	long	moyen	élevé
MiBench BF encrypt	élevé	moyen	long	moyen	faible
MiBench gsm enc	élevé	élevé	court	moyen	élevé
Mediabench MESA invert matrix	élevé	élevé	moyen	moyen	élevé
Mediabench JPEG IDCT	faible	élevé	court	élevé	moyen
PolarSSL aes	élevé	élevé	moyen	élevé	élevé
PolarSSL des	faible	moyen	moyen	moyen	faible

TABLE 3.2 – Caractéristiques des applications analysées.

intermédiaire GeCoS du CDFG de chaque application est transformée en un HCDG (cf. paragraphe 1.1.2.1, page 16) où seules les opérations de calcul et les accès mémoires sont conservés (les gardes sont ignorées).

Les motifs en entrée de l'algorithme de couverture et d'ordonnancement sont générés pour chaque application, contiennent au maximum dix nœuds et ont, au plus, quatre entrées et deux sorties. Ces paramètres constituent un compromis pragmatique entre les opportunités d'optimisation et la complexité de résolution du problème pour un nombre important de motifs.

L'objectif est d'observer le comportement de notre technique de couverture en considérant que chaque occurrence (peu importe sa taille) est exécutée en un seul cycle. Le graphique de la figure 3.12 montre le gain normalisé de notre approche par rapport à celle décrite dans [127] pour une couverture qui minimise la durée d'exécution parallèle sans contraintes de ressources. Les gains observés vont jusqu'à une amélioration de 20% de la durée totale d'exécution et, dans la majorité des cas, le solveur a réussi à prouver l'optimalité de la solution en un temps inférieur à 1s pour des graphes d'environ 150 nœuds en moyenne (alors qu'avec l'approche [127], aucune solution de ces exemples n'est prouvée comme étant optimale). Pour les applications *cast128* et *BF encrypt*, le solveur n'a pas réussi à prouver l'optimalité dans les limites du temps imparti (30s). Ces applications sont caractérisées par un niveau moyen de parallélisme, un long chemin critique ainsi que de longues dépendances de données qui compliquent la sélection et l'ordonnancement des occurrences. Le solveur aura donc besoin de plus de temps pour trouver (et encore plus pour prouver) la solution optimale.

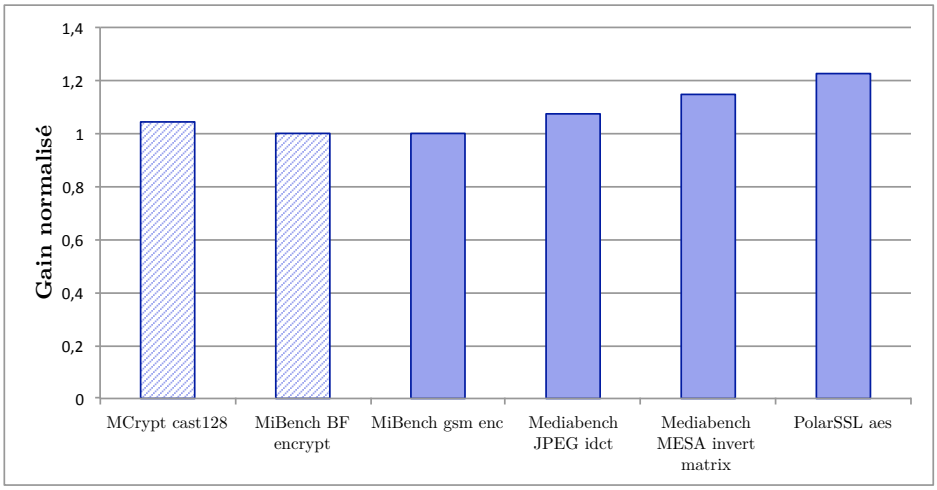


FIGURE 3.12 – Gain normalisé pour la couverture qui minimise la durée d'exécution parallèle par rapport à l'approche [127].

D'autre part, on évalue le passage à l'échelle de l'algorithme de couverture en mesurant le temps de résolution du problème de couverture/ordonnancement pour des DAG générés aléatoirement. Pour cela, on utilise un générateur de graphe acyclique qui crée autant de nœuds N que souhaité et les connecte aléatoirement par un nombre de liens $E = \sqrt{N}$ en respectant la contrainte que chaque nœud ne dispose que de deux prédécesseurs. Pour éviter une étape de génération de motifs trop coûteuse, on considère que tous les nœuds sont de même type et on génère l'ensemble des motifs (4 entrées, 2 sorties, 10 nœuds) sur un graphe, également généré aléatoirement, de 50 nœuds. Le graphe à couvrir disposera néanmoins d'un nombre intéressant d'occurrences pour les motifs générés puisque tous les nœuds ont le même type (cf. figure 3.13).

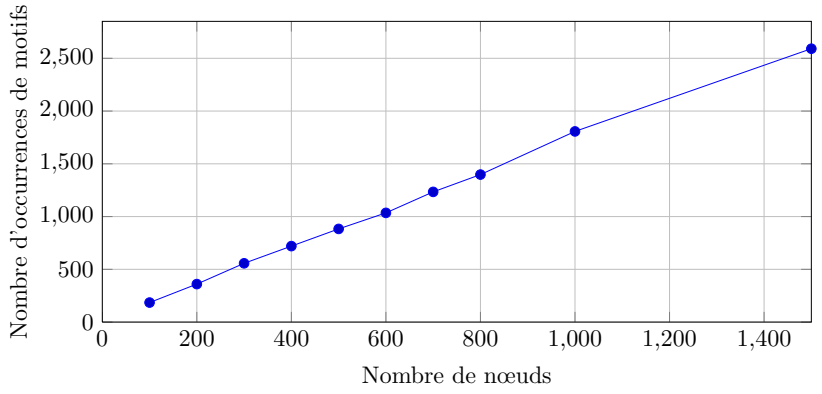


FIGURE 3.13 – Evolution quasi-linéaire du nombre d'occurrences de motifs en fonction du nombre de nœuds dans les graphes générés.

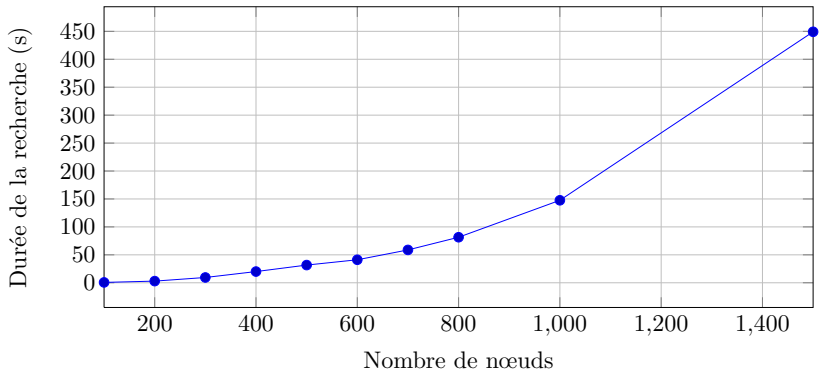


FIGURE 3.14 – Temps de résolution de la couverture qui minimise la durée d'exécution parallèle pour des DAG aléatoires.

La figure 3.14 montre les résultats obtenus pour des graphes contenant entre 100 et 1500 nœuds en mesurant la durée moyenne de résolution de la couverture de 5 graphes aléatoires par nombre de nœuds évalué. De plus, pour améliorer le passage à l'échelle, on utilise une résolution par apprentissage : la durée de résolution est limitée à dix secondes et si aucune solution n'est identifiée en ce temps imparti, les échecs repérés lors de la recherche (cf. chapitre 2, page 40) sont utilisés comme autant de contraintes additionnelles d'une autre recherche également limitée en temps. Ces contrain-

tes additionnelles, appelées *nogood*, référencent les combinaisons de valeurs d'une liste de variables qui conduisent à un échec. Elles évitent ainsi d'explorer, une nouvelle fois, un sous-espace qui mène de toute façon à un échec. L'algorithme s'achève lorsqu'une recherche a réussi à trouver une solution optimale ou encore une solution valide dans le cas où la durée maximale de résolution est atteinte.

Ces résultats montrent que notre technique de couverture permet de traiter des graphes allant jusqu'à 800 nœuds en un temps raisonnable (moins de deux minutes) mais qu'au delà, la durée de résolution peut s'avérer rédhibitoire et dépasse, par exemple, les sept minutes pour 1500 nœuds.

3.4 Processeur couplé à une extension séquentielle

La technique de couverture présentée dans la section précédente est ici adaptée à la sélection et l'ordonnancement d'instructions spécialisées pour un processeur couplé à une extension matérielle. Celle-ci ne permet d'exécuter qu'un seul motif de calcul à la fois, on parle donc d'une extension séquentielle (cf. figure 3.15).

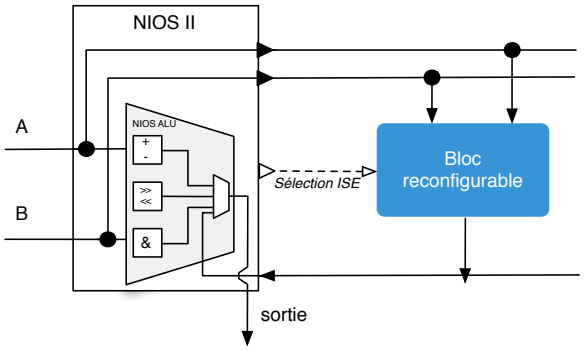


FIGURE 3.15 – Processeur NIOSII couplé à une extension séquentielle.

3.4.1 Architecture de l'extension

Bien qu'un seul motif puisse être exécuté sur l'extension, il existe néanmoins un parallélisme potentiel entre les instructions exécutées sur le processeur et celles qui sont déportées sur l'extension : si une instruction spécialisée occupe l'extension durant plusieurs cycles, le processeur est disponible pour exécuter d'autres instructions primitives.

Pour respecter les contraintes de l'architecture ciblée, un ordonnancement doit respecter les contraintes suivantes :

- Une donnée produite sur l'extension reste sur l'extension si elle est utilisée plus tard par une autre occurrence et elle est alors stockée dans les registres internes de l'extension. Le nombre de registres disponibles n'est pas limité lors de la couverture et de l'ordonnancement. C'est une étape ultérieure d'allocation de registres qui optimisera le nombre de registres réellement nécessaires.
- Une donnée produite sur l'extension est envoyée au processeur si l'occurrence qui la référence est exécutée sur le processeur (occurrence de taille unitaire m^1) ou s'il s'agit d'un lien externe (sortie du graphe analysé).

- Si une occurrence lit plus de deux opérandes venant du processeur, des cycles additionnels sont requis pour faire parvenir toutes les données (deux données au maximum par cycle).
- Si une occurrence envoie plus d'un résultat au processeur, des cycles additionnels sont également nécessaires.
- L'extension est réservée durant l'intégralité du traitement d'une occurrence sélectionnée : alimentation en données, exécution et récupération des résultats.
- Le processeur est réservé durant les phases d'initialisation, de lancement et de récupération des résultats d'une occurrence sélectionnée. Il ne sera disponible qu'au milieu de l'exécution d'une instruction spécialisée multicycle.

3.4.2 Modèle de contraintes

3.4.2.1 Variables utilisées

Pour énoncer les contraintes d'ordonnancement ainsi que celles du partage du processeur, on déclare de nouvelles variables pour chaque occurrence $m_i \in M$.

- $ERT_{m_i} :: [0..\infty]$: pénalité d'alimentation en données issue du nombre limité de bus en entrée de l'extension.
- $WRT_{m_i} :: [0..\infty]$: pénalité de récupération des résultats issue du nombre limité de bus en sortie de l'extension.
- $processing_{m_i} :: Constante$: durée constante d'exécution d'une occurrence (paramètre du flot).
- $swr_{m_i} :: [0..\infty]$: début de la récupération des résultats produits par une occurrence de motif exécutée sur l'extension.
- $NbInFromProcessor_{m_i} :: [0, m_i.inputs.size()]$: compte le nombre d'entrées d'une occurrence qui proviennent du processeur.
- $NbOutToProcessor_{m_i} :: [0, m_i.outputs.size()]$: compte le nombre de sorties d'une occurrence qui sont transmises au processeur.
- $IsWriteTransfer_{m_i} :: [0, 1]$: indique si il existe au moins une donnée produite dans l'occurrence qui est transmise au processeur.
- $IsMultiCycle_{m_i} :: Constante \in \{0, 1\}$: indique si la durée d'une occurrence est multicycle.

Pour chaque nœud $n \in N$, on déclare la variable :

- $isOutToProcessor_n :: [0, 1]$: indique si le nœud produit une donnée à transmettre au processeur.

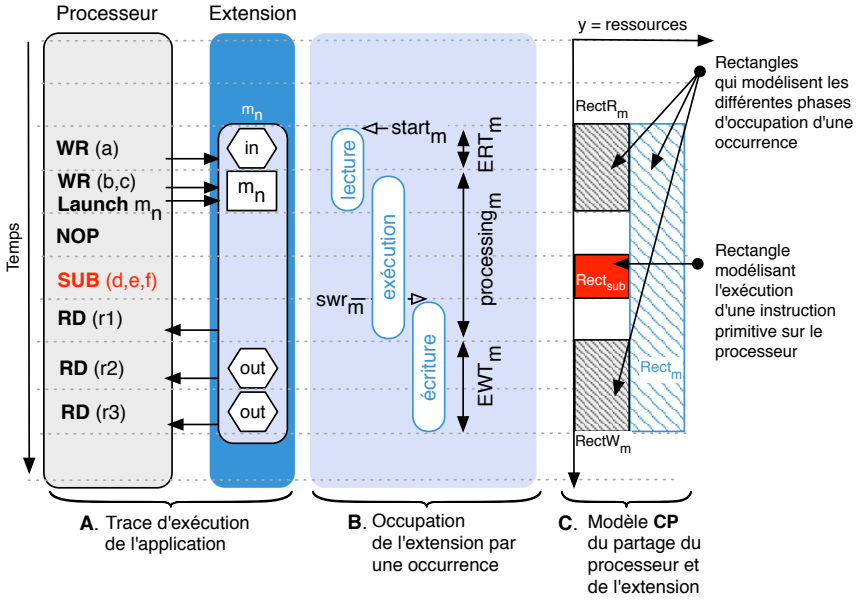


FIGURE 3.16 – Modèle d'exécution sur l'extension séquentielle.

3.4.2.2 Contraintes de partage des ressources

Par défaut, on considère que les occurrences qui ne contiennent qu'un seul nœud, correspondent aux instructions du processeur. Celles qui sont sélectionnées seront exécutées sur le processeur et les autres (i.e., celles de taille supérieure à un) seront exécutées sur l'extension matérielle.

La figure 3.16.A montre un exemple de trace d'exécution d'un graphe d'application pour un NIOSII couplé à un bloc reconfigurable. Une instruction $SUB(d, e, f)$ y est exécutée en parallèle d'une occurrence m qui nécessite trois opérandes. La figure 3.16.B illustre que le premier cycle correspond à l'envoi du premier des trois opérandes de l'occurrence puisqu'il n'est possible de transmettre que deux opérandes simultanément. De même, l'occurrence produit trois résultats et deux cycles supplémentaires seront nécessaires, à la fin de son exécution, pour les récupérer sur le processeur.

L'exécution d'une occurrence est modélisée par des rectangles positionnés dans un espace bidimensionnel dont l'abscisse représente le temps, et dont l'ordonnée représente la ressource utilisée (cf. figure 3.16.C, les axes sont représentés avec une rotation de 90° afin de clarifier la figure). L'axe des ressources ne dispose que de deux valeurs possibles : si $y = 0$ la ressource utilisée est le processeur, si $y = 1$ il s'agit du seul bloc reconfigurable de l'extension. Les coordonnées et dimensions de chaque rectangle sont notées par la syntaxe suivante :

$$Rect = [x, y, \Delta x, \Delta y]$$

avec (x,y) les coordonnées de l'origine du rectangle, Δx sa largeur et Δy sa hauteur.

Les occurrences de taille unitaire, exécutées sur le processeur, sont associées à des rectangles qui modélisent l'occupation du processeur si elles sont sélectionnées. Par exemple, dans la figure 3.16.C, le rectangle $Rect_{SUB}$ occupe le processeur durant un cycle. Les occurrences contenant plusieurs nœuds seront exécutées sur l'extension (rectangle $Rect_m$) et occuperont également le processeur durant la

phase conjointe de transmission et de lancement ainsi que durant celle de réception des données (rectangles $RectR_m$ et $RectW_m$).

La contrainte (3.19) exprime le partage du processeur et de l'extension, en assurant que les rectangles modélisant les tâches de chaque occurrence ne se chevaucheront pas. Pour tous les rectangles, si une occurrence m n'est pas sélectionnée, sa hauteur est nulle ($sel_m = 0$) et n'occupera donc aucune ressource.

Contrainte 4 (Partage des ressources) *À chaque occurrence $m_i^k \in M$ (occurrence i de taille k) exécutable sur l'extension ($k > 1$), on associe deux tâches qui réservent le processeur durant le lancement de son exécution ainsi que durant la transmission et la réception de données. Ces tâches sont modélisées sous forme de rectangles :*

- $RectR_{m_i^k}[start_{m_i^k}, 0, ERT_{m_i^k}+1, sel_{m_i^k}]$ *: transmission des données et lancement de l'exécution de l'occurrence m_i^k.*
- $RectW_{m_i^k}[swr_{m_i^k}, 0, (EWT_{m_i^k} + MultiCycle_{m_i^k}) \cdot IsWriteTransfer_{m_i^k}, sel_{m_i^k}]$ *: récupération des données produites par l'occurrence m_i^k.*

De plus, à chaque occurrence $m_i^k \in M$, on associe une tâche correspondant à la réservation de sa ressource d'exécution :

$$Rect_{m_i^k} \begin{cases} [start_{m_i^k}, 0, delay_{m_i^k}, sel_{m_i^k}] & pour \quad k = 1 \\ [start_{m_i^k}, 1, delay_{m_i^k}, sel_{m_i^k}] & pour \quad k > 1 \end{cases}$$

Les rectangles ne se chevauchent pas :

$$\texttt{Diff2}(ListRect) \tag{3.7}$$
$$avec\ ListRect = [Rect_{m_i^k} \,|\, m_i^k \in M] \ +\!+$$
$$[RectR_{m_i^k} \,|\, m_i^k \in M \wedge k > 1] \ +\!+$$
$$[RectW_{m_i^k} \,|\, m_i^k \in M \wedge k > 1]$$

où $+\!+$ note la concaténation de vecteurs.

Les relations entre les différentes variables utilisées dans la définition des rectangles de chaque occurrence $m \in M$ sont définies par les contraintes (3.8)-(3.18). Le coût de la reconfiguration du bloc entre chaque instruction spécialisée est négligeable en comparaison de la durée d'un cycle du processeur (il peut néanmoins être pris en compte simplement en modifiant la largeur des rectangles $Rect_m$ et $RectW_m$).

Les contraintes (3.8) et (3.9) comptent le nombre de données transmises ou reçues du processeur. Si un nœud exécuté sur l'extension est la source de plusieurs liens, on optimise alors le nombre de communications vers le processeur en ne transmettant la donnée produite qu'une seule fois. Pour cela, on utilise la variable $isOutToProcessor_n$ qui indique si un nœud n produit une donnée à transmettre au processeur, elle vaudra un si la contrainte (3.10) est respectée (i.e., si parmi les successeurs de n il en existe au moins un qui soit couvert par une occurrence m^1, exécutée sur le processeur). Ainsi, pour calculer le nombre de sorties d'une occurrence m, il suffit de référencer parmi chaque nœud $n \in m$, ceux qui produisent une donnée à transmettre au processeur ($isOutToProcessor_n = 1$).

Contrainte 5 (Nombre d'opérandes venant du processeur)

$$NbInFromProcessor_m = \sum_{n \in InNodes_m \wedge n \in match^1} sel_{match^1} \tag{3.8}$$

Contrainte 6 (Nombre de données transmises au processeur)

$$NbOutToProcessor_m = \sum_{n \in m} isOutToProcessor_n \tag{3.9}$$

$$avec\ isOutToProcessor_n \Leftrightarrow \sum_{n \in \{v|(n,v) \in E\} \wedge n \in match^1} sel_{match^1} > 0 \tag{3.10}$$

La variable $IsWriteTransfer_m$ indique si au moins un des nœuds $n \in m$ produit une donnée à communiquer au processeur (3.11). Elle est utilisée dans chaque rectangle $RectW_m$ pour ne pas occuper inutilement le processeur dans le cas où une occurrence ne transmet aucune donnée.

Contrainte 7 (Existence d'une production de donnée vers le processeur)

$$IsWriteTransfer_m \Leftrightarrow NbOutToProcessor_m > 0 \tag{3.11}$$

Les contraintes (3.12) et (3.14) définissent, pour chaque occurrence m, les durées des phases additionnelles de transmission (ERT_m) et de récupération (EWT_m) des données. De plus, si le nombre de données transmises ou reçues est nul ($NbInFromProcessor_m = 0$ ou $NbOutFromProcessor_m = 0$), leurs durées respectives de transmission ou de récupération seront également nulles (3.13 et 3.15). $NbInPerCycle$ et $NbOutPerCycle$ sont des constantes issues de l'architecture du processeur et notent respectivement le nombre de données que le processeur peut envoyer ou transmettre en un seul cycle.

Contrainte 8 (Cycles supplémentaires d'envoi des opérandes)

$$ERT_m = \left\lceil \frac{NbInFromProcessor_m - NbInPerCycle}{NbInPerCycle} \right\rceil \cdot Rt_m \tag{3.12}$$

$$avec\ Rt_m \Leftrightarrow NbInFromProcessor_m \neq 0 \tag{3.13}$$

Contrainte 9 (Cycles supplémentaires de récupération des résultats)

$$EWT_m = \left\lceil \frac{NbOutToProcessor_m - NbOutPerCycle}{NbOutPerCycle} \right\rceil \cdot Wt_m \tag{3.14}$$

$$avec\ Wt_m \Leftrightarrow NbOutToProcessor_m \neq 0 \tag{3.15}$$

Les contraintes (3.16) et (3.17) définissent la date (swr_m) à partir de laquelle les résultats sont récupérés par le processeur. Celle-ci doit être supérieure à la fin du calcul de l'occurrence et doit permettre de transmettre toutes les données produites avant que l'occurrence ne libère l'extension.

Contrainte 10 (Début de la récupération des résultats)

$$swr_m \geq start_m + ERT_m + processing_m - MultiCycle \tag{3.16}$$

$$swr_m + EWT_m + MultiCycle \leq start_m + delay_m \tag{3.17}$$

Enfin, la contrainte 3.18 définit la durée totale d'une occurrence ($delay_m$) comme étant la somme de sa durée d'exécution constante ($processing_m$) et des durées de pénalité issues des transferts de données.

Contrainte 11 (Durée totale d'occupation d'un bloc reconfigurable)

$$delay_m \geq ERT_m + processing_m + EWT_m \tag{3.18}$$

3.4.3 Résultats expérimentaux

Pour valider expérimentalement notre approche, nous avons étudié le temps d'exécution des calculs effectués dans plusieurs applications après couverture et ordonnancement des occurrences de motifs. La cible est un processeur NiosII dans sa déclinaison « rapide »et cadencé à 150Mhz sur un FPGA Stratix2 d'Altera. Le processeur est couplé à une extension matérielle contenant un unique bloc reconfigurable. De même que dans les expériences sur la technique de couverture sans contrainte de ressources (cf. sous-section 3.3.3), pour chaque application, on génère tous les motifs qui contiennent au maximum dix nœuds et ont, au plus, quatre entrées et deux sorties.

Les tableaux 3.3 et 3.4 montrent les résultats obtenus pour des applications sélectionnées dans des *benchmarks* liés aux thèmes applicatifs récurrents dans les systèmes embarqués : multimédia, télécommunications, sécurité, etc. *MediaBench* [113], *MiBench* [83] et *MCrypt* [130]. Dans ces tableaux, $|V|$, $|P|$ et $|M|$ correspondent respectivement au nombre de nœuds des graphes d'application, au nombre de motifs identifiés et au nombre de leurs occurrences respectives. Pour chaque application, on indique le nombre de graphes analysés pour avoir une idée de leurs tailles moyennes (e.g., l'application *JPEG IDCT* contient 171 nœuds répartis dans deux graphes indépendants, on note 171/2 dans la colonne $|V|$). $|Psel|$ et $|Msel|$ correspondent respectivement au nombre de motifs et au nombre d'occurrences sélectionnés par la couverture.

La colonne *Accélération* montre le gain théorique attendu par la sélection et l'ordonnancement des occurrences. Il est calculé en comparant la durée d'exécution totale du nouvel ordonnancement à celle qui n'utilise que le jeu d'instructions standard du processeur. De plus, on se place dans le cas d'une exécution optimiste pour le processeur en supposant que chaque opération qu'il exécute ne dure qu'un seul cycle (pour le NiosII, les opérations de multiplication et de décalage ne prennent qu'un seul cycle mais ont une latence supplémentaire de deux cycles et risquent donc geler le *pipeline* d'exécution en présence d'une dépendance de données). La durée d'exécution d'un graphe est donc égale au nombre d'opérations qui y sont effectuées. Les durées des occurrences sont, quant à elles, calculées à partir de leurs chemins critiques en utilisant les latences matérielles, pour la cible FPGA, de chaque opérateur qui y est impliqué.

Le tableau 3.3 montre les résultats obtenus en utilisant une méthode de recherche standard du solveur de contraintes. Il s'agit d'une exploration en profondeur (cf. chapitre 2) et l'ordre d'évaluation des variables influe significativement sur l'efficacité de l'algorithme. Déterminer un ordre performant

Application	|V|	|P|	|M|	|Psel|	|Msel|	Accélération	Temps de résolution (s)
MiBench BF encrypt	244/1	75	312	6	99	1.89	18,84
MiBench BF decrypt	244/1	75	312	6	98	1,87	13,23
MiBench fft	15/1	17	15	3	9	1,15	0,58
Mediabench JPEG IDCT	171/2	322	1403	17	40	2,55	3,94
Mediabench Sha transform	35/1	33	72	7	17	1,75	0,57
Mediabench MESA invert matrix	150/1	67	295	8	80	1,35	27,99
Mediabench arf coef	28/1	97	662	3	4	3,11	3,39
MCrypt cast128	279/2	129	366	279	279	pas de solution en 30s	
MCrypt gost	16/1	59	85	3	6	2,67	1,95

TABLE 3.3 – Accélération des calculs après sélection et ordonnancement des occurrences de motifs.

est difficile et dépend généralement du problème. Dans notre cas de couverture et de sélection, le fait de commencer par évaluer les variables $start_n$ et $match_n$ des nœuds situés sur le chemin critique du graphe, améliore l'efficacité de la recherche. Cependant, pour des graphes de taille importante (plusieurs centaines de nœuds), le solveur ne trouve parfois pas de solution en un temps raisonnable. C'est le cas, par exemple, de l'application *MCrypt Cast 128* où aucune solution n'est trouvée en trente secondes. Pour améliorer le passage à l'échelle de notre approche, nous proposons une heuristique de résolution qui, sans modifier le modèle de contraintes, partitionne le graphe en plusieurs sous-ensembles de nœuds avant de résoudre itérativement le problème.

Application	|V|	|P|	|M|	|Psel|	|Msel|	Accélération	Temps de résolution (s)
MiBench BF encrypt	244/1	75	312	8	98	2,12	12,31
MiBench BF decrypt	244/1	75	312	8	98	2,12	12,53
MiBench fft	15/1	17	15	3	9	1,15	0,07
Mediabench JPEG IDCT	171/2	322	1403	19	51	2,90	23,76
Mediabench Sha transform	35/1	33	72	6	17	1,75	0,27
Mediabench MESA invert matrix	150/1	67	295	7	61	1,80	2,97
Mediabench arf coef	28/1	97	662	5	8	3,10	8,77
MCrypt cast128	279/2	129	366	10	148	1,52	87,45
MCrypt gost	16/1	59	85	3	6	2,67	1,03

TABLE 3.4 – Accélération des calculs après sélection et ordonnancement des occurrences de motifs en utilisant l'heuristique de résolution.

Cette heuristique se base sur un ordonnancement ALAP [18] du graphe (sans contrainte de ressource). Chaque nœud est alors associé à une date d'exécution afin de partitionner le graphe en plusieurs fenêtres temporelles. Chaque fenêtre est alors composée d'un ensemble de nœuds dont la distance temporelle est inférieure à un paramètre de l'heuristique (choisi expérimentalement). L'intuition derrière ce partitionnement est que, dans la plupart des cas, deux nœuds qui se trouvent dans des fenêtres différentes ne pourront être couverts par la même occurrence ni partager la même ressource. Ainsi, pour chaque fenêtre, on résout le problème en cherchant à ne déterminer que les variables des nœuds se trouvant dans la fenêtre. Puis, avant de résoudre le problème pour la fenêtre suivante, on injecte les solutions partielles des fenêtres précédentes pour contraindre le reste du problème. Ce type d'heuristique montre qu'il est possible de bénéficier d'algorithmes de résolution spécifiques à un problème sans avoir à en modifier le modèle des contraintes.

18. As Late As Possible

Les résultats obtenus avec l'heuristique sont présentés dans le tableau 3.4, ils sont au moins aussi bons que pour le tableau 3.3. De plus, l'heuristique nous permet de trouver une solution à l'application *MCrypt Cast 128* qui accélère de 50% son exécution. Cependant, il est important de noter que cette heuristique est gloutonne et risque, si la taille des fenêtres est trop petite, d'aboutir à une absence de solution au vu des choix effectués pour les fenêtres précédentes.

Les mesures ont été réalisées en utilisant le solveur JaCoP [19] sur une machine disposant d'un processeur intel core2 duo à 2,8Ghz.

3.5 Processeur couplé à une extension parallèle

On s'intéresse maintenant à une extension qui dispose de plusieurs blocs reconfigurables interconnectés par réseau de communication *full-crossbar* (cf. figure 3.17). L'objectif est d'exploiter le parallélisme des opérations au sein d'une instance de la représentation intermédiaire (e.g., DAG) : si deux groupes de nœuds ne sont pas liés, directement ou indirectement, par une dépendance de données, il est possible d'exécuter les deux groupes d'opérations simultanément.

3.5.1 Architecture de l'extension

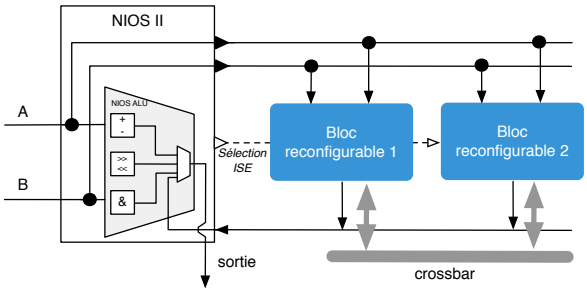


FIGURE 3.17 – Processeur NIOSII couplé à une extension parallèle contenant plusieurs blocs reconfigurables interconnectés par un *crossbar*.

La figure 3.18 montre le modèle d'architecture envisagé pour chaque bloc de l'extension. Les motifs sélectionnés sont mis en œuvre par un chemin de données spécifique, ses interconnexions sont configurées par un contrôleur qui décode l'identifiant de l'instruction spécialisée envoyé par le processeur. Chaque bloc est connecté aux autres par un *full-crossbar* et dispose d'un ensemble de registres et de deux mémoires locales double ports (l'une pour lire les données externes et l'autre pour les enregistrer). Le premier port est connecté à la mémoire principale du processeur, le second au chemin de données. La génération des adresses utilisées par les différents accès est laissée à la charge d'un générateur d'adresse.

Les contraintes d'ordonnancement associées à ce modèle d'architecture sont les suivantes :

- Plusieurs occurrences de motifs peuvent s'exécuter simultanément sur des blocs reconfigurables différents.
- Le processeur peut configurer simultanément plusieurs blocs (comportement VLIW [20]).

19. http://jacop.osolpro.com/
20. Very Long Instruction Word

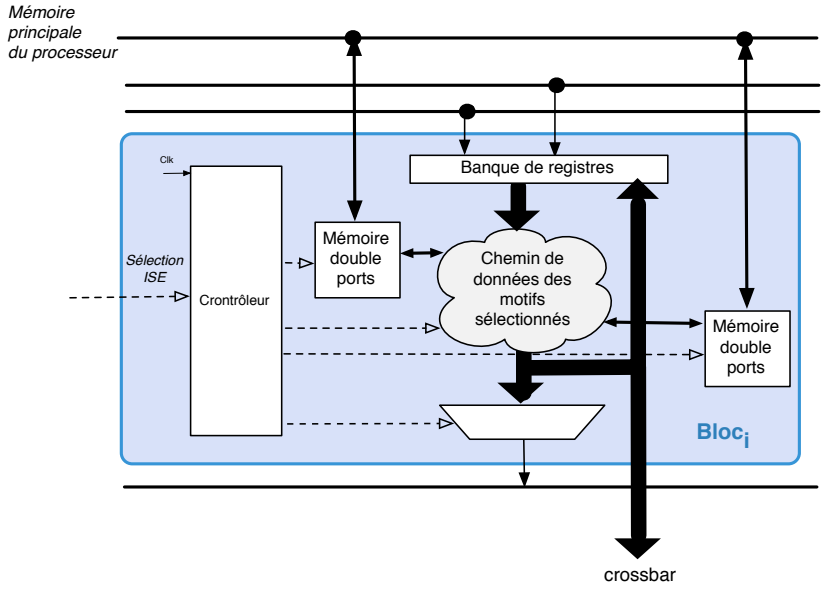


FIGURE 3.18 – Architecture d'un bloc reconfigurable pour le modèle d'extension parallèle.

- Si le processeur exécute une instruction primitive, il ne peut ni notifier une nouvelle configuration des blocs ni échanger des données.
- Les données produites sur un bloc reconfigurable sont directement accessibles à un autre bloc en utilisant le réseau *full-crossbar* (le délai de communication est négligé).
- Des mémoires embarquées dans chaque bloc reconfigurable sont connectées à la mémoire principale et peuvent être utilisées pour accéder aux données externes sans nécessiter un chargement/déchargement au travers du processeur.

3.5.2 Exemple de couverture et d'ordonnancement

Afin de clarifier le modèle d'exécution d'un graphe couvert pour cette architecture, on s'intéresse à l'ordonnancement de deux filtres FIR partiellement déroulés. La figure 3.19 expose les motifs identifiés (figure 3.19.A) ainsi que la couverture du graphe analysé qui contient deux composantes indépendantes (figure 3.19.B) .

L'ordonnancement de la couverture (cf. figure 3.19.C) illustre une exécution qui utilise deux blocs reconfigurables pour exécuter en parallèle l'ensemble des occurrences sélectionnées. Dans cet exemple, on considère que les entrées et sorties externes du graphe sont envoyées et récupérées par le processeur, les mémoires embarquées dans l'architecture ne sont donc pas utilisées. Cependant, les registres internes permettent de mémoriser les résultats intermédiaires qui sont produits sur l'extension et réutilisés par une autre occurrence. Les occurrences $M1, M2, M3, M6, M7$ et $M8$ ne nécessitent donc que deux entrées et les bus en entrée de l'extension suffisent à les fournir lors du lancement de leurs exécutions (L) : aucune instruction supplémentaire (R) n'est requise.

Au cycle 13, le processeur récupère le résultat (W) produit sur le premier bloc par $M4$ tout en lançant l'exécution de $M9$ sur le deuxième bloc et en lui transmettant le reste des opérandes nécessaires à son exécution.

FIGURE 3.19 – Couverture et ordonnancement du cœur de calcul de deux filtres FIR partiellement déroulés. A) Motifs identifiés, B) Couverture du graphe, C) Ordonnancement parallèle des occurrences. Les symboles R,L et W définissent respectivement les tâches de lecture, de lancement et d'écriture du processeur.

Dans cet exemple, la durée des occurrences n'est pas représentative d'une mise en œuvre matérielle réelle : ces durées sont volontairement longues pour faire apparaître une exécution concurrente des occurrences. Cette couverture n'apportera donc pas une accélération importante de l'application qui s'exécutera ici en 15 cycles au lieu de 22 pour une exécution séquentielle, en considérant que le processeur exécute une opération par cycle.

3.5.3 Modèle de contraintes

L'approche présentée dans la section précédente peut être légèrement modifiée pour tirer parti d'un éventuel parallélisme entre de multiples occurrences de motifs qui seront alors déportées sur des blocs reconfigurables différents.

3.5.3.1 Variables utilisées

En plus des variables qui déterminent les pénalités issues de l'alimentation en données et de la récupération des résultats produits par l'extension (cf. paragraphe 3.4.2.1), nous définissons une nouvelle variable pour chaque occurrence $m_i \in M$:

- $resource_{m_i} :: [0..NbCells]$: identifiant de la ressource utilisée pour exécuter l'occurrence. Si $resource_{m_i} = 0$ l'occurrence est exécutée sur le processeur, sinon elle est exécutée sur un des blocs reconfigurables. Où $nbCells$ indique le nombre de blocs reconfigurables disponibles.

De même, à chaque nœud $n \in N$, on ajoute une variable pour identifier la ressource utilisée pour l'exécution de n :

- $resource_n :: [0..NbCells]$: identifiant de la ressource utilisée pour exécuter l'occurrence qui couvre n.

3.5.3.2 Contraintes de partage ressources

La principale différence avec l'extension séquentielle (cf. section 3.4) réside dans le parallélisme issu de la multiplicité des blocs reconfigurables. Une instruction spécialisée s'apparente alors à une instruction VLIW qui configure simultanément le chemin de données de plusieurs blocs. La figure 3.20 montre le modèle d'exécution pour une architecture disposant de deux blocs reconfigurables où les occurrences m_1 et m_2 sont exécutées en parallèle sur les deux blocs.

Si le processeur exécute une opération issue de son jeu d'instructions non étendu (cas d'une occurrence de motif de taille unitaire), il ne pourra pas être utilisé pour transmettre/recevoir des opérandes vers/de l'extension. La contrainte (3.19) garantit le respect de cette propriété et est illustrée par la figure 3.20.A : l'ordonnée $y = 0$ correspond aux bus d'entrée de l'extension et l'ordonnée $y = 1$ au bus de sortie.

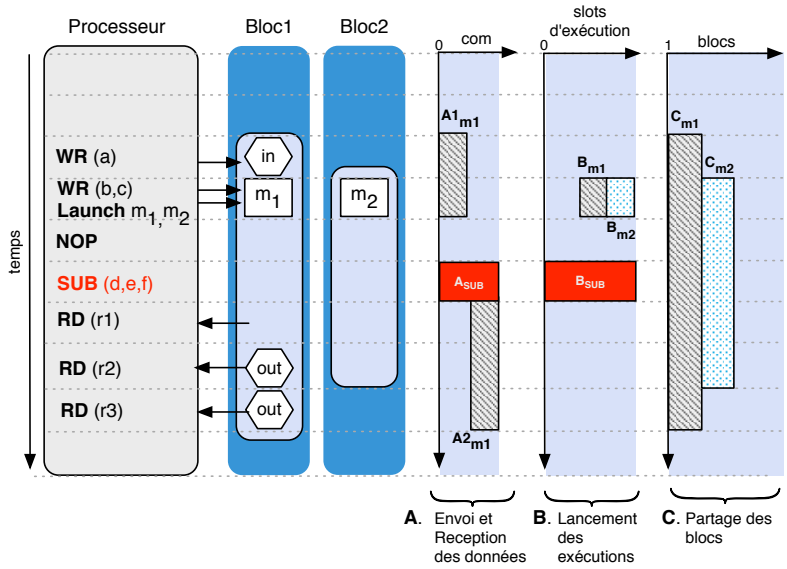


FIGURE 3.20 – Modèle d'exécution pour deux blocs reconfigurables.

Contrainte 12 (Envoi et réception des données par le processeur) *À chaque occurrence* $m_i^k \in M$ *(occurrence* i *de taille* k*) on associe des tâches correspondant aux transmissions/réceptions de données, elles sont modélisées sous forme de rectangles* $A1_{m_i^k}$ *(transmission),* $A2_{m_i^k}$ *(réception) :*

- $A1_{m_i^k}[start_{m_i^k}, 0, ERT + 1, sel_{m_i^k}]$ *si* m_i^k *contient plus d'un nœud* $(k > 1)$*.*
- $A2_{m_i^k}[swr_{m_i^k}, 1, EWT + IsMultiCycle_{m_i^k}, sel_{m_i^k}]$ *si* m_i^k *contient plus d'un nœud.*

De plus, à chaque occurrence $m_i^1 \in M$ *exécutable sur le processeur, on associe un rectangle* $A_{m_i^1}$ *qui bloque toute tâche de transmission ou de réception de données :*

- $A_{m_i^1}[start_{m_i^1}, 0, 2, sel_{m_i^1}]$.

Les rectangles de l'ensemble A *ne se chevauchent pas :*

$$\texttt{Diff2}(ListRectA) \tag{3.19}$$

$$\text{avec } ListRectA = [A1_{m_i^k} \,|\, m_i^k \in M \wedge k > 1] \;+\!+\; [A2_{m_i^k} \,|\, m_i^k \in M \wedge k > 1] \;+\!+\; [A_{m_i^k} \,|\, m_i^k \in M \wedge k = 1]$$

où $+\!+$ *note la concaténation de vecteurs.*

Il est évidemment impossible de configurer un ou plusieurs blocs si le processeur est déjà en train d'exécuter une opération. À chaque occurrence m_i^k, on associe donc un *slot* d'exécution $slot_i$ qui correspond à la date du lancement de son exécution dans le bloc reconfigurable qui lui est affecté. Le nombre de *slots* disponibles est égal au nombre de blocs disponibles dans l'extension et la variable représentant le choix de $slot_i$ correspond alors à la ressource $resource_{m_i^k}$. La contrainte (3.20) garantit que le processeur ne pourra lancer simultanément plus d'exécutions que de blocs disponibles .De plus l'exécution d'une instruction sur le processeur bloque tout autre lancement puisque la tâche liée à l'exécution de cette instruction occupe tous les *slots* disponibles (cf. figure 3.20.B).

Contrainte 13 (Partage des *slots* d'exécution) *À chaque occurrence $m_i^k \in M$ on associe un slot d'exécution (inférieur à nbCells, le nombre de blocs disponibles) modélisé sous la forme d'un rectangle $B_{m_i^k}$:*

$$B_{m_i^k} \begin{cases} [start_{m_i^k} + ERT, resource_{m_i^k}, 1, sel_{m_i^k}] & pour \quad k > 1 \\ [start_{m_i^k}, 0, nbCells, sel_{m_i^k}] & pour \quad k = 1 \end{cases}$$

Les rectangles de l'ensemble B ne se chevauchent pas :

$$\texttt{Diff2}(ListRectB) \quad avec\ ListRectB = [B_{m_i^k} \,|\, m_i^k \in M] \tag{3.20}$$

Il n'est pas possible d'exécuter plus d'une occurrence sur le même bloc (cf. figure 3.20.C). La contrainte (3.21) répartit les occurrences sélectionnées sur les différents blocs reconfigurables disponibles dans l'extension.

Contrainte 14 (Partage des blocs reconfigurables) *À chaque occurrence $m_i^k \in M$ exécutable sur l'extension ($k > 1$), on modélise son occupation d'un bloc par un rectangle $C_{m_i^k}$:*

- *$C_{m_i^k}[start_{m_i^k}, resource_{m_i^k}, 1, sel_{m_i^k}]$ si m_i^k contient plus de un nœud ($k > 1$).*

Les rectangles de l'ensemble C ne se chevauchent pas :

$$\texttt{Diff2}(ListRectC) \quad avec\ ListRectC = [C_{m_i^k} \,|\, m_i^k \in M \wedge k > 1] \tag{3.21}$$

De la même manière que pour le début et la durée d'exécution d'un nœud, la contrainte (3.22) exprime le fait que la ressource utilisée pour un nœud est celle de l'occurrence qui le couvre.

Contrainte 15 (Allocation de ressource pour un nœud) *Pour chaque nœud $n \in G$, la ressource utilisée pour n est celle de l'occurrence identifiée par $match_n$:*

$$\forall_{n \in N}\ Element(match_n, List_{resources}, resource_n), \quad avec\ List_{resources} = [resource_m \,|\, m \in M] \tag{3.22}$$

3.5.3.3 Stratégies d'optimisation

Indépendamment de la fonction d'optimisation qui minimise la durée d'exécution totale du graphe couvert, on définit une nouvelle stratégie d'optimisation qui minimise le nombre de ressources utilisées pour couvrir et ordonnancer un graphe G :

$$numberOfResources(G) = \texttt{Count}([resources_n | \forall n \in N]) \tag{3.23}$$

Afin de conserver toutefois des performances intéressantes, il est alors préférable de contraindre la durée totale d'exécution à être inférieure à $Dmax$ une durée maximale autorisée :

$$\texttt{Max}([start_n + delay_n | \forall n \in N]) < Dmax \tag{3.24}$$

Les annexes A.2 et A.4.3 détaillent la modélisation ARCAdE (cf. chapitre 8) du problème et de cette nouvelle stratégie de résolution.

3.5.4 Résultats expérimentaux

Pour valider expérimentalement notre approche, nous avons étudié le temps d'exécution des calculs effectués dans plusieurs applications. La cible est un processeur NiosII (cadencé à 150Mhz sur un FPGA Stratix2 d'Altera) couplé à une extension qui contient jusqu'à huit blocs reconfigurables. De la même manière que pour l'extension séquentielle (cf. sous-section 3.4.3), les motifs (4 entrées, 2 sorties et 10 nœuds au maximum) sont générés pour chaque application et leurs durées sont calculées à partir de leurs chemins critiques.

Les mesures ont été réalisées en utilisant le solveur JaCoP [21] sur une machine disposant d'un processeur intel core2 duo à 2,8Ghz.

Deux modèles ont été évalués : le premier (modèle A) considère que toutes les données externes sont présentes ou écrites dans les mémoires embarquées dans l'extension, le second (modèle B) utilise le processeur pour y accéder ou les transmettre. Pour passer du modèle B au modèle A, il suffit de ne pas tenir compte des entrées et sorties externes du graphe analysé et les contraintes ne changent donc pas d'un modèle à l'autre.

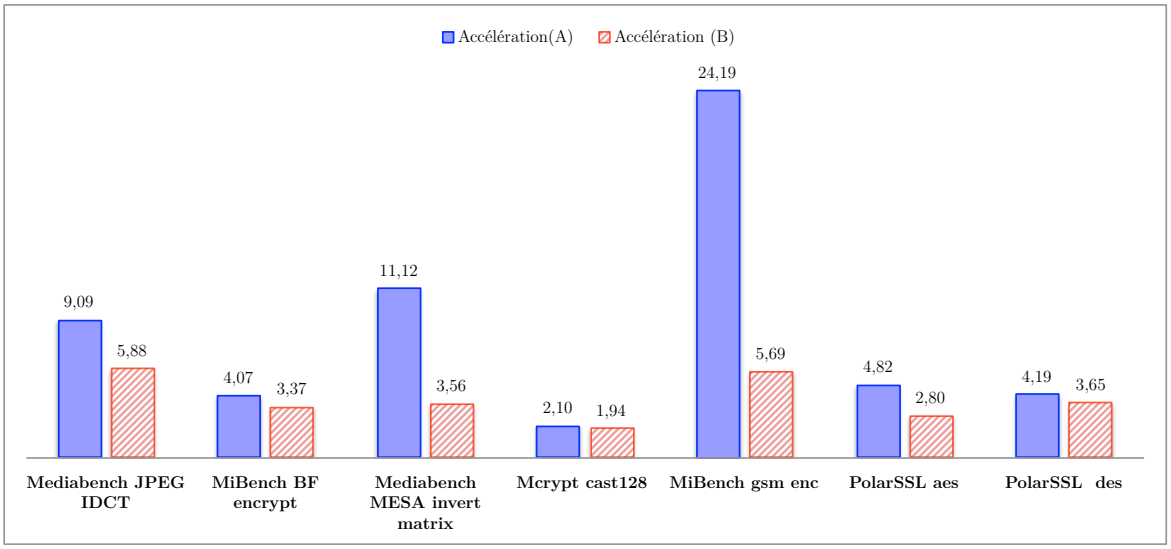


FIGURE 3.21 – Comparaison des accélérations avec (modèle A) et sans mémoires embarquées pour les données externes (modèle B).

Le graphique de la figure 3.21 compare les accélérations théoriques obtenues pour différentes applications dont les caractéristiques sont résumées dans le tableau 3.2, page 62. L'accélération A

21. \http://jacop.osolpro.com/

évalue les performances du modèle A par rapport à une exécution séquentielle sur le processeur : chaque donnée externe est lue ou écrite dans une mémoire embarquée sur l'extension. L'accélération B montre, quant à elle, les résultats obtenus si le processeur est utilisé pour lire ou écrire ces données externes.

Il apparaît clairement que le parallélisme de l'extension offre une accélération beaucoup plus importante pour le modèle A que pour le modèle B (e.g., cas de l'encodeur gsm de *MiBench* où l'accélération est quatre fois plus importante pour le modèle A). Les graphes où l'accélération est particulièrement importante correspondent à de larges graphes (le parallélisme des opérations est donc important) dont le chemin critique est faible et les occurrences de motifs peuvent alors s'exécuter en parallèle sur les multiples blocs reconfigurables. Pour certaines applications, les accès mémoires ne constituent pas le goulot d'étranglement et les différences de performances entre le modèle A et le modèle B sont mineures. C'est le cas des applications de cryptologie (e.g., *MiBench blowfish encrypt*, *MiBench blowfish encrypt* et *PolarSSL des*) qui ont en commun de longues séquences d'opérations exposant un faible niveau de parallélisme potentiel.

Les résultats détaillés des expérimentations sont présentés dans le tableau 3.5. Pour chaque modèle, on mesure le nombre de motifs différents sélectionnés par la couverture, le nombre de blocs reconfigurables utilisés, l'accélération obtenue ainsi que le temps nécessaire au solveur pour déterminer la solution la plus intéressante. Dans tous ces exemples, la taille des graphes est relativement importante (environ 150 nœuds) et le solveur n'arrive pas à prouver l'optimalité des solutions identifiées dans le temps imparti (30s). Dans le cas du modèle A, le nombre de données externes (entrées et sorties) du graphe ainsi que le nombre maximum d'accès qui y sont faits simultanément (à chaque cycle) sur l'extension sont comptabilisés. Pour les applications *Mediabench MESA invert matrix* et *PolarSSL aes*, le nombre d'accès parallèles aux données externes est particulièrement important et nécessitera de coûteuses ressources de mémorisation pour les satisfaire. Afin d'évaluer l'impact d'une limitation des ressources mémoires disponibles, nous avons contraint le nombre maximum d'accès parallèles aux données externes à être inférieur à 10 pour l'application *Mediabench MESA invert matrix*. L'accélération alors obtenue est légèrement réduite (10 au lieu de 11).

	Ordonnancement parallèle (modèle A)						**Ordonnancement parallèle (modèle B)**			
Applications	Motifs sélectionnés	Nombre d'accès mémoires parallèles (max)	Données externes	Blocs utilisés	Accélération A_1	Temps de résolution (s)	Motifs sélectionnés	Blocs utilisés	Accélération A_2	Temps de résolution (s)
Mediabench JPEG IDCT	28	13	78	8	9,09	14,2	28	5	5,88	10,2
MiBench BF encrypt	7	3	152	4	4,07	13,2	7	4	3,37	0,01
Mediabench MESA invert matrix	9	38	134	8	11,12	0,6	9	4	3,56	0,5
Mcrypt cast128	18	8	155	6	2,1	6,6	18	3	1,94	15,9
MiBench gsm enc	9	15	132	8	24,19	6,0	9	2	5,69	7,2
PolarSSL aes	15	27	739	8	4,82	20,2	10	5	2,8	43,6
PolarSSL des	26	6	156	5	4,19	23,9	25	4	3,65	24,8

TABLE 3.5 – Résultats de la couverture et de l'ordonnancement pour une extension comportant jusqu'à huit blocs reconfigurables parallèles.

Ces résultats illustrent l'importance de la gestion mémoire dans la conception d'une extension matérielle parallèle : dans le cas du modèle B, la plupart des applications n'utilisent que la moitié (ou moins) des blocs reconfigurables disponibles. Le processeur est en effet principalement occupé à transférer ou à récupérer les données et le nombre de bus pour les opérandes n'est pas suffisant pour exploiter tout le parallélisme potentiel de l'application.

Il est important de noter que le coût en surface de l'extension matérielle à synthétiser risque d'être élevé : 1) le nombre de motifs sélectionnés est relativement important 2) les opérateurs matériels correspondants peuvent être dupliqués sur plusieurs blocs reconfigurables. De plus, pour chaque bloc reconfigurable, les configurations des chemins de données seront différentes et il n'est pas garanti que les 256 identifiants d'instructions spécialisées disponibles pour le NiosII soient suffisants. De futurs travaux, portant sur la synthèse matérielle des différents blocs reconfigurables, permettront d'étudier plus précisément leur faisabilité et feront certainement apparaître la nécessité de compromis. Ceux-ci pourront être intégrés à l'algorithme en ajoutant des contraintes supplémentaires au problème d'optimisation.

3.6 Conclusion

Dans ce chapitre, nous avons présenté une technique conjointe de couverture et d'ordonnancement d'un graphe par une bibliothèque de motifs. La caractérisation du problème sous forme d'un CSP apporte une modularité qui nous permet de le raffiner en un problème de sélection et d'ordonnancement d'instructions spécialisées.

Ces problèmes sont résolus par un solveur de contraintes et les expérimentations ont montré qu'il était possible de traiter des graphes de quelques centaines de nœuds en un temps raisonnable (moins de 30s pour chaque graphe). En fonction des caractéristiques du graphe et de la bibliothèque de motifs utilisée, il est parfois nécessaire d'utiliser une heuristique gloutonne pour réussir à résoudre le problème. Celle-ci n'influe cependant que sur la résolution du problème et est complètement dissociée du modèle de contraintes.

Les résultats obtenus dans le cas d'une extension parallèle (instructions spécialisées VLIW) font notamment apparaître d'importantes accélérations des temps de calcul (jusqu'à 24 fois plus rapide que le temps de calcul nécessaire au processeur) pour des applications exposant suffisamment de parallélismes. Ce résultat est cependant à nuancer par le fait que la gestion des mémoires nécessaires à une architecture supportant un tel niveau de parallélisme risque d'être trop complexe (les données devront être présentes avant l'exécution d'une occurrence qui en a besoin) et coûteuse en termes de surface matérielle. Or, il a également été observé que l'absence de ces mémoires pénalise fortement les performances de l'architecture (e.g., l'accélération x24 est alors réduite à x5) : la pression issue des transferts de données est alors trop forte sur le processeur pour qu'il puisse exploiter pleinement le parallélisme des blocs reconfigurables de l'extension.

La problématique du dimensionnement des mémoires pour les données externes d'un graphe est, au moins en partie, issue du fait que l'on ne dispose d'aucune information sur la source concrète d'un accès tableau. Or, bien souvent, un graphe d'application est extrait à partir d'un corps de boucles et une analyse fine des dépendances de données entre les différentes itérations de ce même graphe, nous permettrait d'établir qu'une donnée externe lue est en fait produite, lors d'une itération précédente,

sur l'extension. Dans ce cas, une mémoire, ou parfois même un simple registre, suffit à temporiser les données sans nécessiter une étape de remplissage des mémoires en amont de l'exécution de chaque graphe ordonnancé.

Ces aspects d'analyse du contexte inter-itération des corps de boucles dans une optique d'extension de jeu d'instructions font l'objet du chapitre 6, ce dernier repose sur le *modèle polyédrique* présenté dans le prochain chapitre.

Deuxième partie

Sélection d'instructions spécialisées et optimisation de code

Chapitre 4

Optimisation de code dans le modèle polyédrique

Le *modèle polyédrique* offre une abstraction mathématique puissante qui permet notamment d'exprimer des combinaisons complexes de transformations de boucles sous la forme de simples fonctions affines. L'objectif de ce chapitre est d'introduire les concepts et notations de ce modèle tout en tentant de résumer les nombreux travaux des vingt dernières années dans ce domaine. Ces bases seront utiles à la compréhension de notre approche conjointe de transformation de code et d'extension de jeu d'instructions qui fait l'objet du chapitre 6.

Sommaire

4.1 Introduction

Dans un compilateur, la représentation intermédiaire d'un programme conditionne la découverte et la qualité des optimisations. La majorité des représentations utilisées sont proches de la structure impérative des programmes. Pour obtenir de meilleures performances, le compilateur compose des séquences d'optimisations issues d'un ensemble existant de transformations. Le choix et l'ordre de ces transformations constituent des problèmes complexes souvent assistés par un expert de l'architecture ou de l'application cible.

On distingue principalement deux familles d'optimisation : celles qui travaillent sur la sémantique des instructions et celles qui modifient la structure de contrôle du code. Les parties critiques d'une application se trouvant la plupart du temps dans des nids de boucles, il est naturel de chercher à les transformer pour en améliorer les performances pour une architecture matérielle donnée. Les gains obtenus se situent au niveau de la localité des données et du parallélisme. Par exemple, la transformation d'un nid de boucles peut permettre de limiter les aléas d'accès au cache si sa structure réduit la durée de vie utile des données en mémoire. De même, des données contiguës en mémoire favorisent l'efficacité du cache si celles-ci sont utilisées à des instants proches.

Au cours des deux dernières décennies, le modèle polyédrique a été appliqué avec succès à de nombreux problèmes d'optimisation de boucles. Il offre une connaissance exacte des dépendances de données et exprime, dans une forme géométrique compacte, une infinité de compositions de transformations usuelles.

Ce chapitre a pour objectif de synthétiser les principales techniques d'optimisation de nids de boucles basées sur le modèle polyédrique. Nous présenterons tout d'abord l'aspect formel et le domaine d'application du modèle dans la section 4.2. Le modèle polyédrique permet de transformer le code source d'une application en une version plus efficace via des transformations affines. Ces techniques d'optimisation seront détaillées dans la section 4.3 qui fera notamment apparaître leur puissante expressivité.

4.2 Le modèle polyédrique

Le modèle polyédrique est une abstraction mathématique qui ne représente plus un nid de boucles sous une forme impérative. Selon ce modèle, les instructions sont définies dans des espaces multidimensionnels ; ceux-ci représentent les conditions d'existence de chaque instructions en fonction des indices de ses boucles englobantes.

Afin d'exprimer l'intuition de cette abstraction, la figure 4.1b illustre le domaine d'itération d'une instruction S contenue dans deux boucles imbriquées. Ce domaine est défini par un espace bidimensionnel où chaque dimension représente un indice du nid boucle entourant S. Dans cet exemple, les bornes hautes des boucles i et j sont respectivement inférieures à des paramètres N et M qui ne seront connus qu'à l'exécution du programme. La figure 4.1b correspond donc à un cas particulier où N et M sont égaux à 6. Dans ce cas, chaque itération du nid de boucle est identifiée par un unique point de coordonnées (i, j) tel que $6 \geq i, j \geq 1$. L'instruction S disposera donc d'autant d'instances que de points dans son domaine d'itération ; les dépendances de données de S porteront alors sur une instance de S à l'itération produisant une donnée et sur les instances de S qui y accéderont.

Dans cette section, nous introduisons les principales définitions et notations du modèle polyédrique,

(a) Exemple de nid de boucles.

```
1 for(i=1;i<N+1;i++){
2   for(j=1;j<M+1;j++)
3 S: A[i][j] = A[j][i] + A[i][j-1];
4 }
```

(b) Domaine d'itération de S pour $N = M = 6$.

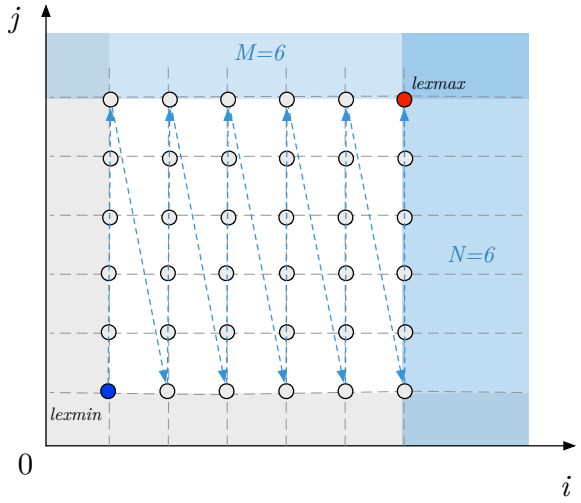


FIGURE 4.1 – Domaine d'itération d'un nid de boucles. Chaque point du domaine représente une itération du nid de boucles.

nous détaillons également, plus formellement que dans le paragraphe précédent, son application à l'analyse des dépendances de données d'un programme.

4.2.1 Notations

Le modèle polyédrique s'appuie sur l'algèbre linéaire pour représenter de manière compacte des nids de boucles. Quelques définitions sont nécessaires à la compréhension du formalisme mathématique utilisé. Pour une description plus approfondie, le lecteur pourra se référer aux travaux de Schrijver [167].

Définition 8 (Fonction affine) *Une fonction $f : \mathbb{Z}^m \to \mathbb{Z}^n$ est affine ssi il existe une matrice $A \in \mathbb{Z}^{m*n}$ et un vecteur $\vec{b} \in \mathbb{Z}^n$ tels que :*

$$\forall \vec{x} \in \mathbb{Z}^m, f(\vec{x}) = A\vec{x} + \vec{b}$$

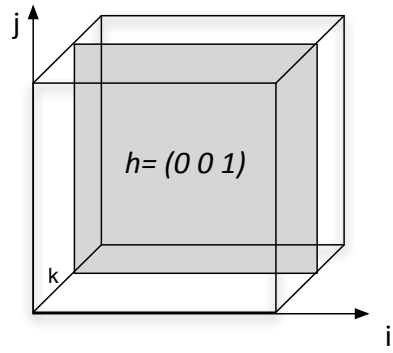
Définition 9 (Hyperplan affine) *Un hyperplan affine de dimension $m-1$ est un sous-espace d'un espace de dimension m. Il est défini par une fonction affine $\phi(\vec{v}) : \mathbb{Z}^m \to \mathbb{Z}$ telle que :*

$$\phi(\vec{v}) = h.\vec{v} + c$$

Avec h le vecteur ligne normal à l'hyperplan et $c \in \mathbb{Z}$.

Un hyperplan affine $h.\vec{v} = k$ divise un espace affine en un demi-espace positif ($h.\vec{v} \geq k$) et un demi-espace négatif ($h.\vec{v} \leq k$). Chacun de ces demi-espaces peut être représenté par une inégalité affine. La figure 4.2 illustre les notions de la définition 9 et de demi-espace. Le schéma 4.2b montre ainsi un espace mono-dimensionnel k' où ont été projetés l'ensemble des instances de l'hyperplan du schéma 4.2a.

(a) Instance de l'hyperplan (0 0 1) dans l'espace (i j k)



(b) Transformation de l'espace par l'hyperplan (projection)

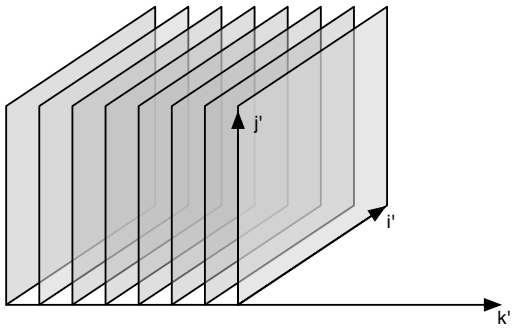


FIGURE 4.2 – Hyperplan dans un espace à trois dimensions.

Définition 10 (Polyèdre) *Un polyèdre est l'intersection d'un ensemble fini de demi-espaces. Chaque demi-espace correspondant à une face du polyèdre, il peut être décrit par un ensemble d'inégalités :*

$$P = \{\vec{x} \in \mathbb{Z}^m | A\vec{x} + \vec{b} \geq \vec{0}\}$$

Définition 11 (Polyèdre paramétrique) *Soit $\vec{n}$ un vecteur de paramètres. Un polyèdre paramétrique est défini par :*

$$P = \{\vec{x} \in \mathbb{Z}^m | A\vec{x} + B\vec{n} + \vec{b} \geq \vec{0}\}$$

Soit $\vec{x} = (x_1, \ldots, x_p)$ le vecteur des i indices et $\vec{n} = (n_1, \ldots, n_q)$ le vecteur des j paramètres. Les k faces d'un polyèdre paramétrique P peuvent être exprimées sous forme matricielle :

$$\mathrm{P} = \begin{array}{c} \\ l_1 \\ \vdots \\ l_k \end{array} \begin{array}{c} \begin{array}{ccccccc} x_1 & \ldots & x_p & n_1 & \ldots & n_j & c \end{array} \\ \begin{pmatrix} a_{1,1} & \ldots & a_{1,p} & b_{1,1} & \ldots & b_{1,q} & c_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k,1} & \ldots & a_{k,p} & b_{k,1} & \ldots & b_{k,q} & c_k \end{pmatrix} \end{array}$$

Chaque ligne l_i correspond à la contrainte suivante :

$$a_{i,1}.x_1 + \cdots + a_{i,p}.x_p + b_{i,1}.n_1 + \cdots + b_{i,q}.n_q + c_i \geq 0$$

Un polyèdre possède également une représentation duale définissant chaque point d'un polyèdre comme étant la somme de combinaisons de sommets, de lignes et de rayons. L'algorithme de Chernikova [120] permet de passer de la représentation donnée dans la définition 10 à celle-ci dite des sommets. Polylib [145] et PPL [151] sont deux bibliothèques polyédriques permettant de manipuler ces deux représentations dans $\mathbb{R}$. ISL [96] est une bibliothèque qui n'utilise pas la représentation des sommets mais qui est uniquement dans $\mathbb{Z}$, la syntaxe est proche de celle d'Omega [105].

Un polyèdre est convexe par définition. Cependant, il est souvent utile de manipuler des formes géométriques non convexes. Celles-ci sont composées d'une union de polyèdres (définition 12) et disposent des mêmes opérateurs que des polyèdres.

Définition 12 (Domaine polyédrique) *Soit $\vec{n}$ un vecteur de paramètres, un domaine polyédrique D de dimension m est une union de n polyèdres défini par :*

$$D = \{\vec{x} \in \mathbb{Z}^m | \bigcup_{k=1}^{n} A_k\vec{x} + B_k\vec{n} + \vec{b_k} \geq \vec{0}\}$$

4.2.2 Parties de code à contrôle statique (SCoP)

La majorité des travaux basés sur le modèle polyédrique sont applicables à une sous-classe de programmes dite de SCoP [1] [54, 200] (également appelée ACL [2]). Initialement défini pour un programme complet, le concept a été étendu à des parties d'un programme. Une SCoP possède les caractéristiques suivantes :

- les instructions de contrôle ne peuvent être que des boucles *for* ou des instructions conditionnelles (pas de boucles *while*).
- les bornes des boucles et les conditions sont des fonction affines des indices de boucles et des paramètres du programme.
- les fonctions d'accès tableaux sont également des fonctions affines des indices de boucles et des paramètres du programme.
- l'incrément des boucles est unitaire.

Extraction de SCoP Si un programme n'est pas une SCoP dans son intégralité, il est souvent possible d'en extraire des sous-parties qui le sont. Les indices de boucles extérieures aux SCoP sont des invariants et vus comme des paramètres. La couverture d'un programme par des SCoP peut être améliorée par des transformations classiques comme la propagation de constantes ou la transformation de boucles *while* possédant une variable d'induction en boucles *for*. D'autre part, une passe de normalisation de boucle permet toujours de traiter les pas non unitaires et négatifs. Les travaux de Girbal [70, 71] ont montré l'importance et la fréquence élevée de ces parties dans les programmes de calcul scientifique.

Domaine d'itération Un *domaine d'itération* est un polyèdre définit dans un espace où chaque dimension correspond à un indice de boucle. Un point de ce domaine est un *vecteur d'itération* positionnant de manière unique les instances dynamiques des instructions de calcul englobées.

Par exemple, dans le code de la figure 4.1a chaque instance de l'instruction S est identifiée par un vecteur d'itération $\vec{x}_S = \langle i, j \rangle$. Une instance de S est notée $\langle S, \vec{x}_S \rangle$ avec $\vec{x}_S$ dans le domaine :

$$D_S : \{i, j | N \geq i \geq 1, M \geq j \geq 1\}$$

La figure 4.1b représente D_S dans un espace bidimensionnel. Les contraintes de bornes des indices définissent la forme et la taille du polyèdre, mais celui-ci ne suffit pas à modéliser un nid de boucles. En effet, dans le programme original, les instances d'une instruction sont évaluées séquentiellement. La modification de cet ordre peut entraîner le non-respect des dépendances de données.

1. Static Control Part
2. Affine Control Loop

Ordre lexicographique Un domaine d'itération est associé à un parcours explicite appelé *ordre lexicographique* (cf. flèches en pointillés sur la figure 4.1b). Le premier et dernier point évalué dans l'ordre lexicographique correspondent respectivement au minimum (*lexmin*) et au maximum lexicographique (*lexmax*) du polyèdre. Soit deux vecteurs d'itération $\vec{x}_1$ et $\vec{x}_2$ tels que le premier soit lexicographiquement inférieur au second, on note :

$$\vec{x}_1 \prec \vec{x}_2$$

Relâchement du modèle polyédrique Des travaux étendent le modèle polyédrique aux programmes dits irréguliers. Barthou [16] et Benabderrahmane [20] utilisent une approximation conservatrice surestimant les dépendances de données. Cette politique permet d'élargir la couverture du modèle polyédrique à des programmes dont le flot de contrôle dépend des données tout en empêchant des transformations risquant de modifier la sémantique. L'analyse décrite par Größlinger [78] s'intéresse aux bornes, conditions et fonctions d'accès qui ne sont pas des fonctions affines.

4.2.3 Représentation des dépendances de données

Le modèle polyédrique offre une analyse exacte des dépendances de données. À l'inverse des analyses classiques, une dépendance n'implique pas deux instructions, mais deux instances dynamiques d'instructions. Intuitivement, deux instances d'instructions seront dépendantes si elles accèdent au même élément d'un tableau. Les fonctions d'accès à cet élément étant affines, il est possible de déterminer de manière exacte les conditions et les positions relatives des instances dépendantes.

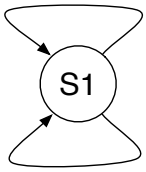
Dépendance dynamique Une dépendance dynamique $e_{S_1 \to S_2}$ exprime une dépendance de donnée entre une instance de S_1 et une instance de S_2. Elle est associée à un *polyèdre de dépendance* P_e contenant les conditions d'existence de la dépendance et les équations positionnant l'instance productrice par rapport à l'instance consommatrice d'une donnée. Le nombre de dimensions de P_e est donné par la relation suivante :

$$dim(P_e) = dim(\vec{x}_{S_1}) + dim(\vec{x}_{S_2}) + dim(\vec{n}) + 1$$

Graphe de dépendances généralisées L'ensemble des instructions d'une SCoP et des dépendances dynamiques peuvent être représentées dans un graphe $G(N, E)$ appelé graphe de dépendances généralisées ou PRDG [3]. Chaque nœud $n \in N$ correspond à une instruction et chaque lien indique une dépendance dynamique entre deux instructions. Dans le cas d'une dépendance entre deux instances d'une même instruction, la source et la destination du lien sont donc identiques. Le graphe de dépendances (cf. figure 4.3) associé à l'exemple précédent fait ainsi apparaître deux dépendances dynamiques qui ont pour source et destination des instances de la même instruction. Le polyèdre P_{e_1} donne les conditions d'existence de la dépendance et indique que pour une instance destination $\vec{x}_{dest} = \langle i, j \rangle$, l'instance source est $\vec{x}_{source} = \langle i, j-1 \rangle$.

3. Polyhedral Reduced Dependency Graph

$$P_{e_1} = \{i' = i, j' = j - 1, N \geq i \geq 1, M \geq j \geq 2\ \}$$



$$P_{e_2} = \{i' = j, j' = i, i - j \geq 1, N \geq i \geq 1, M \geq j \geq 2\ \}$$

FIGURE 4.3 – Graphe de dépendances généralisées du code de la figure 4.1a.

Analyse exacte des dépendances dynamiques Des algorithmes permettent d'obtenir une connaissance exacte des dépendances de données d'une SCoP. Le plus répandu est celui de Feautrier [54] qui énonce un problème de programmation linéaire paramétrique [53] pour déterminer les instances sources les plus récentes de chaque référence. Le résultat est une forme arborescente appelée *quast*[4]. Un nœud est une condition affine ou la négation d'une condition et une feuille correspond à la position relative d'une source par rapport à une référence. Le *quast* fournit donc toutes les informations nécessaires à la construction du graphe de dépendances généralisées. La résolution du test Omega [153] est quant à elle basée sur une extension de l'élimination de variables de Fourrier-Motzkin adaptée à la programmation entière. Il est cependant moins précis que l'algorithme de Feautrier dans le cas où des paramètres sont impliqués dans les fonctions affines.

4.3 Formalisme et expressivité des transformations dans le modèle polyédrique

Chaque instance dynamique d'une instruction possède une date d'exécution définie par l'ordre lexicographique (cf. paragraphe 4.2.2). Une transformation dans le modèle polyédrique modifie les dates d'exécution des instances d'instructions et leurs domaines d'itérations respectifs.

4.3.1 Transformation affine

Deux instances partageant la même date peuvent être exécutées en parallèle. Dans une SCoP, il toujours possible de représenter l'ordre lexicographique des instructions par des fonctions affines d'ordonnancement qui associeront à chaque instance d'instruction une date d'exécution en fonction des indices de boucles et des paramètres.

Définition 13 (Ordonnancement affine) *Soit une instruction S, l'ordonnancement affine Θ_S de dimension p est une fonction affine des indices de boucles $\vec{x}_S$ pour chaque dimension et des paramètres $\vec{n}$. Elle est définie par :*

$$\Theta_S(\vec{x}_S) = T_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}, T_S \in \mathbb{Z}^{p \cdot dim(\vec{x}_S) + dim(\vec{n}) + 1}$$

4. quasi-affine selection tree

Par exemple, une fonction d'ordonnancement pour une instruction S définie dans un domaine D_S bidimensionnel et ne disposant que d'un unique paramètre, respectera toujours le *prototype d'ordonnancement* suivant :

$$\Theta_S(\vec{x}_S) = t_{S,0} + t_{S,1} \cdot x_1 + t_{S,2} \cdot x_2 + t_{S,3} \cdot n_1 \tag{4.1}$$

Une fonction d'ordonnancement associe une date d'exécution à un point d'un domaine d'itération. Si $p = 1$, Θ_S est un hyperplan partitionnant l'espace le long de la normale $\vec{x}_S$. La date d'exécution est alors une fonction affine déterminant une date scalaire, on parle d*'ordonnancement monodimensionnel*. Si $p > 1$, Θ_S est un ensemble d'hyperplans et la date est un vecteur. L'ordonnancement est alors *multidimensionnel*. Un exemple intuitif est la position d'un moment d'une journée décrite par un vecteur dont les dimensions sont heures, minutes et secondes. Le résultat d'un ordonnancement multidimensionnel est exprimable par un nid de boucles dont la profondeur correspond à la dimension du vecteur de date.

Légalité d'un ordonnancement Une dépendance de donnée $e_{S \to R}$ implique que la date d'exécution de l'instance source S doit être antérieure à celle destination R. Leurs fonctions d'ordonnancement respectives Θ_S et Θ_R seront donc légales si et seulement si :

$$\Theta_S(\vec{x}_S) \prec \Theta_R(\vec{x}_R), \forall \vec{x}_S \in D_S, \forall \vec{x}_R \in D_R \tag{4.2}$$

Feautrier a montré [56] qu'à l'inverse des ordonnancements monodimensionnels, il est toujours possible de déterminer un ordonnancement multidimensionnel légal pour une SCoP.

Transformer un programme dans le modèle polyédrique consiste à appliquer un ordonnancement affine pour chaque instruction. Le programme transformé reste dans le modèle polyédrique. La transformation est légale si tous ses ordonnancements affines sont légaux.

Remarque : Le nom utilisé pour désigner un ordonnancement affine change en fonction de son domaine d'application. Les ordonnancements peuvent être vus comme des fonctions transformant l'espace d'itération et appelés transformations. Dans le cas de la génération de code on parle la plupart du temps de fonction de *scattering*. Ces appellations désignent le même formalisme mathématique. Pour plus de clarté, une transformation affine désigne ici les ordonnancements affines de toutes les instructions d'une SCoP.

Définition 14 (Transformation affine) *Soit une SCoP contenant n instructions $S_1, ..., S_n$. Une transformation affine de cette SCoP est formée par l'ensemble des couples :*

$$\langle S_k, \Theta_{S_K}(\vec{x}_{S_k}) \rangle, \forall k \in [1, n]$$

Les matrices des ordonnancements affines d'une transformation ont toutes la même dimension :

$$dim(T_{S_i}) = dim(T_{S_j}), \forall i, j \in [1, n]$$

Expressivité d'une transformation affine Les transformations usuelles (permutation, décalage, interchange, etc.) d'un nid de boucles peuvent être exprimées dans le modèle polyédrique par des transformations unimodulaires [193]. Une transformation unimodulaire Θ_S est décrite par T_S, une matrice carrée dont le déterminant vaut 1 ou -1. Une transformation unimodulaire est donc par définition inversible. Le modèle polyédrique supporte également les transformations non unimodulaires [158]. Les résultats de celles-ci peuvent être des polyèdres non denses modélisables en introduisant de nouvelles dimensions ou par un formalisme adapté [140, 82].

Une transformation affine possède une sémantique forte pouvant exprimer une séquence complexe de transformations classiques. Toutes les transformations de boucles peuvent être représentées dans le modèle polyédrique [194] et de nombreux travaux d'optimisation automatique [56, 71, 26, 147] tirent parti de cette expressivité. Il est notamment possible de modéliser l'ensemble des optimisations légales d'un programme dans un unique espace convexe.

Le résultat d'une transformation monodimensionnelle légale est une boucle séquentielle contenant éventuellement un ensemble de boucles parallèles et donc mutuellement permutables. Une transformation multidimensionnelle introduit quant à elle une dimension par ligne des matrices T_S. L'hyperplan définissant la k-ième dimension d'un ordonnancement Θ_S est noté Θ_S^k. Si un hyperplan Θ_S^i ne dépend pas des indices de boucles $(a_{k,1}, ..., a_{k,dim(\vec{x}_S)} = 0)$, la dimension i est une dimension scalaire. Les dimensions scalaires imposent un ordre explicite entre les instances d'instructions transformées. Les instances partageant la même valeur d'une dimension scalaire sont fusionnées, les autres sont distribuées.

Exemple de transformation affine Dans l'exemple de la figure 4.5, on applique une transformation sur un programme comportant trois instructions réparties dans trois nids de boucles. Les matrices de la transformation (figure 4.4) correspondent aux ordonnancements multidimensionnels suivants :

$$\begin{cases} \Theta_{S_1}(i,j) &= (i,j,0,0,0) \\ \Theta_{S_2}(i,j,k) &= (i,j,1,0,k) \\ \Theta_{S_3}(i,j,k) &= (k,j,1,1,i) \end{cases}$$

La transformation comporte donc cinq dimensions dont deux scalaires (c_3, c_4). Le code transformé (figure 4.5b) montre l'entrelacement des instances d'instructions dans un unique nid de boucles obtenu après transformation.

$$T_{S_1} = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{array} \begin{array}{c} \begin{array}{ccccc} i & j & k & N & c \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad T_{S_2} = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{array} \begin{array}{c} \begin{array}{ccccc} i & j & k & N & c \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad T_{S_3} = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{array} \begin{array}{c} \begin{array}{ccccc} i & j & k & N & c \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

FIGURE 4.4 – Matrices d'une transformation affine multidimensionnelle pour le programme C de la figure 4.5a.

(a) Code original

```
1  for (i = 0; i < N; i++) {
2    for (j = 0; j < N; j++) {
3  S1:  C[i][j] = 0;
4    }
5  }
6  for (i = 0; i < N; i++) {
7    for (j = 0; j < N; j++) {
8      for (k = 0; k < N; k++) {
9  S2:   C[i][j] += A[i][k] * B[k][j];
10     }
11   }
12 }
13 for (i = 0; i < N; i++) {
14   for (j = 0; j < N; j++) {
15     for (k = 0; k < N; k++) {
16 S3:   D[i][j] += E[i][k] * C[k][j];
17     }
18   }
19 }
```

(b) Code transformé

```
1  for(c1=0;c1<N;c1++){
2    for (c2=0; c2<N; c2++) {
3      C[c1][c2] = 0;
4      for (c5=0; c5 <N; c5++) {
5        C[c1][c2] += A[c1][c5]*B[c5][c1];
6      }
7      for (c5=0; c5 <N; c5++) {
8        D[c5][c2] += E[c5][c1]*C[c1][c2];
9      }
10   }
11 }
```

FIGURE 4.5 – Application d'une transformation affine à un programme C effectuant le produit de trois matrices A,B et E.

4.3.2 Transformation monodimensionnelle

Les travaux sur l'ordonnancement monodimensionnel construisent un problème d'optimisation linéaire pour déterminer les meilleurs coefficients d'ordonnancement. Cependant, ces approches ciblent généralement une architecture parfaite, c'est-à-dire sans contraintes de ressources. La prise en compte de ces ressources est complexe et un moyen simple de s'en abstraire est d'explorer l'espace des ordonnancements légaux et d'en sélectionner un qui offre de bonnes performances après compilation (approche source à source).

Algorithme d'ordonnancement monodimensionnel Le principe de l'algorithme est de résoudre un problème de programmation linéaire pour trouver les coefficients $\vec{T}_{S_k}$ optimaux de chaque instruction S_k.

L'ensemble des contraintes de dépendances de données est exprimé dans un seul polyèdre de dépendances D dont le nombre de dimensions pour $S_1, ..., S_k$ instructions est donné par la relation suivante :

$$dim(D) = dim(\vec{x}_{S_1}) + ... + dim(\vec{x}_{S_k}) + dim(\vec{n}) + 1$$

Pour que l'ordonnancement soit valide, l'ordre lexicographique des dépendances de données doit être préservé. Ainsi, la contrainte 4.2, page 90 pour une dépendance de donnée $e_{S \to R}$ se formule par une contrainte dite de causalité :

$$\theta_R(\vec{x}_R) - \theta_S(\vec{x}_S) - 1 \geq 0 \tag{4.3}$$

Avec $\theta_R(\vec{x}_R)$ et $\theta_S(\vec{x}_S)$ qui respectent les prototypes d'ordonnancement suivants :

$$\theta_R(\vec{x}_R) = t_{R,0} + t_{R,1} \cdot x_1 + \cdots + t_{R,p} \cdot x_p + t_{R,p+1} \cdot n_1 + \cdots + t_{R,p+q} \cdot n_q$$
$$\theta_S(\vec{x}_S) = t_{S,0} + t_{S,1} \cdot x_1 + \cdots + t_{S,p} \cdot x_p + t_{S,p+1} \cdot n_1 + \cdots + t_{S,p+q} \cdot n_q$$

Malheureusement, les causalités des dépendances ne sont pas des contraintes linéaires puisque les coefficients d'ordonnancements sont des variables à déterminer (e.g., le terme $t_{R,1} \cdot x_1$ n'est pas linéaire), il n'est donc à priori pas possible d'exprimer le problème d'ordonnancement en programmation linéaire. Néanmoins, il existe deux techniques qui permettent de contourner ce problème : (a) méthode des sommets [156] (b) utilisation de la forme affine du lemme de Farkas [55].

La méthode des sommets utilise la représentation duale du polyèdre et récupère la liste des sommets du polyèdre de dépendances pour y appliquer les conditions de légalité. En effet, si une fonction affine est positive dans un polyèdre alors elle l'est également en ses sommets. Un solveur de programmation linéaire détermine ensuite la solution optimale au problème d'ordonnancement.

L'utilisation du lemme de Farkas (cf. lemme 1) permet également de construire un problème d'optimisation linéaire. Le principe est d'exprimer l'ordonnancement de chaque instruction comme une combinaison des faces de leurs domaines de définition respectifs. Les formes obtenues sont ensuite utilisées dans une reformulation des contraintes de légalité qui dans le cas de dépendances uniformes amène à une simple contrainte linéaire. Pour des dépendances non uniformes, la forme affine du lemme de Farkas permet d'obtenir un ensemble de contraintes linéaires en éliminant les coefficients de Farkas (i.e., les coefficients λ_k) par projection de Fourrier Motzkin. De même que pour la méthode des sommets, un solveur de programmation linéaire détermine la solution optimale. Cette méthode est généralement préférée à celles des sommets car le nombre de contraintes y dépend du nombre de faces du polyèdre et non du nombre de sommets.

Lemme 1 (Forme affine du lemme de Farkas) *Soit P un polyèdre non vide définit par p contraintes. Une forme affine ψ est positive dans P si et seulement si c'est une combinaison positive des faces de P.*

$$\psi(\vec{x}) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k \vec{x} + b_k) \text{ avec } \lambda_0, \lambda_1, \ldots, \lambda_p \geq 0$$

Exploration de l'espace des ordonnancement monodimensionnels L'algorithme de Pouchet [148] explore de manière itérative l'ensemble des ordonnancements monodimensionnels légaux pour sélectionner empiriquement le plus performant pour une cible matérielle donnée.

La construction de cet espace est détaillée à travers l'exemple du produit vectoriel de la figure 4.6 possédant deux dépendances : $e_{S_1 \to S_2}$ et $e_{S_2 \to S_2}$.

```
1
2     for(i=0;i<N;i++)
3  S1: T[i]=0;
4     for(j=0;j<M;j++)
5  S2: T[i]+=A[i][j]*B[j];
6     }
```

FIGURE 4.6 – Produit d'une matrice par un vecteur

Le polyèdre $P_{e_{S_1 \to S_2}}$ est représenté par la matrice suivante :

$$P_{e_{S_1 \to S_2}} = \begin{array}{c} \\ \lambda_1 \\ \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \\ \lambda_7 \end{array} \begin{array}{c} \begin{array}{ccccc} i_{S_1} & i_{S_2} & j_{S_2} & N & 1 \end{array} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ \\ 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{pmatrix} \end{array}$$

La première ligne de la matrice représente une équation ($i_{S_1} = i_{S_2}$), les autres des inéquations positives. L'objectif est de déterminer les ordonnancements $\theta_{S_1}(\vec{x}_{S_1})$ et $\theta_{S_2}(\vec{x}_{S_2})$ dont les prototypes sont :

$$\theta_{S_1} = t_{1,1}.i_{S_1} + t_{1,2}.N + t_{1,3}$$
$$\theta_{S_2} = t_{2,1}.i_{S_2} + t_{2,2}.j_{S_2} + t_{2,3}.N + t_{2,4}$$

Respecter la dépendance $e_{S_1 \to S_2}$ implique que :

$$\theta_{S_1}(\vec{x}_{S_1}) \prec \theta_{S_2}(\vec{x}_{S_2}), \forall \vec{x}_{S_1}, \vec{x}_{S_2} \in P_{e_{S_1 \to S_2}}$$

Soit une variable $\Delta_{S1,S2}$ telle que :

$$\Delta_{S1,S2} = \theta_{S_1}(\vec{x}_{S_1}) - \theta_{S_2}(\vec{x}_{S_2}) - 1$$

La dépendance sera alors satisfaite si et seulement si $\Delta_{S1,S2} \geq 0$. En utilisant la forme affine du lemme de farkas (cf. lemme 1) on obtient :

$$\Delta_{S1,S2} = \lambda_0 + \lambda_1(i_{S_1} - i_{S_2}) + \lambda_2(i_{S_1}) + \lambda_3(N - i_{S_1}) + \lambda_4(i_{S_2}) + \lambda_5(N - i_{S_2}) + \lambda_6(j_{S_2}) + \lambda_7(N - j_{S_2})$$

À partir des prototypes des ordonnancements et de la forme précédente de $\Delta_{S1,S2}$, on en déduit le système :

$$\begin{cases} -t_{1,1} = \lambda_1 + \lambda_2 - \lambda_3 \\ -t_{2,1} = -\lambda_1 + \lambda_4 - \lambda_5 \\ -t_{2,2} = \lambda_6 - \lambda_7 \\ t_{2,3} - t_{1,2} = \lambda_3 + \lambda_5 + \lambda_7 \\ t_{2,4} - t_{1,3} = \lambda_0 + 1 \end{cases}$$

Après résolution, on obtient un polyèdre $T_{S_1 \to S_2}$ dont les dimensions sont les coefficients d'ordonnancement $t_{i,j}$ de chaque instruction S_i. Chaque point positionné dans ce polyèdre correspond donc à un ordonnancement satisfaisant la dépendance $e_{S_1 \to S_2}$. Pour déterminer l'ensemble des ordonnancements légaux du programme, il suffit de procéder de la même manière pour chaque dépendance et de construire le polyèdre d'ordonnancement global T :

$$T = \bigcap_{i,j \forall e_{S_i \to S_j} \in E} T_{S_i \to S_j}$$

Le nombre de points dans T peut être très grand ou même infini. L'exploration exhaustive des versions du programme n'est donc possible qu'en bornant les coefficients des indices de boucles (et non des paramètres). Il est important de noter que des faibles coefficients d'indices limiteront le coût du contrôle du code généré. Les expérimentations [148] ont montré que les bornes $[-1, 1]$ sont suffisantes dans la plupart des cas.

4.3.3 Transformation multidimensionnelle

Il n'est pas toujours possible de déterminer un ordonnancement monodimensionnel satisfaisant toutes les contraintes de légalité. Néanmoins, pour n'importe quelle SCoP il existe toujours un ordonnancement multidimensionnel. Les techniques d'ordonnancement multidimensionnel s'appuient sur la notion de dépendance faiblement ou fortement satisfaite à une dimension donnée.

Définition 15 (Dépendance faiblement satisfaite) *Une dépendance $e_{S_1 \to S_2}$ est faiblement satisfaite à une dimension d si*

$$\forall \langle \vec{x}_{S_1}, \vec{x}_{S_2} \rangle \in P_{e_{S_1 \to S_2}}, \theta^d_{S_1}(\vec{x}_{S_1}) \leq \theta^d_{S_2}(\vec{x}_{S_2}) \tag{4.4}$$

Définition 16 (Dépendance fortement satisfaite) *Une dépendance $e_{S_1 \to S_2}$ est fortement satisfaite à une dimension d si*

$$\forall \langle \vec{x}_{S_1}, \vec{x}_{S_2} \rangle \in P_{e_{S_1 \to S_2}}, \theta^d_{S_1}(\vec{x}_{S_1}) < \theta^d_{S_2}(\vec{x}_{S_2}) \tag{4.5}$$

Une dépendance fortement satisfaite à une dimension d n'impose aucune contrainte pour les dimensions $k > d$. Pour toutes les dimensions précédant celle de forte satisfaction, les contraintes de faible satisfaction doivent être respectées. Ainsi, pour déterminer une transformation multidimensionnelle légale, il est nécessaire d'identifier la dimension de forte satisfaction pour chaque dépendance.

Algorithme glouton de Feautrier L'objectif de l'algorithme de Feautrier [56] est de minimiser le nombre de dimensions nécessaires pour obtenir un ordonnancement légal. Chaque itération de l'algorithme correspond à une nouvelle dimension d'ordonnancement et maximise le nombre de dépendances fortement satisfaites à ce niveau. Une variable binaire z_e indique pour chaque dépendance analysée si elle fortement satisfaite ($z_e = 1$) ou non ($z_e = 0$). Mis à part cette variable, l'énoncé et la résolution du problème d'optimisation sont similaires à l'ordonnancement monodimensionnel. Seules les dépendances non fortement satisfaites au niveau courant sont analysées. Puisque l'algorithme minimise d le nombre de dimensions nécessaires, il maximise le parallèlisme et a été prouvé comme étant optimal dans son contexte [190].

Espace convexe L'ordonnancement de Feautrier maximise le parallélisme pour une architecture parfaite (i.e., sans contraintes de ressources), cependant la prise en compte des ressources de l'architecture et du comportement global du compilateur est loin d'être triviale. Dans ce contexte, l'exploration itérative de Pouchet est une alternative pertinente qui est généralisable au cas multidimensionnel mais qui souffre alors des choix explicites de l'algorithme glouton de Feautrier (i.e., niveau auquel une dépendance est fortement satisfaite).

Il est néanmoins possible de formuler une espace d'ordonnancement multidimensionnel convexe en considérant les niveaux de forte satisfaction des dépendances comme des dimensions supplémentaires de cet espace [149]. Le principe de cette formalisation est d'associer une variable $\delta^k_{e_{S\to R}} \in [0,1]$ pour chaque dépendance $e_{S\to R}$ et pour chaque dimension k de l'ordonnancement recherché. Cette variable indique alors si cette dépendance est faiblement ($\delta^k_{e_{S\to R}} = 0$) ou fortement ($\delta^k_{e_{S\to R}} = 1$) satisfaite à la dimension k.

Le problème est alors de modéliser le fait que si une dépendance est fortement satisfaite à un niveau k alors pour toute dimension $d > k$, l'ordonnancement est assuré comme étant légal et aucune contrainte supplémentaire n'est nécessaire. La technique présentée dans [149] s'appuie sur la *nullification* de la contrainte (4.5). Cette nullification consiste à déterminer une borne basse lb telle que la relation suivante soit toujours vraie à la dimension k :

$$\theta^k_R(\vec{x}_R) - \theta^k_S(\vec{x}_S) > lb \tag{4.6}$$

L'intuition derrière la contrainte (4.6) est que si une borne basse lb existe alors elle peut être assimilée à $-\infty$, le fait de remplacer le zéro par lb dans la contrainte (4.5) ne contraindra pas l'espace des ordonnancements puisque la contrainte sera toujours respectée. Or, il a été montré dans [186, 149] qu'il existe toujours [5] un entier K suffisamment grand pour que la contrainte suivante soit respectée :

$$min(\theta^k_R(\vec{x}_R) - \theta^k_S(\vec{x}_S)) > -K \cdot \vec{n} - K \tag{4.7}$$

À partir des contraintes (4.5), (4.6) et (4.7), il est donc possible d'exprimer l'espace des ordonnancements multidimensionnels dans une forme convexe (lemme 2 [149]).

Lemme 2 (Forme convexe des ordonnancements multidimensionnels) *Soit $e_{S\to R}$, une dépendance dont le domaine est $D_{e_{S\to R}}$, la sémantique du programme est préservée si les contraintes suivantes sur θ_R et θ_S (fonctions d'ordonnancement de dimension m) sont respectées :*

$$(i) \qquad \forall p \in \{1,\dots,m\}, \delta^p_{e_{S\to R}} \in \{0,1\} \tag{4.8}$$

$$(ii) \qquad \sum_{p=1}^{m} \delta^p_{e_{S\to R}} = 1 \tag{4.9}$$

$$(iii) \qquad \forall p \in \{1,\dots,m\}, \theta^p_R(\vec{x}_R) - \theta^p_S(\vec{x}_S) \geq \delta^p_{e_{S\to R}} - \sum_{k=1}^{p-1} \delta^k_{e_{S\to R}}.(K.\vec{n} + K) \tag{4.10}$$

4.3.4 Ordonnancement structuré

Les ordonnancements affines multidimensionnels peuvent rapidement poser des problèmes de passage à l'échelle. En effet, à titre d'exemple, déterminer une transformation multidimensionnelle d'une SCoP contenant quatre instructions réparties dans des nids de boucles de profondeur 3 et dont les bornes dépendent de trois paramètres, implique de parcourir un espace à $4 * (3 * (3 * 3 + 1)) = 120$ dimensions. De plus, si cette SCoP contient par exemple 5 dépendances, la forme convexe introduira 15 dimensions supplémentaires.

5. si les bornes paramètrées des domaines ne sont pas négatives

Ce constat a motivé la conception de techniques [57, 157] cherchant à bénéficier d'une hiérarchisation des ordonnancements pour appliquer une approche de type diviser pour régner, on parle alors d'*ordonnancement structuré*. De plus, cette hiérarchisation est généralement présente naturellement dans les programmes puisque ceux-ci sont souvent écrits en factorisant les comportements communs par des fonctions pouvant s'appeler mutuellement. Il semble alors naturel de chercher à capitaliser les optimisations de chacune de ces fonctions pour éviter de calculer un nouvel ordonnancement à chaque fois qu'on y fait appel.

Cependant, l'ordonnancement affine structuré d'un programme est loin d'être trivial puisque si chacune des fonctions est ordonnancée indépendamment, il est loin d'être garanti que le programme complet puisse l'être à partir des ordonnancements sélectionnés pour ces fonctions.

L'approche proposée par Feautrier [57] s'inspire des réseaux de processus de Kahn pour décrire le problème sous forme de processus communiquant par des canaux. Chaque processus est alors vu comme un super-nœud du PRDG global du réseau de processus et l'existence d'un ordonnancement légal de chaque processus est une condition nécessaire mais pas suffisante à l'existence d'un ordonnancement légal du réseau. Le principe de l'algorithme est d'exprimer, de manière modulaire, des contraintes de légalité qui ne portent que sur les ordonnancements des canaux de communication. Ces contraintes peuvent être construites pour chaque processus et indépendamment du réseau global. Une fois que toutes ces contraintes sont construites, une solution légale à l'ordonnancement des communications peut être identifiée par une résolution standard avec un solveur de programmation linéaire. La solution obtenue est ensuite injectée dans les contraintes de chaque processus qui peuvent alors être ordonnancés indépendamment tout en garantissant le respect des contraintes globales du réseau.

Cependant, l'ordonnancement structuré de Feautrier n'est applicable que pour des ordonnancements monodimensionnels. En effet, le caractère glouton de l'algorithme d'ordonnancement multidimensionnel [56] ne permet pas d'assurer la modularité des contraintes des processus : le choix de la dimension satisfaisant fortement une dépendance y est déterminé explicitement lors d'une itération de l'algorithme ; les contraintes de communication ne pourront donc pas être exprimées indépendamment du contexte global du réseau.

4.3.5 Pavage

Le pavage [95, 202] consiste à découper un espace d'itération en tuiles régulières et uniformes (cf. figure 4.7). Le parcours du domaine d'itération consiste alors à énumérer chaque tuile et chaque point des tuiles. Un pavage est défini par la forme et la taille des tuiles. Ce découpage permet de favoriser la localité spatiale des données, de limiter les communications ou encore d'exploiter du parallélisme gros-grain. Le pavage est une transformation ciblant une architecture en regroupant par exemple les données nécessaires au traitement d'une tuile en fonction des capacités mémoires de la cible. La gestion d'une hiérarchie mémoire peut également être optimisée en utilisant un pavage hiérarchique où chaque niveau est dimensionné en fonction du niveau dans la hiérarchie mémoire (e.g., cache L1, L2).

Partionnement de l'espace d'itération en tuiles régulières La taille des tuiles définit la régularité du pavage et peut être dépendante de paramètres symboliques et l'on parle alors de pavage paramétré. La forme d'une tuile peut être rectangulaire, parallélépipèdiques, triangulaire, etc. La

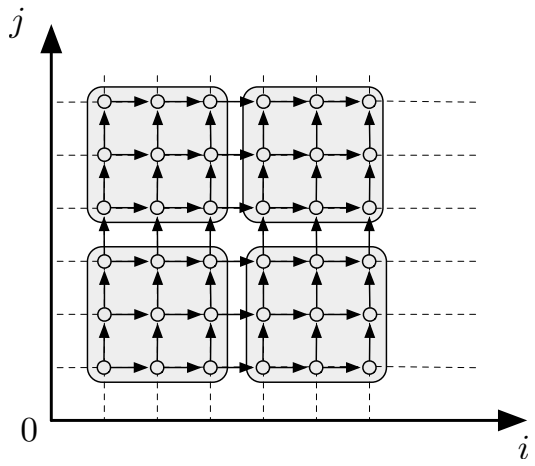


FIGURE 4.7 – Pavage rectangulaire 3x3 d'un espace 2D

plupart des travaux s'intéressent uniquement aux formes les plus simples, c'est-à-dire rectangulaires et parallélépipèdiques. Celles-ci sont définies par un ensemble d'hyperplans linéairement indépendants. L'intersection des demi-espaces modulo leurs tailles forme l'ensemble des points entiers se trouvant dans une tuile du domaine d'itération.

Définition 17 (Forme d'une tuile) *Soit un domaine d'itération D de dimension d, la forme d'une tuile dans D est donnée par une matrice H exprimant d hyperplans linéairement indépendants :*

$$H = \begin{pmatrix} \vec{h}_1 \\ \vdots \\ \vec{h}_d \end{pmatrix}, \forall (a_1, \ldots, a_d) \in K, a_1.\vec{h}_1 + \ldots + a_n.\vec{h}_n = \vec{0} \Rightarrow a_1 = \ldots = a_d = 0$$

Pour qu'un pavage soit valide, un ordre d'énumération tenant compte des dépendances de données doit exister. Cet ordre existe si et seulement si les tuiles forment des sous-espaces convexes. Autrement dit, une tuile t_1 ne doit pas avoir de dépendances de données sortantes vers une autre tuile t_2 si celle-ci possède une dépendance vers t_1. La figure 4.8 illustre deux exemples de pavage $2 * 2$ dans un espace à deux dimensions. La tuile t_2 dépend de t_1 et t_1 dépend de t_2, les tuiles ne sont donc pas convexes et ce pavage est illégal. Les conditions d'Irigoin et Triolet [95] garantissent la légalité d'un pavage pour un nid de boucles parfait. Elles ont été généralisées aux autres nids de boucles par Bondhugula [26] en associant la recherche d'un ordonnancement affine légal à la notion de pavage. Les coordonnées dans l'espace pavé de chaque instance d'instruction S_k sont alors déterminées par une fonction d'ordonnancement affine ϕ_{S_k}, où chaque ligne i de la matrice T_{S_k} est un hyperplan de pavage $\phi^i_{S_k}$.

Théorème 4.3.1 (Légalité d'un hyperplan de pavage) *Un hyperplan de pavage ϕ^i est légal si et seulement si pour chaque lien de dépendance $e_{S_i \to S_j} \in E$, la contrainte suivante est vérifiée :*

$$\phi^i_{S_j}(\vec{t}) - \phi^i_{S_i}(\vec{s}) \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_{e_{S_i \to S_j}}$$

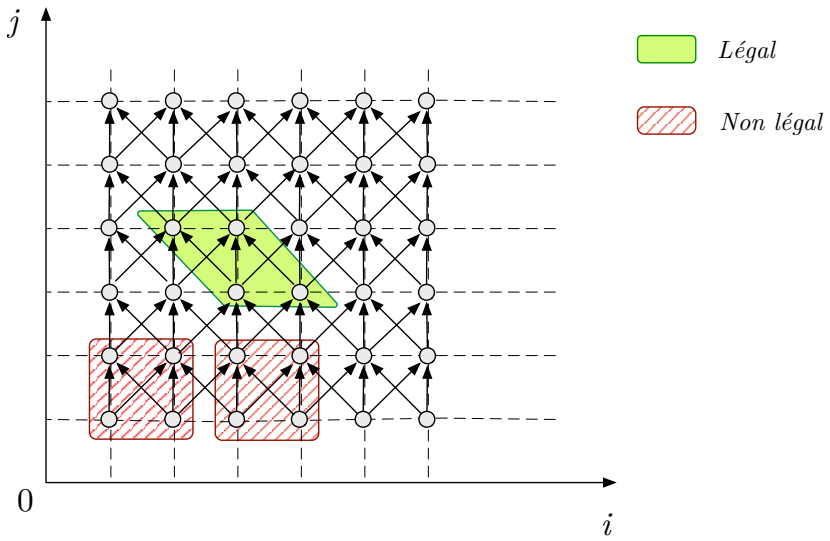


FIGURE 4.8 – Légalité du pavage

Génération de boucles pavées La génération du code parcourant l'ensemble des points du domaine original dans l'espace pavé implique l'utilisation de nouvelles dimensions énumérant la liste des tuiles obtenues et les points de chaque tuile. Dans le cas de tuiles de taille fixe, le problème est proche de celui dit de parcours d'un polyèdre. Ce dernier consiste à générer un nid de boucles énumérant chaque point entier d'un polyèdre. Les difficultés se trouvent notamment dans la gestion de la complexité du contrôle et dans l'efficacité du parcours qui doit éviter d'énumérer des points inutiles. Parmi les techniques développées, l'algorithme de Quilleré [154] et son extension implémentée dans l'outil CLooG [18] est à ce jour le plus utilisé. Cependant, la complexité double exponentielle de l'algorithme combinée à celle du polyèdre dont le nombre de dimensions a été augmenté (dimensions d'énumération des tuiles et des points) ont amener à considérer d'autres algorithmes [75, 162, 108, 85] dédiés au problème de génération de boucles pavées. De plus, les algorithmes [108, 85] adressent la problématique du pavage paramétré.

Partitionnement automatique Les méthodes d'optimisation basées sur le partitionnement évaluent les dates d'exécution des instances d'instructions en identifiant la tuile et la position relative de chaque instance. Optimiser un programme par partitionnement consiste alors à déterminer une forme et une taille de tuile en adéquation avec l'architecture et les performances souhaitées. La forme d'une tuile conditionne le volume de communications entre les tuiles c'est-à-dire le nombre de dépendances de données à satisfaire entre chaque tuile. Les tailles des tuiles influent quant à elles sur le volume de calcul réalisé dans chaque bloc. Il est important de noter que l'efficacité d'un pavage dépend de la cible architecturale. Les méthodes sont donc en majorité adaptées à des architectures spécifiques. Le pavage a été principalement étudié sous deux axes d'optimisation : réduire les durées de communications et minimiser la durée d'exécution parallèle.

Les approches optimisant la localité partent du constat qu'un des facteurs d'accélération se situe dans une meilleure gestion de l'alimentation en données. Favoriser la localité dans les tuiles et entre les tuiles limite les accès à des niveaux profonds et donc couteux dans la hiérarchie mémoire. Relativement

peu de travaux traitent le problème dans son ensemble, c'est-à-dire en déterminant la forme et la taille des tuiles par une résolution conjointe. Afin de réduire la complexité du problème, celui-ci est souvent divisé en deux phases d'optimisation disjointes : (1) détermination de la forme, (2) sélection de la taille des tuiles.

Ramanujam et al. formulent un problème d'optimisation linéaire pour parcourir l'ensemble des matrices H légales minimisant les communications par tuile [159]. Il s'agit néanmoins d'un sous-espace d'optimisation où la matrice H doit être unimodulaire et triangulaire inférieure. Andonov et al. [9, 8] proposent des techniques pour déterminer la forme et la taille de tuiles semi-obliques pour un espace bidimensionnel.

Les travaux [30, 201] sont dans la lignée de la minimisation des communications par tuiles proposée dans [159], mais s'intéressent uniquement à la forme des tuiles. L'approche décrite dans [30] souligne que la restriction sur la forme des matrices H pose un réel problème et propose une fonction de coût indépendante de la taille. Il est montré dans [201] que le problème peut être reformulé en réduisant l'espace de recherche sans perte d'optimalité.

PLUTO [26, 143] est un outil qui transforme automatiquement le programme pour obtenir une forme de pavage légale favorisant la localité pour une taille fixée. Un des intérêts de cette approche est que l'espace des transformations exploré correspond au pavage, mais également à toutes les transformations usuelles (décalage, fusion, distribution, etc.). L'algorithme est itératif (cf. figure 4.9) et détermine les hyperplans de pavage qui minimiseront la distance δ_e entre les dates d'exécution source et destination de chaque dépendance :

$$\delta_e(\vec{s},\vec{t}) = \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s})$$

A chaque itération, c'est-à-dire pour chaque hyperplan, l'exploration est basée sur la résolution d'un problème d'optimisation linéaire contenant les contraintes de légalité des dépendances (cf. théorème 4.3.1) non satisfaites jusqu'à présent. L'algorithme s'arrête quand on a obtenu assez d'hyperplans. La fonction δ_e peut ne pas être linéaire, mais elle est toujours bornée par une forme affine $v(\vec{n})$ dépendante uniquement des paramètres $\vec{n}$:

$$v(\vec{n}) = u.\vec{n} + w$$
$$v(\vec{n}) - \delta_e(\vec{s},\vec{t}) \geq 0 \text{ (contrainte de borne de } \delta_e)$$

Aux contraintes de dépendances de données s'ajoutent donc les contraintes de bornes. L'utilisation de la forme affine du lemme de Farkas associée à l'élimination des variables de Fourrier Motzkin permet d'exprimer l'ensemble des contraintes dans un polyèdre où le minimum lexicographique correspond à la solution optimale. Avant chaque recherche, des contraintes d'orthogonalité sont ajoutées au problème afin de garantir que l'hyperplan solution soit linéairement indépendant des précédents (cf. définition 17). Dans les cas où il est impossible de trouver une solution, l'ajout de dimensions scalaires permet de satisfaire les dépendances problématiques. L'insertion de ces dimensions correspond à une distribution de boucles. Le problème correspond à sélectionner les dépendances qui seront satisfaites séquentiellement, on parle alors de coupe de dépendance. Bondhugula propose trois heuristiques pour couper les dépendances : *nofuse, smartfuse* et *maxfuse* classées par ordre décroissant d'agressivité. La dernière est donc celle qui coupera le moins de dépendances, elle est néanmoins la plus coûteuse en cas d'échecs répétés.

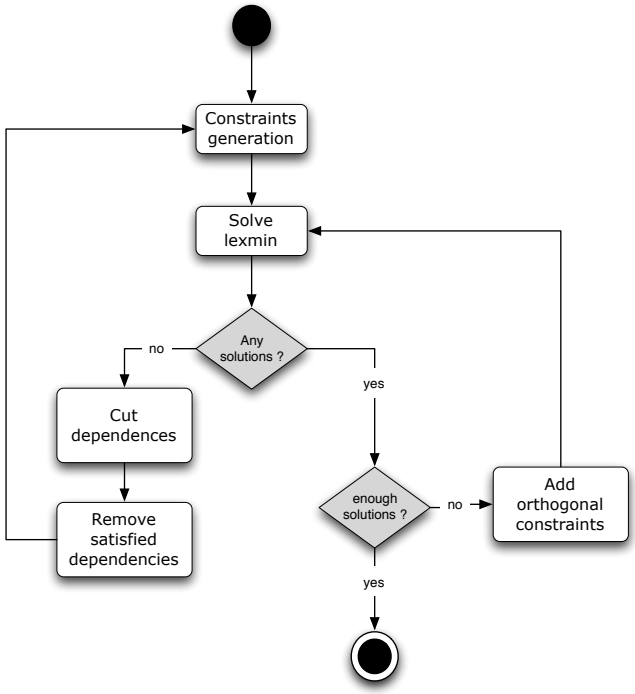


FIGURE 4.9 – Algorithme itératif PLUTO (diagramme d'activités)

La taille des tuiles est un paramètre difficile à déterminer et de nombreux modèles de coûts [52, 43, 35, 163, 91, 161] existent et sont basés notamment sur la limitation des interférences des caches. Il est intéressant de noter que beaucoup de ces modèles sont en fait exprimable dans un formalisme commun. Les travaux [91, 161] recensent ainsi les principales fonctions de coût existantes et les différentes variables utilisées (taille du cache, taille des pages mémoires, etc.). Dans [161] le formalisme utilisé est basé sur des fonctions polynomiales positives (appelées posynomiales) et le problème est résolu via un solveur de programmation géométrique tirant parti de la positivité de ces fonctions.

La complexité de la sélection des tailles des tuiles est induite par celle de l'architecture cible. Dans le cas d'architectures telles que des processeurs multicœurs et leurs hiérarchies mémoires complexes, les résultats des optimisations sont difficilement prévisibles. Partant de ce constat, certains explorent les tailles de tuiles à l'aide de méta-heuristiques [37, 59] guidées par des modèles de coûts simplifiés. L'approche [207] propose un apprentissage automatique et spécifique à l'architecture.

Minimiser la durée d'exécution parallèle suppose d'avoir un modèle de coût évaluant la durée d'exécution de l'ensemble des tuiles sur l'architecture. Un tel modèle est difficile à déterminer et à optimiser dans le cas d'architecture complexe. Les méthodes [90, 87, 76] cherchent une forme de pavage minimisant la durée d'exécution parallèle d'un nid de boucles parfait aux dépendances uniformes. Certaines approches simplifient le problème en considérant des communications constantes [90] ou négligeables [87]. L'algorithme [76] analyse les topologies possibles d'un pavage exécuté sur une grille de calcul. Pour une taille donnée, la forme est déterminée par une recherche exhaustive tenant compte des communications dans la fonction de coût.

4.4 Synthèse

La transformation d'un programme est considérablement simplifiée par le modèle polyédrique. Une instruction est définie dans un domaine d'itération et correspond à un ensemble d'instances positionnées par leurs vecteurs d'itérations. L'expressivité du formalisme permet d'envisager des combinaisons de transformations sous forme de simples fonctions affines. Si un certain nombre de verrous ont été en grande partie levés (analyse des dépendances, génération de code), la phase d'optimisation reste un problème ouvert. Les algorithmes de Feautrier déterminent des ordonnancements théoriques optimaux mais ne tiennent pas compte des ressources. L'approche itérative de Pouchet explore, quant à elle, l'espace des ordonnancements en évaluant empiriquement les performances de l'ordonnancement après avoir regénéré le code et l'avoir exécuté sur la cible matérielle.

Le pavage permet notamment d'améliorer la localité et donc de réduire les pénalités d'alimentation en données d'une architecture. La encore, la gestion des ressources reste critique. Certaines techniques attaquent le problème en utilisant des modèles de coûts linéaires. D'autres ont recours à une exploration itérative guidée par un modèle simplifié et/ou un apprentissage.

La complexité de la recherche d'une solution à un problème d'ordonnancement peut être réduite en utilisant un algorithme d'ordonnancement structuré qui décomposera la résolution du problème en plusieurs sous-problèmes plus simples. Cependant, la généralisation de cet algorithme au cas multidimensionnel restait, jusqu'à ce jour, un problème ouvert. En effet, nous montrerons dans le chapitre 6 que l'algorithme d'ordonnancement structuré de Feautrier est en fait généralisable aux cas multidimensionnels grâce à la forme convexe proposée par Pouchet [149].

Chapitre 5

Modèle polyédrique et optimisations non linéaires via la programmation par contraintes

Dans le chapitre précédent, nous avons présenté le modèle polyédrique et ses applications en compilation optimisante. Ce chapitre s'intéresse à la jonction du modèle polyédrique et de la programmation par contraintes. L'objectif est de fournir la possibilité d'exprimer, simplement, une contrainte globale qui capture la notion de domaine polyédrique (non paramétré) afin de pouvoir utiliser les ordonnancements affines dans des problèmes d'optimisation non linéaires. Cette contrainte sera utilisée dans le chapitre 6 pour étendre la technique de couverture de graphe, présentée dans la sous-section 3.3.1, aux problématiques de transformation affine d'une SCoP.

Sommaire

5.1 Introduction

Par définition, le modèle polyédrique n'est applicable que dans le cadre de problèmes d'optimisation linéaire. L'objectif premier de ce chapitre est d'étendre les analyses et optimisations polyédriques à un cadre plus large permettant son utilisation dans un flot de sélection d'instructions spécialisées (cf. chapitre 6).

Si les optimisations polyédriques et la PPC ont rarement été abordées conjointement [80, 138], il s'avère que l'intégration de la programmation linéaire et de la PPC au sein d'un même algorithme d'optimisation est un thème actif de recherche. Ces travaux s'intéressent particulièrement à la complémentarité des deux approches. La programmation par contrainte offre une expressivité élevée particulièrement efficace pour la recherche locale (arc-consistance). À l'inverse, la programmation linéaire considère le problème d'une manière plus globale. Son formalisme strict favorise le développement de techniques d'optimisation complexes et peu dépendantes de l'énoncé du problème. Un CSP peut être reformulé sous forme de problème linéaire en nombres entiers. Cependant, dans [24, 48] il est montré que cette formulation introduit un nombre important de variables intermédiaires et de contraintes linéaires pouvant rendre inefficace la recherche d'une solution.

Les efforts d'intégration de la PPC et la PL peuvent être regroupés autour de deux axes principaux : modélisation hybride et algorithmes hybrides. La modélisation hybride consiste à laisser au concepteur le choix explicite des techniques utilisées. Celui-ci regroupe alors les contraintes linéaires du problème dans une contrainte globale spécifique. Cette contrainte globale bénéficie des techniques de relaxation lagrangiennes [58, 50] pour guider la recherche en fournissant une borne basse (ou haute) à la fonction de coût à minimiser (ou maximiser). Les algorithmes hybrides s'appuient sur une représentation duale (PPC/PL) d'un problème et sur un algorithme unique de recherche basé sur l'un ou l'autre de leurs algorithmes respectifs. Ils supposent donc d'être capables de décomposer un problème PPC en problème ou sous-problèmes linéaire(s). Une fois la représentation duale obtenue, les algorithmes hybrides nécessitent d'identifier les interactions des deux représentations. Afin de définir finement cette coopération, des langages spécifiques tel que SCIP [2] permettent de décrire une méthode de recherche dédiée au problème modélisé. Pour un état de l'art détaillé, le lecteur est invité à consulter le livre [132] résumant les quinze dernière années de recherche dans le domaine.

Ce chapitre n'a pas pour ambition d'apporter de nouvelles solutions à ce thème actif de recherche opérationnelle. Il s'agit plutôt de décrire une implémentation fonctionnelle et de faire apparaître les intérêts d'une approche hybride (PPC/PL) ainsi que les défis issus du compromis entre la richesse d'information donnée par la programmation linéaire et la complexité à l'obtenir. L'approche proposée consiste à formaliser une contrainte globale polyédrique et à comparer différentes techniques d'implémentation. Nous proposons dans la section 5.3 une solution d'intégration triviale qui décompose un domaine polyédrique en un ensemble de contraintes usuelles et de variables intermédiaires. Dans la section 5.4, nous étudions comment combiner une bibliothèque de calcul polyédrique performante avec un solveur de programmation par contraintes. Les expériences de la section 5.5 montrent que l'utilisation de cette bibliothèque permet de réduire le nombre de mauvaises décisions dans l'espace de recherche. Cependant, les algorithmes de propagation mis en oeuvre ont souvent un coût trop élevé pour en justifier l'utilisation par rapport à une décomposition.

5.2 Formalisation et restrictions d'une contrainte polyédrique

Les variables dans un problème de programmation par contraintes sont à domaines finis. Par définition, il n'est donc pas possible de résoudre des problèmes paramétriques (cf. définition 11). Il est important de noter que dans notre contexte cela ne sera pas génant puisque les domaines que nous manipuleront seront en fait domaines d'ordonnancements qui, par construction, ne sont pas paramétrés. Dans ce contexte non paramétré, il est possible de formuler une contrainte (définition 18) dont le respect implique que les valeurs des variables d'un espace multidimensionnel forment un point se trouvant dans le domaine polyédrique associé.

Définition 18 (Contrainte polyédrique) *Soit* $D = \{\vec{x} \in \mathbb{Z}^m | \bigcup_{k=1}^{n} A_k\vec{x} + \vec{b}_k \geq \vec{0}\}$ *un domaine polyédrique non paramètré, une contrainte polyédrique sera satisfaite si et seulement si* $\vec{x}_{dom}$ *le vecteur des domaines finis associés aux dimensions* $\vec{x}$ *se trouve dans* D*.*

Par la suite, une contrainte polyédrique pour un domaine D est notée :

$$PolyCP(D)$$

La majorité des solveurs de programmation par contraintes disposent de contraintes optimisées pour travailler sur des domaines de variables dans $\mathbb{Z}$. Par conséquent, les domaines polyédriques manipulés sont des unions de $\mathbb{Z}$-polyèdres.

5.3 Contrainte polyédrique décomposée

Décomposer une contrainte consiste à exprimer la sémantique d'une contrainte spécifique sous forme d'un ensemble de contraintes usuelles. Il s'agit uniquement d'une facilité de modélisation puisque dans ce cas aucune technique tirant parti de cette spécificité ne filtre l'espace de recherche. Dans le cas d'une contrainte polyédrique, il est également nécessaire d'introduire de nouvelles variables pour traiter notamment les domaines polyédriques non convexes.

5.3.1 Décomposition d'une contrainte affine

Tous les solveurs de programmation par contraintes permettent d'exprimer des contraintes arithmétiques couvrant notamment le spectre des contraintes linéaires. Une solution simple pour représenter une contrainte affine c_k consiste à décomposer l'ensemble de ses termes en une somme pondérée (contrainte usuelle) dont le résultat est stocké dans une variable sum_{c_k}.

Contrainte 16 (Somme des termes affines) *Soit une contrainte affine* $c_k : A\vec{x}+b \geq 0$*, la somme* sum_{c_k} *des termes linéaires et de la partie constante est contrainte par :*

$$sum_{c_k} = SumWeight\left(\begin{bmatrix} A \\ b \end{bmatrix}, \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}\right)$$

Pour une solution valide, une contrainte affine n'est pas nécessairement satisfaite si celle-ci se trouve dans un domaine polyédrique non convexe. Dans l'exemple de la figure 5.1, la contrainte $j \leq 5 - i$ n'est pas satisfaite pour le point $(6, 1)$ qui se trouve pourtant dans le domaine $D = P_1 \cup P_2$.

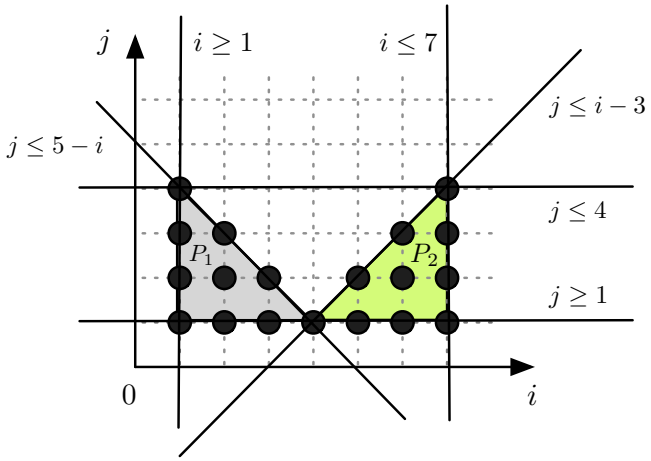


FIGURE 5.1 – Exemple de domaine non convexe

Pour traiter ce problème de convexité, il est nécessaire d'affecter une variable booléenne sat_{c_k} à chaque contrainte c_k. Si cette variable vaut 1 alors la contrainte est satisfaite et la somme de ses termes est supérieure ou égale à 0. Réciproquement, si sum_{c_k} est strictement inférieure à 0 alors sat_{c_k} vaut 0.

Contrainte 17 (Satisfaction d'une contrainte affine) *Soit une contrainte linéaire c_k et sum_{c_k} la somme de ses termes. Une variable booléenne sat_{c_k} est liée par réification à la satisfaction de c_k :*

$$sat_{c_k} \Leftrightarrow sum_{c_k} \geq 0$$

Une contrainte affine peut donc être décomposée par deux contraintes usuelles (somme pondérée et contrainte réifiée) en introduisant deux nouvelles variables dont une booléenne indiquant si la contrainte est satisfaite.

5.3.2 Décomposition d'un polyèdre convexe

Un polyèdre est défini par l'intersection d'un ensemble fini de demi-espaces (définition 10) et peut-être représenté sous forme d'une liste de contraintes affines. Un polyèdre convexe peut donc être modélisé en programmation par contraintes par l'ensemble des décompositions de ses contraintes affines. Cependant, puisqu'un polyèdre contenu dans un domaine non convexe n'est pas nécessairement satisfait, il ne faut pas imposer que toutes ses contraintes affines le soient. Un polyèdre ne sera satisfait que si toutes ses contraintes affines le sont. Il est possible de contraindre la satisfaction d'un polyèdre (contrainte 18) par une contrainte de conjonction logique et l'ensemble des contraintes issues de la décomposition de ses contraintes affines.

Contrainte 18 (Satisfaction d'un polyèdre) *La satisfaction d'un polyèdre P_k est associée à une variable booléenne sat_{p_k} valant 1 si l'ensemble de ses n contraintes affines décomposées sont satisfaites :*

$$sat_{p_k} = sat_{c_1} \wedge sat_{c_2} \wedge ... \wedge sat_{c_n}$$

Pour un polyèdre P_k constitué de n contraintes affines, le nombre de variables (V_{P_k}) et le nombre de contraintes (C_{P_k}) introduites par la décomposition sont :

$$\begin{cases} V_{P_k} & = & n+1 \\ C_{P_k} & = & 2*n+1 \end{cases}$$

5.3.3 Décomposition d'un domaine polyédrique

Une contrainte polyédrique impose qu'une solution valide au problème corresponde à un point se trouvant dans l'espace des points d'un domaine polyédrique. Si le domaine est constitué de plusieurs polyèdres, alors il est peut-être non convexe. La figure 5.1 fait apparaître deux polyèdres contenus dans un domaine non convexe. Une solution valide se trouve alors dans P_1, dans P_2 ou encore dans les deux polyèdres. Par conséquent, un domaine polyédrique est satisfait (définition 19) si la somme des variables de satisfaction de ses polyèdres est strictement positive.

Contrainte 19 (Satisfaction d'un domaine polyédrique) *Un domaine polyédrique P_k est satisfait si au moins un de ses polyèdres est satisfait :*

$$\sum_{k=1}^{n} sat_{P_k} > 0$$

Pour un domaine polyédrique D composé de n polyèdres, le nombre de variables (V_D) et le nombre de contraintes (C_D) introduites par la décomposition sont :

$$\begin{cases} V_D & = & \sum_{k=1}^{n}(V_{P_k}) \\ C_D & = & (\sum_{k=1}^{n} C_{P_k}) + 1 \end{cases}$$

Remarque : Si le domaine est constitué d'un unique polyèdre convexe, alors les contraintes de ce polyèdre doivent toutes être satisfaites. Les variables booléennes et contraintes réifiées associées à ce polyèdre ne sont donc pas nécessaires.

5.4 Contrainte polyédrique spécifique

Les techniques de propagation utilisées dans le cadre d'une contrainte décomposée ne profitent pas des informations géométriques des polyèdres. Celles-ci permettent pourtant de filtrer parfois davantage les valeurs possibles des dimensions. Exploiter une bibliothèque polyédrique permet d'extraire ces informations dans l'optique de réduire l'espace de recherche, quand cela est possible. Réciproquement, la satisfaction de contraintes non affines dans le problème réduit également les domaines des variables. Ceci implique donc de couper les domaines polyédriques par de nouvelles contraintes affines issues de ces réductions. Il est alors possible d'utiliser une bibliothèque de calcul polyédrique pour des problèmes d'optimisation non linéaires. La cohérence est assurée par une technique de propagation spécifique associée à l'algorithme *Branch & Bound* utilisé dans un solveur de programmation par contraintes.

5.4.1 Contrainte hybride

La programmation linéaire borne les domaines des variables plus précisément que les consistances d'arcs des contraintes CSP. Cependant, le coût de ces algorithmes de consistance est souvent plus

faible que de recourir à un solveur LP. Ainsi, certaines approches existantes [132] maintiennent une représentation LP et CP du problème pour bénéficier de leurs avantages respectifs. De manière générale, le recours à une représentation redondante des contraintes peut être une bonne pratique de la programmation par contraintes [132]. Les différents algorithmes de propagation filtrent plus efficacement les domaines des variables et accélèrent la recherche d'une solution sous réserve que le coût issu de cette redondance reste raisonnable. Pour tirer avantage de ces caractéristiques, la contrainte polyédrique mise en œuvre dispose d'un algorithme de consistance spécifique mais également d'une décomposition (contraintes 16, 17, 18 et 19) en contraintes standard. Ces contraintes disposent de techniques de propagation rapides dont le manque de précision est comblé par le recours à la programmation linéaire entière.

Algorithme 2 Algorithme de consistance de la contrainte polyédrique hybride

```
- - vars : ensemble des variables dont les domaines ont été modifiés
- - store : conteneur de l'état courant des variables
- - K : nombre minimal de variables
- - D : domaine polyédrique courant
- - BD : domaine polyédrique correspondant aux bornes des variables modifiées
CONSISTANCE (vars, store, K)
1   si vars.taille ≥ K alors
2      D ← DOMAINE(store)
3      BD ← BORNES(vars, store)
4      D ← BD ∩ D
5      si D ≠ ∅ alors
6         retourner PROPAGATION(D, store)
7      sinon
8         retourner faux
9      fin si
10  sinon
11     retourner vrai
12  fin si
13
```

L'algorithme 2 lie le solveur CSP avec une bibliothèque de calcul polyédrique. La fonction retourne $Vrai$ si la contrainte est consistante avec l'état courant du problème et $faux$ si le domaine polyédrique est incohérent avec les valeurs des variables ou les autres contraintes du problème.

De même que dans [72], elle utilise un paramètre K contrôlant la fréquence d'utilisation de la bibliothèque polyédrique. À chaque fois que le domaine d'une variable de dimension est modifié, celle-ci est ajoutée à la liste des prochaines variables à analyser ($vars$). Tant que la taille de cette liste est inférieure à K, l'algorithme de propagation spécifique ne sera pas appliqué. Ainsi, pour obtenir un comportement basé uniquement sur les contraintes usuelles issues de la décomposition, il suffit d'utiliser un K strictement supérieur au nombre de dimensions du domaine polyédrique de la contrainte. La première étape de l'algorithme consiste à récupérer D, le domaine polyédrique associé à l'état courant du conteneur de variables ($store$). Initialement, il correspond au domaine fourni en paramètre de la contrainte. Ce domaine est ensuite mis à jour à partir des nouvelles informations sur les bornes des variables en réalisant l'intersection de D et de BD. Si le domaine obtenu est vide alors l'information venant de la bibliothèque polyédrique permet d'affirmer que le problème est dans un état inconsistant et nécessite d'évaluer d'autres branches de l'arbre de recherche. Dans le cas contraire, une propagation de D sur l'état courant des domaines des variables est nécessaire pour tirer parti d'une

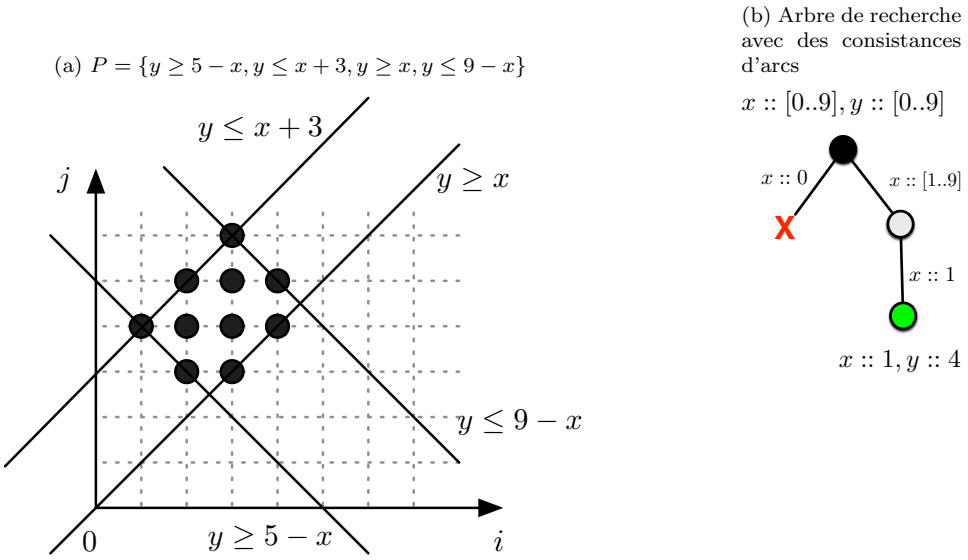


FIGURE 5.2 – Recherche d'un point d'un polyèdre (contraintes usuelles)

éventuelle réduction de leurs bornes. Cette propagation peut amener également à un état inconsistant si une des autres contraintes du problème est violée par les bornes issues de D.

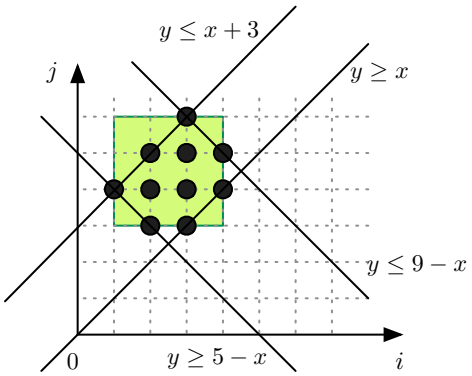
Lorsque le problème est dans un état inconsistant, la dernière variable évaluée dans l'arbre de recherche est la source de la violation des contraintes. Il est donc nécessaire d'analyser l'autre branche du nœud correspondant à la dernière évaluation. Si les deux branches amènent à un état inconsistant, l'algorithme de recherche remonte dans l'arbre jusqu'à trouver une branche inexplorée. Pour éviter de créer un domaine polyédrique à partir de l'état courant de toutes les variables pour chaque niveau de l'arbre de recherche, on construit de manière incrémentale une hiérarchie de domaines polyédriques. D est le domaine courant de cette hiérarchie. Si une inconsistance est détectée, alors les domaines dont la profondeur est supérieure ou égale au niveau à restaurer sont supprimés de la hiérarchie.

5.4.2 Algorithmes de propagation

La figure 5.2 montre un cas où les consistances d'arcs des contraintes linéaires ne permettent pas de réduire efficacement l'espace de recherche. Après la consistance initiale, les domaines des variables x et y ne peuvent pas être réduits plus que $[0..9]$ et la branche où $x = 0$ est évaluée. Cette dernière viole les contraintes de P et deux décisions sont alors nécessaires pour identifier une solution valide. Sur la figure, il est pourtant évident que x et y sont en fait bornés par $B = \{4 \geq x \geq 1, 6 \geq y \geq 3\}$. Ce paragraphe propose deux algorithmes de propagation utilisant ces informations géométriques pour mettre à jour les domaines des variables et réduire l'espace de recherche.

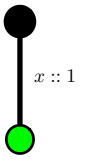
Boîte englobante convexe La boîte englobante convexe d'un polyèdre peut être obtenue en isolant chaque dimension par des projections successives. Dans la figure 5.3, B est la boîte englobante de P. Les bornes de B sont utilisées pour filtrer les domaines de x et y. Ces domaines filtrés réduisent la recherche d'une solution à une unique décision.

(a) boîte englobante de P, $B = \{4 \geq x \geq 1, 6 \geq y \geq 3\}$



(b) Arbre de recherche avec consistance de boîte englobante

$x :: [1..4], y :: [3..6]$



$x :: 1, y :: 4$

FIGURE 5.3 – Recherche d'un point d'un polyèdre (propagation par boîte englobante)

Algorithme 3 Propagation par boîte englobante convexe

– D : domaine polyédrique analysé
– *store* : conteneur de l'état courant des variables
– *hull* : enveloppe convexe de D
– *bornes* : polyédre correspondant aux bornes d'une dimension de *hull*
– *min* : borne basse d'une variable
– *max* : borne haute d'une variable
– *consistant* : booléen valant $Vrai$ si l'application des bornes d'une variable est consistante

```
PROPAGATION (D, store)
1   hull ← ENVELOPPE_CONVEXE(D)
2   pour ∀dim ∈ x⃗_D faire
3     si RESTRICTION(DIM) alors
4       bornes ← ISOLER(dim, hull)
5       min ← BORNE_BASSE(dim, bornes)
6       max ← BORNE_HAUTE(dim, bornes)
7       consistant ← FILTRER(dim, min, max)
8       si ¬consistant alors
9         retourner faux
10      fin si
11    fin si
12  fin pour
13  retourner vrai
```

L'algorithme 3 décrit comment propager les informations de la boîte englobante sur les domaines des dimensions. Si le domaine polyédrique n'est pas convexe, il est nécessaire d'obtenir l'enveloppe convexe du domaine polyédrique. Il s'agit d'une opération fournie par la bibliothèque de calcul polyédrique. Construire la boîte englobante d'un polyèdre consiste alors à isoler chaque dimension. L'isolation d'une dimension est une opération coûteuse qui nécessite de projeter toutes les autres dimensions du polyèdre. Or, dans beaucoup de cas, cette isolation n'est pas nécessaire car les bornes obtenues sont redondantes avec l'état courant du domaine de la dimension. Pour cette raison, on teste tout d'abord si la boîte englobante apporte de nouvelles informations (ligne 3). Si une dimension implique une restriction, les bornes hautes et basses obtenues par isolation (ligne 4) sont propagées sur son

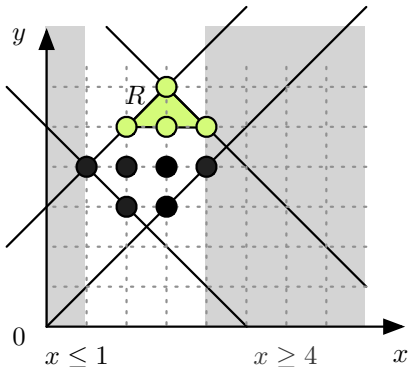


FIGURE 5.4 – Exemple de dimension restrictive. R est le domaine courant après une coupe de D par $y \geq 5$. Puisque $R \cap \{x \leq 1\} = \varnothing$, x nécessite une restriction de son domaine de variable.

domaine de variable (ligne 7). Dans le cas où cette propagation est consistante, la dimension suivante est analysée. Dans le cas contraire, la contrainte n'est pas satisfaite et le problème est inconsistant.

Pour tester efficacement si une dimension introduit une restriction, il suffit de vérifier que le domaine courant n'intersecte pas l'un des polyèdres correspondant à la borne haute ou à la borne basse de l'état courant de la dimension. La figure 5.4 illustre un exemple de dimension restrictive. Le domaine de la variable x est [1..4] et le domaine de y a été réduit à [5..6] par des contraintes externes. En appliquant la contrainte de coupe $y \geq 5$ au domaine précédent D, on obtient un polyèdre R qui n'intersecte pas le polyèdre $x_{lb} = \{x \leq 1\}$ issu de la borne basse connue de x. Le polyèdre R est donc plus petit que D et la dimension x est dite restrictive. Cette technique permet d'obtenir un gain de l'ordre de 50% sur les temps d'exécution des propagations mesurés dans les résultats expérimentaux (cf. section 5.5).

Bornes syntaxiques L'analyse des dimensions restrictives permet de limiter le nombre de projections aux dimensions qui définissent une boîte englobante utile. Cependant, cette analyse constitue également une partie critique de l'algorithme de propagation. Une autre approche consiste à se baser tout simplement sur les bornes qui apparaissent syntaxiquement dans l'enveloppe convexe du domaine polyédrique courant (algorithme 4). L'information est évidemment beaucoup moins précise que celle issue de la boîte englobante mais elle a le mérite d'être très simple à détecter. Elle constitue un complément performant pour couper des branches non consistantes dans l'espace de recherche.

Algorithme 4 Propagation par bornes syntaxiques

```
PROPAGATION (D, store)
1   hull ← ENVELOPPE_CONVEXE(D)
2   pour ∀dim ∈ x⃗_D faire
3      min ← BORNE_BASSE(dim, hull)
4      max ← BORNE_HAUTE(dim, hull)
5      consistant ← FILTRER(dim, min, max)
6      si ¬consistant alors
7         retourner faux
8      fin si
9   fin pour
10  retourner vrai
```

5.5 Analyse empirique de la complexité

La contrainte hybride et la contrainte décomposée on été implémentées pour le solveur Jacop [177]. Les techniques de consistance spécifiques utilisent ISL [1] [96]. Les mesures des temps de résolution ont été réalisées sur un Intel core 2 duo à 2,8 Ghz.

Le problème du carré magique consiste à choisir les valeurs d'une grille de telle manière que les sommes de chaque ligne, colonne et diagonale du carré soient égales. De plus, les valeurs des cases doivent être différentes. Ce problème est composé d'un ensemble de contraintes linéaires exprimant les égalités lignes/colonnes/diagonales et d'une contrainte globale $AllDifferent$ difficilement linéarisable. Des contraintes linéaires supplémentaires permettent de supprimer de l'espace de recherche les solutions symétriques. La figure 5.5 détaille l'ensemble des contraintes associées à un carré magique de taille trois. Toutes les contraintes linéaires peuvent être modélisées dans une contrainte polyédrique dont les dimensions correspondent aux variables des cases de la grille. L'utilisation d'une contrainte hybride permet de trouver une solution en cinq décisions au lieu de sept (cf. figure 5.6).

Lignes
$$\begin{cases} c_{11}+c_{12}+c_{13} = c_{21}+c_{22}+c_{23} \\ c_{11}+c_{12}+c_{13} = c_{31}+c_{32}+c_{33} \end{cases}$$

Colonnes
$$\begin{cases} c_{11}+c_{12}+c_{13} = c_{11}+c_{21}+c_{31} \\ c_{11}+c_{12}+c_{13} = c_{12}+c_{22}+c_{32} \\ c_{11}+c_{12}+c_{13} = c_{13}+c_{23}+c_{33} \end{cases}$$

Diagonales
$$\begin{cases} c_{11}+c_{12}+c_{13} = c_{11}+c_{22}+c_{33} \\ c_{11}+c_{12}+c_{13} = c_{13}+c_{22}+c_{31} \end{cases}$$

c_{11}	c_{12}	c_{13}
c_{21}	c_{22}	c_{23}
c_{31}	c_{32}	c_{33}

Symetries
$$\begin{cases} c_{11} < c_{13} \\ c_{11} < c_{31} \\ c_{11} < c_{33} \end{cases}$$

$$\begin{cases} 1 \leq c_{11}, c_{12}, c_{13}, c_{21}, c_{22}, c_{23}, c_{31}, c_{32}, c_{33} \leq 9 \\ AllDifferent(c_{11}, c_{12}, c_{13}, c_{21}, c_{22}, c_{23}, c_{31}, c_{32}, c_{33}) \end{cases}$$

FIGURE 5.5 – Contraintes du carré magique de taille 3.

Un des paramètres importants lors de la résolution d'un problème de programmation par contraintes est de choisir l'ordre de sélection des variables évaluées. Dans la recherche associée à une contrainte décomposée, le meilleur ordre de variables est c_{11} puis c_{13} alors que pour une contrainte hybride il s'agit de c_{11} puis c_{23}. Cet ordre peut dépendre uniquement du problème posé, on parle alors d'ordre statique. Par exemple, il est souvent pertinent d'évaluer en premier lieu les variables qui sont impliquées dans le plus de contraintes puisque leur évaluation aura le plus de chance de filtrer l'espace de recherche. Une autre solution est de tenir compte de l'état courant du problème en choisissant par exemple les variables qui ont le plus petit domaine au moment de la sélection. Il existe de nombreux algorithmes de sélection de variables et choisir le plus adapté à un problème donné peut être difficile. Il s'avère que ce choix est critique dans la résolution du carré magique décrit à l'aide d'une contrainte polyédrique hybride.

Dans les résultats expérimentaux suivants, on s'intéresse au carré magique de taille quatre et l'on compare les combinaisons des deux techniques de consistances proposées à différents ordres de variables. Ces ordres sont issus de sélecteurs identifiés au cours des expérimentations comme étant les plus performants pour ce problème. Le sélecteur MS [2] choisit les variables qui sont liées au plus

1. Integer Set Library
2. MostConstrained Static

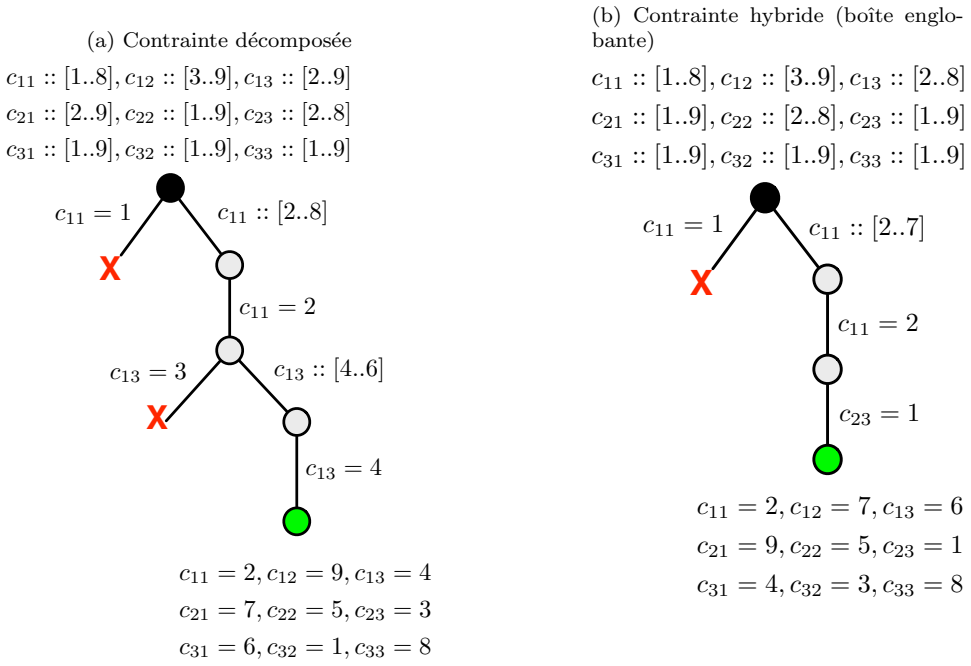


FIGURE 5.6 – Comparaison des arbres de recherche pour un carré magique de taille 3.

de contraintes. Le sélecteur MD [3] utilise une priorité basée sur la valeur minimale d'un domaine divisée par le nombre courant de contraintes associées à cette variable. Le sélecteur $MR + MD$ [4] donne la priorité aux variables dont la plus petite valeur est la plus éloignée de la prochaine puis, en cas d'égalité entre deux variables, fait appel au sélecteur MD.

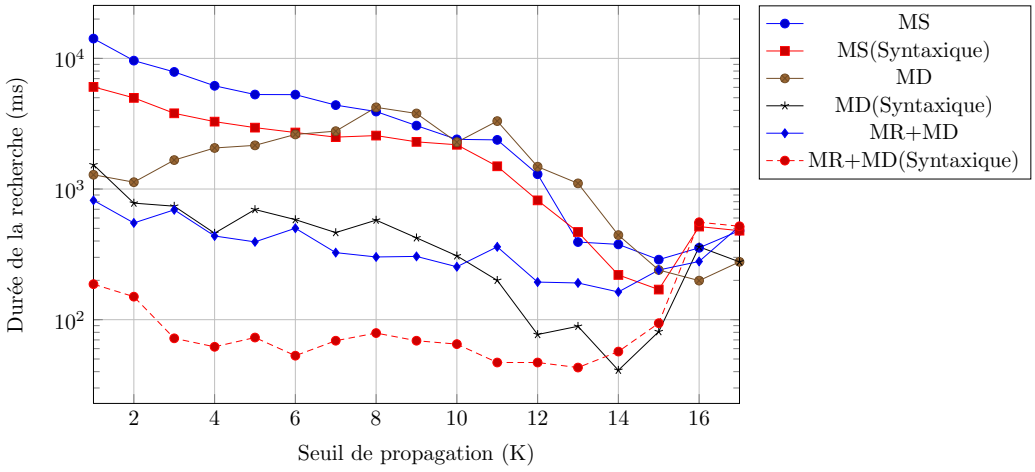


FIGURE 5.7 – Temps de résolution pour le carré magique de taille 4 (échelle semi-logarithmique). Pour $K = 17$ il n'y a aucune consistance spécifique.

Les résultats de la figure 5.7 font apparaitre une importante variation du temps de résolution en fonction du couple sélecteur/consistance utilisé. Pour chacune des courbes, le point où $K = 17$

3. MinDomain over Degree
4. Max Regret + MinDomain over Degree

représente le temps de référence où la consistance spécifique n'est pas utilisée (décomposition pure) puisque la grille comporte seize cases. La meilleure combinaison correspond à l'utilisation du sélecteur $MR + MD$ associé à une contrainte hybride basée sur les bornes syntaxiques. Quelle que soit la fréquence d'application de la consistance, le temps de résolution est dans la plupart des cas au moins deux fois inférieur au temps de référence. Pour $K = 13$, une solution est trouvée en 43 ms au lieu des 277 ms pour la recherche la plus rapide (tout sélecteur confondu) sans consistance polyédrique. Il apparaît également qu'utiliser les algorithmes de boîte englobante et de bornes syntaxiques peut s'avérer être beaucoup trop coûteux si le sélecteur est mal choisi et que leurs fréquences d'application sont trop élevées. C'est notamment le cas du sélecteur MS qui ralentit énormément la recherche d'une solution, quelle que soit la consistance spécifique utilisée, jusqu'à un certain seuil de propagation ($K = 13$) où le temps de résolution devient légèrement inférieur au temps de référence.

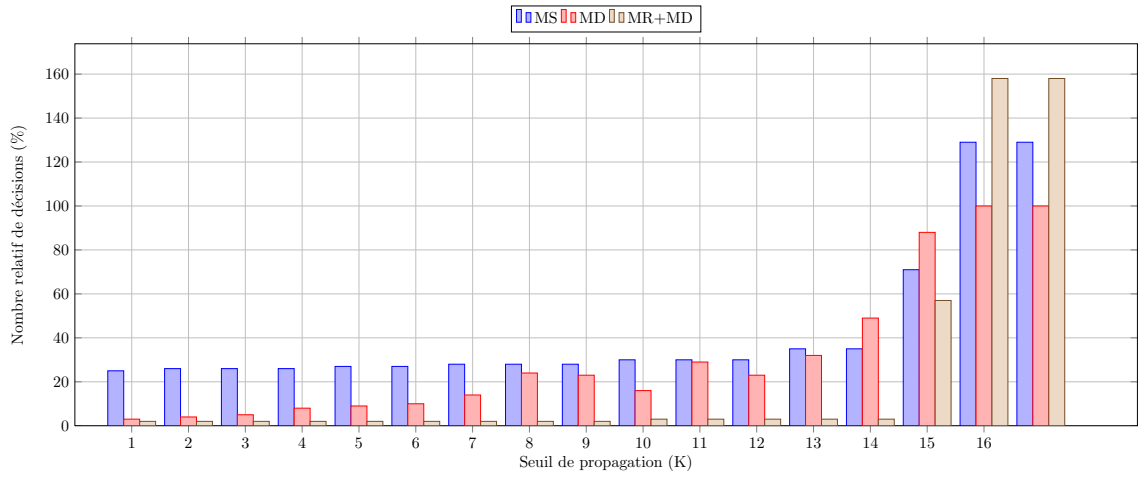


FIGURE 5.8 – Nombre de décisions relatif au plus faible nombre de décisions sans contrainte spécifique (boîte englobante). Pour $K = 17$ il n'y a aucune consistance spécifique.

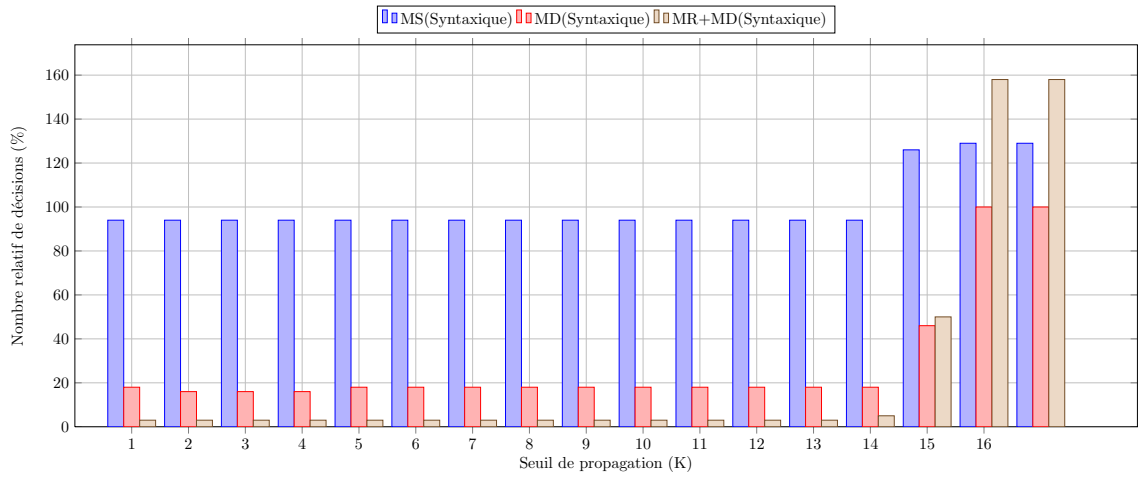


FIGURE 5.9 – Nombre de décisions rélatif au plus faible nombre de décisions sans contrainte spécifique (Bornes syntaxiques). Pour $K = 17$ il n'y a aucune consistance spécifique.

Il est important de noter que l'utilisation de techniques de consistance spécifiques élague considérablement l'espace de recherche. Les figures 5.8 et 5.9 révelent que pour les deux techniques de

consistances et $K \leq 14$, le sélecteur $MR + MD$ élimine au minimum 97% des décisions de cet espace. Le nombre de décisions de référence correspond au nombre minimum de décisions requises (sélecteur MD) en utilisant uniquement une contrainte décomposée. L'algorithme basé sur les bornes syntaxiques n'élague quasiment pas l'espace de recherche si le sélecteur utilisé est MS. Ce n'est pas le cas de la consistance par boîte englobante qui permet de gagner 80% de décisions mais au prix d'une durée de résolution supérieure d'un facteur cent.

Chapitre 6

Espace conjoint de spécialisation et d'optimisation de code

Dans le chapitre 3, nous avons présenté une méthodologie d'extension de jeu d'instructions applicable à un graphe flot de données. Des instructions spécialisées y sont identifiées, sélectionnées et ordonnancées pour une architecture donnée dans un problème d'optimisation basé sur la programmation par contraintes.

Dans ce chapitre, nous proposons une approche plus générale qui aborde la problématique de l'optimisation de nids de boucles dans l'optique d'y faire apparaître de nouvelles opportunités d'optimisation exploitables par l'extension du jeu d'instructions d'un processeur ASIP. Cette approche tire parti de l'expressivité du modèle polyédrique pour sélectionner des instructions spécialisées favorisant la localité des données et le parallélisme des calculs déportés sur l'extension matérielle.

Sommaire

6.1 Introduction

Les méthodologies automatisées d'extension de jeu d'instructions ne se basent généralement que sur le flot de données d'un bloc de base d'un programme. Comme nous l'avons montré dans le chapitre 1, l'objectif de ces approches est d'identifier et de sélectionner des regroupements d'opérateurs (i.e., motifs). Ceux-ci seront déportés sur une extension matérielle pour accélérer l'exécution d'une famille d'application.

De nombreuses informations sont néanmoins ignorées par de telles approches, les nids de boucles constituent en effet souvent les parties critiques d'un programme. L'analyse précise de leurs comportements permet notamment d'évaluer le degré de parallélisme des corps de boucles et la durée de vie des données qui y sont produites. Ces informations sont pourtant très pertinentes dans le cadre d'un processeur couplé à une extension matérielle. Ainsi, une donnée produite sur l'extension, lors d'une itération d'un corps de boucle, n'est peut être qu'un résultat intermédiaire utilisé par une autre itération du même corps de boucle. Le rapatriement de cette donnée sur le processeur (dans la mémoire ou la file de registres) n'est donc pas nécessaire si elle est mémorisée dans une mémoire embarquée sur l'extension. D'autre part, si un groupement d'opérateurs se trouve dans un corps de boucles dont la dimension la plus interne est parallèle, il est alors envisageable de concevoir une instruction SIMD [1](e.g., MMX, SSE1-4 d'Intel, 3DNow ! d'AMD ou NEON de ARM) qui exécutera simultanément la même opération sur un vecteur de données.

De nombreux travaux sur les ASIP montrent le gain de performance important obtenu en utilisant des instructions spécialisées SIMD (e.g., [165, 184, 110, 170]). Pourtant, il n'existe pas, à notre connaissance, d'approche permettant de transformer automatiquement les nids de boucles afin d'y identifier de *nouvelles* instructions spécialisées vectorisables. De même, si la fusion de boucles permet de détecter des instructions spécialisées de taille plus importante, cette transformation est effectuée en amont de la sélection des motifs exécutés sur l'extension. De manière générale, dans un flot d'extension de jeux d'instructions, les transformations de code guidant la sélection des motifs requièrent l'intervention explicite du concepteur ou encore celle d'un « oracle » prédisant l'intérêt et le choix des transformations.

Dans ce chapitre, nous proposons une approche originale qui s'appuie à la fois sur la transformation de boucles basée sur le modèle polyédrique (cf. chapitre 4) et sur la programmation par contraintes (cf. chapitre 2). L'objectif est d'exploiter l'expressivité des ordonnancements affines (cf. définition 13) pour sélectionner des instructions spécialisées se trouvant dans des nids de boucles dont la dimension la plus interne est parallèle. Les instructions sélectionnées seront donc potentiellement vectorisables (on ne tient pas compte, pour l'instant, des problèmes d'alignements et de contiguïté mémoire). De plus, afin de réduire la pression sur la mémoire du processeur, on souhaite également pouvoir déporter sur l'extension matérielle la mémorisation de certains résultats intermédiaires qui y ont été produits.

Le principe général de notre algorithme est de formuler un problème de couverture de graphe (cf. sous-section 3.3.1, page 57) sur une représentation fine du graphe des dépendances de données polyédrique (PRDG). Une approche naïve consisterait à assimiler une occurrence de motif à une nouvelle instruction spécialisée qui, si elle est sélectionnée, sera ordonnancée afin de respecter les contraintes susmentionnées. Cependant, dans un PRDG, deux nœuds qui sont connectés par une dépendance de données ne sont pas nécessairement définis dans le même espace d'itération, un regroupement de

1. Single Instruction Multiple Data

nœuds nécessite alors une première étape d'extraction de leur domaine de définition commun. Cette étape implique que la sélection d'instructions spécialisées ne pourra être faite que par un algorithme glouton itératif à l'efficacité critiquable. En effet la sélection d'une instruction spécialisée, lors d'une itération de l'algorithme, risque d'empêcher d'en détecter de nouvelles qui respecteront les contraintes spécifiques énoncées précédemment.

Si les ordonnancements affines structurés [57, 157] ont été conçus dans une optique de passage à l'échelle. Ils offrent néanmoins une réponse adaptée à notre problème de transformation de boucles et de sélection d'instructions spécialisées. En effet, la notion d'occurrence de motif de calcul dans un PRDG est intuitivement très proche de celle d'un processus dans l'algorithme de Feautrier [57] ou de celle d'un sous-système dans celui de Quinton et. al. [157] : les éléments ordonnancés peuvent être vus comme des instances de sous-PRDG liés par des dépendances de données interconnectant leurs ports (entrée ou sortie) respectifs.

La modularité des ordonnancements structurés nous permet d'énoncer, pour chaque occurrence de motif candidate, un ensemble de contraintes modulaires portant sur la légalité de son ordonnancement, sur des exigences de performance (e.g., vectorisation) ou encore sur le respect de considérations architecturales (e.g., durée de vie des données mémorisées sur l'extension). Nous parlons d'approche « conjointe » de spécialisation et d'optimisation de code, car la sélection d'instructions spécialisées dans le PRDG conditionnera les contraintes d'ordonnancement : si une occurrence de motif n'est pas sélectionnée, ses contraintes modulaires n'ont pas lieu d'être respectées.

Nous modélisons ici le problème comme un CSP contenant les contraintes issues de la couverture (cf. sous-section 3.3.1) ainsi qu'une contrainte polyédrique (cf. chapitre 5) pour les contraintes modulaires de chaque occurrence de motif candidate. La solution du CSP correspond alors à la fois à une couverture du PRDG et à un ordonnancement affine des ports de chaque occurrence de motif sélectionnée. Les coefficients de cet ordonnancement global sont ensuite injectés dans les contraintes modulaires des occurrences sélectionnées. Les contraintes ainsi obtenues expriment, pour chaque occurrence, le respect de l'ordonnancement global de la couverture et les conditions de légalité pour l'ordonnancement de ses nœuds internes. Il est alors possible de considérer les occurrences comme autant de PRDG à ordonnancer indépendamment les uns des autres (e.g., avec l'algorithme de Feautrier [56] ou encore PLUTO [25]). Le résultat final est un ensemble d'occurrences de motifs dont l'exécution est déportée sur l'extension matérielle. Ces occurrences contiendront une ou plusieurs instructions spécialisées associées à une fonction d'ordonnancement affine qui respectera les contraintes spécifiques imposées lors de la couverture (e.g., vectorisation).

Les contributions de ce chapitre sont les suivantes.

1. La possibilité de sélectionner des instructions spécialisées se trouvant à l'origine dans différentes boucles et sans nécessiter une préalable transformation du code.
2. La sélection automatique d'instructions spécialisées vectorisables par un algorithme conjoint de transformation de boucles et de couverture de graphe.
3. La possibilité d'utiliser des contraintes non linéaires (programmation par contraintes) dans un problème d'ordonnancement affine.
4. La généralisation des ordonnancements structurés de Feautrier aux cas multidimensionnels.

Le chapitre est organisé comme suit. La première section détaille l'ordonnancement affine modulaire de Feautrier et le généralise au cas multidimensionnel La deuxième section montre que les

ordonnancements modulaires permettent de transformer le code original pour y faire apparaître de nouvelles instructions spécialisées vectorisables. La troisième section présente l'algorithme utilisé. Celui-ci mêle les transformations affines à des contraintes non linéaires de couverture sur le PRDG. Enfin, les différentes étapes de l'algorithme sont illustrées par un exemple complet.

6.2 Ordonnancement modulaire

6.2.1 L'algorithme original de Feautrier

Dans les paragraphes suivants, nous détaillons le principe de l'algorithme d'ordonnancement structuré de Feautrier [57]. Il constitue les fondements de notre approche et fait donc l'objet d'une attention particulière.

6.2.1.1 Explication de l'algorithme

L'algorithme s'inspire des réseaux de processus de Kahn pour améliorer le passage à l'échelle de l'algorithme d'ordonnancement affine monodimensionnel [55]. Le principe est de décomposer le PRDG d'une application en *processus* et d'énoncer, de manière *modulaire*, les contraintes de causalité d'un ordonnancement affine légal. Cette modularité permet une approche *diviser pour régner* qui décompose l'ordonnancement d'une application en deux étapes successives.

Pour cela, on considère qu'une application est structurée en processus indépendants qui communiquent entre eux par des *canaux*. Chaque processus est alors décrit par un PRDG disposant de ports d'entrée et de sortie pour consommer ou recevoir des données d'un autre processus. Les ports respectifs du producteur d'une donnée et d'un consommateur sont reliés par un canal de communication.

Un canal de communication A est assimilé à un tableau. Le producteur n'y écrit une donnée $A[x]$ qu'une seule fois et un consommateur peut y accéder plusieurs fois. À chaque canal A, on associe une fonction d'ordonnancement affine θ_A indiquant la date à partir de laquelle une donnée $A[x]$ est disponible dans ce canal :

$$\theta_A(x) = \mu_{A1}x + \mu_{A2} \tag{6.1}$$

Il est alors possible de garantir la modularité des processus, en exprimant la causalité d'une dépendance de communication par deux contraintes indépendantes :

- Soit $W : A[f_{WA}(\vec{i})] = \ldots$ un nœud produisant la donnée $A[f_{WA}(\vec{i})]$, le nœud W doit être ordonnancé avant l'écriture de la donnée dans le canal :

$$\forall \vec{i} \in \mathcal{D}_W, \theta_A(f_{WA}(\vec{i})) \geq \theta_W(\vec{i}) \tag{6.2}$$

- Soit $R : \ldots = A[f_{RA}(\vec{i})]$ un nœud consommant la donnée $A[f_{RA}(\vec{i})]$, le nœud R ne peut lire la donnée que si elle est disponible dans le canal :

$$\forall \vec{i} \in \mathcal{D}_R, \theta_A(f_{RA}(\vec{i})) + 1 \leq \theta_R(\vec{i}) \tag{6.3}$$

Les contraintes de communication d'un processus sont construites en appliquant la contrainte 6.2 à chaque canal de sortie ainsi que la contrainte 6.3 pour toutes les dépendances de chaque canal

```
1
2  process P(outport int X[]){
3    int i;
4    for(i=0;;i++)
5  W:  X[i] = f(i)
6  }
7
8  process Q(inport int Y[]){
9    int i,s;
10 Z: s=0;
11   for(i=0;;i++)
12 M:  s = s + Y[i]*Y[i+2]
13 }
14
15 void main(){
16   channel int A[];
17   P(A);
18   Q(A);
19 }
```

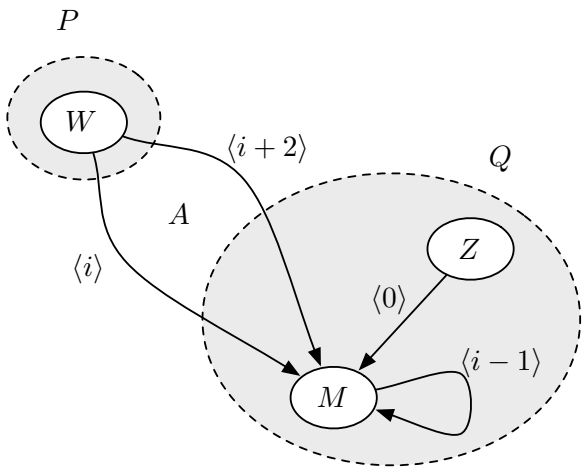


FIGURE 6.1 – Syntaxe CRP et PRDG d'un réseau de deux processus (P et Q) communiquant par un canal A.

d'entrée.

L'algorithme modulaire d'ordonnancement monodimensionnel de Feautrier est divisé en cinq étapes.

1. Construction des contraintes de légalité de chaque processus : respect de la causalité des dépendances internes et des canaux de communication extérieurs.
2. Élimination, par projection (e.g., par Fourrier-Motzkin), des coefficients d'ordonnancement des nœuds internes à chaque processus. L'ensemble des contraintes de légalité ne porte alors plus que sur les coefficients d'ordonnancement des canaux. Cette étape ainsi que la précédente sont indépendantes du programme et peuvent provenir d'une bibliothèque référençant les ensembles de contraintes associés à chaque motif de processus.
3. Ordonnancement des canaux de communication de l'ensemble des processus de l'application.
4. Injection du résultat de l'ordonnancement des communications dans les contraintes de chaque processus. Si nécessaire, les processus sont ordonnancés indépendamment pour identifier tous les coefficients d'ordonnancement internes.
5. Génération du code (e.g., avec CLOOG [18]) pour les nœuds ordonnancés de tous les processus de l'application.

6.2.1.2 Exemple

Afin de simplifier la compréhension de l'algorithme de Feautrier, nous déroulons ses cinq étapes pour l'exemple de la figure 6.1, présenté dans la publication originale. Dans cet exemple, un réseau de processus est décrit dans le langage dédié (CRP [2]) proposé par Feautrier, deux processus (P et

2. Communicating Regular Processes

Q) y communiquent par l'intermédiaire d'un canal A. Nous détaillons maintenant les contraintes de communication de chaque processus afin de faire apparaître leur aspect modulaire : chaque ensemble de contraintes peut être construit indépendamment, mais l'ordonnancement des processus doit être résolu conjointement.

À chaque nœud du PRDG issu du réseau de processus, on associe un prototype d'ordonnancement et un domaine de définition.

- $\theta_W(i) = \mu_{W1} i + \mu_{W2}$ avec $\mathcal{D}_W : i \geq 0$
- $\theta_Z(i) = \mu_{Z2}$ avec $\mathcal{D}_Z : i = 0$
- $\theta_M(i) = \mu_{M1} i + \mu_{M2}$ avec $\mathcal{D}_M : i \geq 0$

Étape 1 : Construction des contraintes de légalité

Cas du producteur (processus P) Ce processus ne contient aucune dépendance de données interne. Ses contraintes ne portent donc que sur l'écriture d'une donnée dans le canal A. Son domaine de définition $\mathcal{D}_{W->A}$ est :

$$\mathcal{D}_{W->A} : \{\, i \mid i \geq 0 \,\}$$

En appliquant la contrainte 6.2 à l'écriture de W dans le canal A on obtient :

$$i(\mu_{A1} - \mu_{W1}) + (\mu_{A2} - \mu_{W2}) \geq 0$$

La forme matricielle du domaine $\mathcal{D}_{W->A}$ de la dépendance illustre les multiplieurs de Farkas (un multiplieur par contrainte affine du domaine) :

$$\begin{array}{c} \\ i \geq 0 :: \lambda_1 \end{array} \begin{array}{c} \begin{array}{cc} i & 1 \end{array} \\ \begin{pmatrix} 1 & 0 \end{pmatrix} \end{array}$$

En utilisant la forme affine du lemme de Farkas, on en déduit que :

$$i(\mu_{A1} - \mu_{W1}) + (\mu_{A2} - \mu_{W2}) = \lambda_0 + i\lambda_1 \geq 0$$

$$\left\{ \begin{array}{lcll} \mu_{A1} - \mu_{W1} & = & \lambda_1 & \geq 0 \\ \mu_{A2} - \mu_{W2} & = & \lambda_0 & \geq 0 \end{array} \right. \tag{6.4}$$

Après avoir éliminé, par projection, les multiplieurs de Farkas, on obtient :

$$\left\{ \begin{array}{l} \mu_{A2} \geq \mu_{W2} \\ \mu_{A1} \geq \mu_{W1} \end{array} \right. \tag{6.5}$$

Cas du consommateur (processus Q) Le processus contient deux nœuds (Z et M) ainsi que deux dépendances de données internes.

La dépendance entre Z et M a pour domaine $\mathcal{D}_{Z->M} : \{\, i \mid i = 0 \,\}$, sa causalité est respectée si et seulement si :

$$i\mu_{M1} + (\mu_{M2} - \mu_{Z2} - 1) \geq 0$$

En appliquant la forme affine du lemme de Farkas et après avoir éliminé les multiplieurs, on en déduit que :

$$\mu_{M2} \geq 1 + \mu_{Z2} \tag{6.6}$$

D'autre part, la dépendance cyclique sur M et qui a pour domaine $\mathcal{D}_{M->M} : \{\, i \mid i \geq 1 \,\}$, est uniforme. On en déduit que $\mu_{M1} \geq 1$.

Le processus Q est associé à deux dépendances de communication. La première est une lecture de la donnée $A[i+2]$. Le respect de la contrainte 6.3 implique que :

$$\theta_M(i) - \theta_A(i+2) - 1 \geq 0$$

$$i(\mu_{M1} - \mu_{A1}) + (\mu_{M2} - 2\mu_{A1} - \mu_{A2} - 1) \geq 0$$

En appliquant la forme affine du lemme de Farkas et après avoir éliminé les multiplieurs de Farkas, on en déduit que :

$$\begin{cases} \mu_{M2} & \geq & 1 + 2\mu_{A1} + \mu_{A2} \\ \mu_{M1} & \geq & \mu_{A1} \end{cases} \tag{6.7}$$

En procédant de même pour la dépendance de communication correspondant à la lecture de $A[i]$ on obtient :

$$\begin{cases} \mu_{M2} & \geq & 1 + \mu_{A2} \\ \mu_{M1} & \geq & \mu_{A1} \end{cases} \tag{6.8}$$

Étape 2 : Élimination des coefficients d'ordonnancement internes Une fois les contraintes obtenues pour chaque processus, on élimine les coefficients internes de chaque ensemble de contraintes.

Cas du producteur (processus P) Les coefficients internes d'ordonnancement de P sont μ_{W1} et μ_{W2}, leur élimination de (6.5) forme l'ensemble des contraintes de légalité des communications de P. Dans ce cas, il ne reste aucune contrainte après projection.

Cas du consommateur (processus Q) L'ensemble des contraintes de légalité de Q construit à partir de (6.7) et (6.8) est :

$$\begin{cases} \mu_{M2} & \geq & 1 + \mu_{Z2} \\ \mu_{M1} & \geq & 1 \\ \mu_{M2} & \geq & 1 + 2\mu_{A1} + \mu_{A2} \\ \mu_{M1} & \geq & \mu_{A1} \\ \mu_{M2} & \geq & 1 + \mu_{A2} \end{cases} \tag{6.9}$$

De même que pour P, après l'élimination des coefficients internes de Q, il ne reste aucune contrainte.

Étape 3 : Ordonnancement des canaux Dans cet exemple, il n'y a aucune contrainte d'imposée sur les ordonnancements des canaux. La solution d'ordonnancement des canaux sélectionnée par Feautrier est $\mu_{A1} = \mu_{A2} = 0$.

Étape 4 : Ordonnancement des nœuds Une fois les coefficients d'ordonnancement des canaux déterminés, on injecte l'information dans les contraintes construites lors de la première étape pour

chaque processus.

Ainsi, pour P, on obtient l'ensemble de contraintes suivantes :

$$\begin{cases} 0 \geq \mu_{W2} \\ 0 \geq \mu_{W1} \end{cases} \tag{6.10}$$

L'ordonnancement de W est donc $\theta_W(i) = 0$.

De même, pour le processus Q, l'ensemble des contraintes devient :

$$\begin{cases} \mu_{M2} & \geq & 1 + \mu_{Z2} \\ \mu_{M1} & \geq & 1 \\ \mu_{M2} & \geq & 1 \end{cases} \tag{6.11}$$

Les ordonnancements sélectionnés pour M et Z sont $\theta_M(i) = i + 1$ et $\theta_Z = 0$.

On déduit de ces ordonnancements que l'intégralité des itérations du nœud W sont exécutées à la même date ($\theta_W(i) = 0$), toutes les données produites sont donc écrites simultanément dans le canal A. Ce constat amène à considérer des contraintes supplémentaires pour cibler une mise en œuvre concrète sur une mémoire dont la taille est limitée. L'approche de Feautrier sur ces contraintes mémoires fait l'objet de la sous-section suivante.

6.2.2 Contraintes mémoires

Le modèle de mémoire envisagé par Feautrier pour mettre en œuvre un canal est un *buffer* circulaire qui, dans le cas d'un tableau multidimensionnel $A[][]$, enregistre une donnée $A[x][y]$ au même moment que toutes les autres données du sous-tableau $A[x][*]$. Cette forte restriction lui permet d'énoncer un lien simple entre la date d'allocation $\alpha_A(x)$ des données $A[x][*]$ sur un canal A et la date $\phi_A(x)$ à laquelle ces données sont écrasées ou désallouée. La date d'allocation des données $A[x][*]$ est restreinte à une fonction affine :

$$\alpha_A(x) = \alpha_{A1}x + \alpha_{A2} \tag{6.12}$$

L'objectif est de contracter la mémoire d'un canal afin de respecter une taille fixe connue à l'avance. Il s'agit alors d'une allocation modulo : la date à laquelle est écrasée une donnée correspond à sa date d'allocation modulo la taille de la mémoire utilisée. Si un canal A utilise une mémoire de taille B, et que le fait d'allouer et d'écraser des données dure δ cycles, la date d'allocation des données $A[x][*]$ est liée à celle de leur désallocation par la relation suivante :

$$\alpha_A(x + B) + \delta = \phi_A(x) \tag{6.13}$$

De plus, la date d'allocation des données $A[f_{WA}][*]$ est nécessairement antérieure à celle à laquelle est ordonnancée le nœud W produisant ces données :

$$\forall \vec{i} \in \mathcal{D}_W, \alpha_A(f_{WA}(\vec{i})) \leq \theta_W(\vec{i}) \tag{6.14}$$

Il est évident qu'une lecture des données $A[f_{RA}(\vec{i})][*]$ par un nœud R doit être antérieure à leur

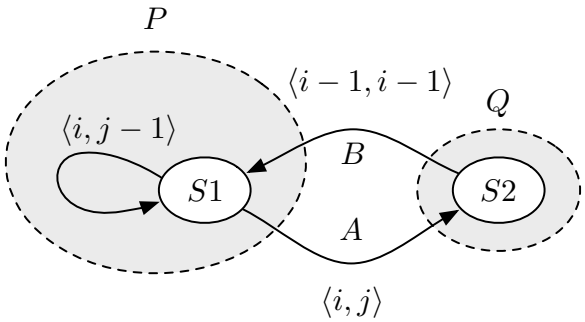


FIGURE 6.2 – Deux processus (P et Q) communiquant par un canal A. Les dépendances $\langle i, j\rangle$ et $\langle i-1, i-1\rangle$ interdisent tout ordonnancement monodimensionnel.

désallocation :

$$\forall \vec{i} \in \mathcal{D}_R, \phi_A(f_{RA}(\vec{i})) \geq \theta_R(\vec{i}) \tag{6.15}$$

En utilisant la relation (6.13), la contrainte précédente devient :

$$\forall \vec{i} \in \mathcal{D}_R, \alpha_A(f_{RA}(\vec{i}) + B) + \delta \geq \theta_R(\vec{i}) \tag{6.16}$$

De la même manière que pour les contraintes (6.2) et (6.3) de légalité d'une communication, les contraintes (6.14) et (6.16) sont exprimées respectivement pour le processus producteur et les processus consommateurs d'une communication. Elles sont linéarisables en utilisant la forme affine du lemme de Farkas ou la méthode des sommets.

Si un *buffer* circulaire est un modèle suffisamment simple pour réaliser simultanément l'ordonnancement des processus et une contraction mémoire, l'hypothèse de Feautrier n'en demeure pas moins pénalisante : pourquoi transmettre toutes les données $A[x][*]$ alors que seule une donnée $A[x][y]$ sera utilisée lors d'une itération ? Ce choix de conception provient certainement de la difficulté de modéliser l'allocation modulo pour un canal multidimensionnel.

D'autre part, malgré nos efforts, nous n'avons pas réussi à généraliser cette approche aux cas multidimensionnels (cf. sous-section suivante) : comment impacter le modulo issu de la taille maximale mémoire dans les fonctions d'ordonnancement multidimensionnel tout en conservant des contraintes linéaires ? Cet aspect reste, à notre connaissance, un problème ouvert.

6.2.3 Généralisation aux cas multidimensionnels

De même que pour les ordonnancements affines usuels, il n'est pas toujours possible de déterminer un ordonnancement monodimensionnel légal pour un réseau de processus. Le cas le plus évident est celui d'un processus ne disposant pas d'ordonnancement monodimensionnel. Une autre possibilité est que ce soit le réseau lui-même qui ne soit pas ordonnançable. Quand bien même chaque processus indépendant disposerait d'un domaine d'ordonnancement non vide, ce sera pourtant le cas si l'intersection des domaines des processus forme un espace vide.

Malheureusement, l'algorithme d'ordonnancement multidimensionnel [56] de Feautrier n'est pas applicable aux ordonnancements affines structurés, son caractère itératif entraîne en effet la perte de la modularité.

6.2.3.1 Exemple (cas monodimensonnel)

On considère le réseau de processus illustré par la figure 6.2. Dans cet exemple, nous allons montrer qu'aucun ordonnancement monodimensionnel n'existe alors que chaque processus est ordonnançable indépendamment des autres.

Les différents domaines de définition des nœuds et des dépendances de données sont décrits dans le tableau suivant :

Nœuds	Domaines	
$S1$	$\mathcal{D}_{S1} = N \rightarrow \{ (i,j) \mid N \geq i \geq 0, i \geq j \geq 0 \}$	
$S2$	$\mathcal{D}_{S2} = N \rightarrow \{ (i,j) \mid N \geq i \geq 0, i \geq j \geq 0 \}$	
Dépendances	Domaines	Vecteur d'itération
$S1 \rightarrow S2$	$\mathcal{D}_{S1 \rightarrow S2} = N \rightarrow \{ (i,j) \mid \}$	$\langle i, j \rangle$
$S1 \rightarrow S1$	$\mathcal{D}_{S1 \rightarrow S1} = N \rightarrow \{ (i,j) \mid j \geq 1 \}$	$\langle i, j-1 \rangle$
$S2 \rightarrow S1$	$\mathcal{D}_{S2 \rightarrow S1} = N \rightarrow \{ (i,j) \mid i \geq 1, j < 1 \}$	$\langle i-1, i-1 \rangle$

Les prototypes d'ordonnancement des nœuds et des canaux de communication sont :

$$\begin{aligned}
\theta_{S1}(\vec{i}) &= \mu_{S11} i + \mu_{S12} j + \mu_{S13} N + \mu_{S14} \\
\theta_{S2}(\vec{i}) &= \mu_{S21} i + \mu_{S22} j + \mu_{S23} N + \mu_{S24} \\
\theta_{A}(\vec{i}) &= \mu_{A1} i + \mu_{A2} j + \mu_{A3} N + \mu_{A4} \\
\theta_{B}(\vec{i}) &= \mu_{B1} i + \mu_{B2} j + \mu_{B3} N + \mu_{B4}
\end{aligned}$$

En appliquant l'algorithme d'ordonnancement structuré de Feautrier, on obtient le domaine non vide pour les communications de P :

$$\left\{ \begin{array}{rcl}
-\mu_{B3} + \mu_{A3} & \geq & 0 \\
-1 - \mu_{B1} - \mu_{B2} - \mu_{B3} + \mu_{A1} + \mu_{A2} + \mu_{A3} & \geq & 0 \\
-2 + \mu_{B2} - \mu_{B3} - \mu_{B4} + \mu_{A1} + \mu_{A2} + \mu_{A3} + \mu_{A4} & \geq & 0 \\
-\mu_{B1} - \mu_{B2} - \mu_{B3} + \mu_{A1} + \mu_{A3} & \geq & 0 \\
-1 + \mu_{B2} - \mu_{B3} - \mu_{B4} + \mu_{A1} + \mu_{A3} + \mu_{A4} & \geq & 0
\end{array} \right.$$

De même, les contraintes de communication pour l'ordonnancement monodimensionnel du processus Q sont :

$$\left\{ \begin{array}{rcl}
-\mu_{A3} + \mu_{B3} & \geq & 0 \\
-\mu_{A1} - \mu_{A2} - \mu_{A3} + \mu_{B1} + \mu_{B2} + \mu_{B3} & \geq & 0 \\
-1 - \mu_{A4} + \mu_{B4} & \geq & 0 \\
-\mu_{A1} - \mu_{A3} + \mu_{B1} + \mu_{B3} & \geq & 0
\end{array} \right.$$

En utilisant un outil de calcul polyédrique (i.e., polylib ou ISL), on obtient que l'intersection des deux ensembles de contraintes forme un espace vide : les communications entre P et Q n'ont donc pas d'ordonnancement monodimensionnel légal.

6.2.3.2 Ordonnancement multidimensionnel des canaux de communication

À la différence d'une dépendance de données classique, une communication par un canal n'exprime pas directement la causalité. C'est cette dissociation qui apporte la modularité des ordonnancements structurés. De même que pour les dépendances classiques, il est possible d'appliquer les notions de

dépendances faiblement et fortement satisfaites (cf. 4.3.3) à cette relation de causalité indirecte. La différence se situe uniquement dans le fait que la date de la source d'une dépendance n'est pas celle de la production de la donnée, mais celle de son écriture dans le canal.

Définition 19 (Dépendance de canal fortement satisfaite) *Soit R une instruction lisant une donnée $A[f_R(\vec{i})]$ dans un canal A. Une dépendance fortement satisfaite à la dimension p de R sur le canal A est définie par :*

$$\forall \vec{i} \in \mathcal{D}_R, \theta_R^p(\vec{i}) \geq \theta_A^p(f_R(\vec{i})) + 1 \tag{6.17}$$

Définition 20 (Dépendance de canal faiblement satisfaite) *Soit R une instruction lisant une donnée $A[f_R(\vec{i})]$ dans un canal A. Une dépendance faiblement satisfaite à la dimension p de R sur le canal A est définie par :*

$$\forall \vec{i} \in \mathcal{D}_R, \theta_R^p(\vec{i}) \geq \theta_A^p(f_R(\vec{i})) \tag{6.18}$$

Pour que l'ordonnancement multidimensionnel d'un canal soit légal, chaque dépendance doit être fortement satisfaite pour une dimension p_{sat} et faiblement satisfaite pour les $p_{sat} - 1$ dimensions précédentes. L'algorithme usuel d'ordonnancement multidimensionnel [56] est itératif et requiert un choix explicite de p_{sat}. En effet, à chaque itération k de l'algorithme, on cherche à satisfaire fortement le plus de dépendances possibles ($p_{sat} = k$). Celles qui ne peuvent être fortement satisfaites doivent l'être faiblement et seront de nouveau analysées lors de la prochaine itération. Cette méthode est donc incompatible avec la modularité des processus puisque le choix de p_{sat} doit être fait en amont de la projection des coefficients d'ordonnancement internes du processus sur ceux des canaux.

Préserver la modularité des processus requiert donc de déterminer p_{sat} uniquement lors de la recherche d'un ordonnancement légal de l'ensemble des canaux du problème. Vasilache [186] et Pouchet [149] ont montré que les ordonnancements multidimensionnels légaux d'un PRDG sont exprimables dans une forme convexe (cf. 4.3.3). Dans la forme de Pouchet, le choix de p_{sat} correspond à ce qu'une variable binaire $\delta^{p_{sat}}$ soit égale à un. Le corollaire 1 est une conséquence directe de ce théorème et montre qu'une forme convexe existe pour contraindre les ordonnancements multidimensionnels légaux d'un canal. Elle repose sur la notion de *nullification* d'une contrainte affine (cf. 4.2.1) pour une constante $K \in \mathbb{Z}$ suffisamment grande.

Corrolaire 1 (Forme convexe d'une dépendance de canal) *Soit R une instruction lisant une donnée $A[f_R(\vec{i})]$ dans un canal A. Pour m dimensions, la dépendance de R sur le canal A est satisfaite si les trois contraintes suivantes sont respectées :*

$$(i) \qquad \forall p \in \{1, \ldots, m\}, \delta_A^p \in \{0, 1\} \tag{6.19}$$

$$(ii) \qquad \sum_{p=1}^{m} \delta_A^p = 1 \tag{6.20}$$

$$(iii) \qquad \forall p \in \{1, \ldots, m\}, \theta_R^p(\vec{i}) - \theta_A^p(f_R(\vec{i})) \geq \delta_A^p - \sum_{k=1}^{p-1} \delta_A^k.(K.\vec{n} + K) \tag{6.21}$$

La forme convexe d'une dépendance de canal nous permet donc d'ajouter le choix des p_{sat} de chaque communication à l'espace de légalité des communications de son processus consommateur. Aucun choix explicite n'est requis lors de la construction des contraintes : la modularité est donc préservée.

6.2.3.3 Exemple (cas multidimensionnel)

Intéressons-nous de nouveau au réseau de processus de la figure 6.2. On cherche maintenant à déterminer un ordonnancement avec, au maximum, deux dimensions.

Processus P Le processus P contient une dépendance interne, un canal de sortie (A) ainsi qu'un canal d'entrée (B). Pour chaque dépendance, on construit la forme convexe de ses contraintes de légalité.

Dans le cas du canal B, on exprime la forme convexe de la dépendance (corollaire 1) pour deux dimensions :

$$\begin{aligned} \theta^1_{S1}(i,j) - \theta^1_B(i-1,i-1) &\geq \delta^1_B \\ \theta^2_{S1}(i,j) - \theta^2_B(i-1,i-1) &\geq \delta^2_B - \delta^1_B(K.N+K) \\ \delta^1_B + \delta^2_B &= 1 \end{aligned}$$

Ainsi, si la dépendance est fortement satisfaite à la première dimension, alors les contraintes issues de la seconde dimension sont toujours respectées. En effet, dans ce cas on a $\delta^1_B = 1$ et $\delta^1_B = 0$, on en déduit que $\theta^2_{S1}(i,j) - \theta^2_B(i-1,i-1) \geq -(K.N+K)$ ce qui est toujours vrai pour un K suffisamment grand [186].

Après l'application de la forme affine du lemme de Farkas et l'élimination des multiplieurs associés on obtient les contraintes suivantes :

$$p=1 \begin{cases} \mu^1_{S11} + \mu^1_{S13} + \mu^1_{S14} \geq \delta^1_B + \mu^1_{B3} + \mu^1_{B4} \\ \mu^1_{S11} + \mu^1_{S13} \geq \mu^1_{B1} + \mu^1_{B2} + \mu^1_{B3} \\ \mu^1_{S13} \geq \mu^1_{B3} \\ \delta^1_B \geq 0 \\ 1 \geq \delta^1_B \end{cases}$$

$$p=2 \begin{cases} -\mu^2_{B3} - \mu^2_{B4} + \mu^2_{S11} + \mu^2_{S13} + \mu^2_{S14} + K\delta^1_B - \delta^2_B \geq 0 \\ -\mu^2_{B1} - \mu^2_{B2} - \mu^2_{B3} + \mu^2_{S11} + \mu^2_{S13} + K\delta^1_B \geq 0 \\ -\mu^2_{B3} + \mu^2_{S13} + K\delta^1_B \geq 0 \\ \delta^2_B \geq 0 \\ 1 - \delta^2_B \geq 0 \end{cases}$$

Dans le cas du canal A, les contraintes sont identiques pour chacune des dimensions puisque la causalité n'est pas nécessaire en écriture (cf. inéquation 6.2). On procède donc de la même manière qu'avec un ordonnancement monodimensionnel et on obtient les contraintes suivantes quelle que soit la dimension $p \in [1,2]$:

$$\begin{cases} -\mu^p_{S14} + \mu^p_{A4} \geq 0 \\ -\mu^p_{S11} - \mu^p_{S12} - \mu^p_{S13} + \mu^p_{A1} + \mu^p_{A2} + \mu^p_{A3} \geq 0 \\ -\mu^p_{S13} + \mu^p_{A3} \geq 0 \\ -\mu^p_{S11} - \mu^p_{S13} + \mu^p_{A1} + \mu^p_{A3} \geq 0 \end{cases}$$

Pour la dépendance interne $S1 \rightarrow S1$, on applique la méthode de Pouchet. Les contraintes issues

de cette dépendance sont alors :

$$p=1 \begin{cases} \delta^1_{S1\to S1} & \geq & 0 \\ 1 & \geq & \delta^1_{S1\to S1} \\ \mu^1_{S12} & \geq & \delta^1_{S1\to S1} \end{cases}$$

$$p=2 \begin{cases} \mu^2_{S12}+K\delta^1_{S1\to S1} & \geq & \delta^2_{S1\to S1} \\ \delta^2_{S1\to S1} & \geq & 0 \\ 1 & \geq & \delta^2_{S1\to S1} \end{cases}$$

L'élimination des coefficients internes produit D_P le domaine des ordonnancements multidimensionnels légaux des canaux de P. Il s'agit d'un espace à 18 dimensions qui par souci de lisibilité n'est pas représenté. La différence essentielle par rapport au cas monodimensionnel repose sur les nouvelles dimensions $(\delta^p, \forall p \in [1,2])$ qui modélisent la causalité des dépendances dans l'espace de recherche.

Processus Q On procède de la même manière pour le processus Q qui ne dispose d'aucune dépendance interne, mais qui lit dans le canal A et écrit dans le canal B. Le domaine des ordonnancements multidimensionnels des canaux de Q est nommé $\mathcal{D}_Q$.

Ordonnancement des canaux puis des processus Soit $\mathcal{D} = \mathcal{D}_P \cap \mathcal{D}_Q$, le domaine des ordonnancements légaux du réseau formé par P et Q. Une solution des contraintes de communication est :

$$\begin{cases} \delta^2_A = \delta^2_B = 1 \\ \delta^1_A = \delta^1_B = 0 \\ \mu^1_{B1} = \mu^2_{B3} = \mu^2_{B4} = \mu^1_{A1} = \mu^2_{A3} = 1 \\ \mu^1_{B2} = \mu^1_{B3} = \mu^1_{B4} = \mu^2_{B1} = \mu^2_{B2} = \mu^1_{A2} = \mu^1_{A3} = \mu^1_{A4} = \mu^2_{A1} = \mu^2_{A2} = \mu^2_{A4} = 0 \end{cases}$$

A partir des coefficients des canaux, on ordonnance indépendamment les processus P et Q pour obtenir :

$$(P) \begin{cases} \delta^2_{S1\to S1} = 1 \\ \delta^2_{S1\to S1} = 0 \\ \mu^1_{S11} = 1 \\ \mu^1_{S12} = \mu^1_{S13} = \mu^1_{S14} = 0 \\ \mu^2_{S12} = 1 \\ \mu^2_{S11} = \mu^2_{S13} = \mu^2_{S14} = 0 \end{cases} \qquad (Q) \begin{cases} \mu^1_{S21} = 1 \\ \mu^1_{S22} = \mu^1_{S23} = \mu^1_{S24} = 0 \\ \mu^2_{S23} = \mu^2_{S24} = 1 \\ \mu^2_{S21} = \mu^1_{S22} = \mu^2_{S21} = \mu^2_{S22} = 0 \end{cases}$$

On peut en déduire un ordonnancement multidimensionnel de $S1$ et $S2$:

$$\theta_{S1}(\vec{i}) = (i, j)$$

$$\theta_{S2}(\vec{i}) = (i, N+1)$$

6.3 Formulation du problème

Dans la section précédente, nous avons détaillé l'algorithme d'ordonnancement affine modulaire de Feautrier ainsi que sa généralisation aux cas multidimensionnels. Dans cette section, nous étudions le lien entre l'ordonnancement affine modulaire et la problématique d'extension de jeux d'instructions.

6.3.1 Représentation fine des dépendances de données

Le PRDG (cf. paragraphe 4.2.3) offre un formalisme particulièrement intéressant dans le cadre d'un flot de compilation pour processeurs spécialisés. En effet, toutes les instructions d'une SCoP [3] sont représentées dans un unique graphe. Les techniques de couverture de graphe permettent alors d'identifier et de sélectionner des occurrences de motifs regroupant des instructions se situant pourtant dans des blocs de base différents. Les travaux existants qui exploitent cette représentation effectuent des optimisations de boucles sans s'intéresser aux calculs effectués dans les instructions, mais uniquement à leurs dépendances de données qui contraignent l'espace des ordonnancements possibles.

```
1  for(i=0;i<N;++i){
2    for(j=0;j<N;++j){
3      C[i][j] = 0;
4      for(k=0;k<N;++k){
5        C[i][j] += A[i][k]*B[k][j];
6      }
7    }
8  }
9  for(i=0;i<N;++i){
10   for(j=0;j<N;++j){
11     F[i][j] = 0;
12     for(k=0;k<N;++k){
13       F[i][j] += D[i][k]*E[k][j];
14     }
15   }
16 }
17 for(i=0;i<N;++i){
18   for(j=0;j<N;++j){
19     G[i][j] = 0;
20     for(k=0;k<N;++k){
21       G[i][j] += C[i][k]*F[k][j];
22     }
23   }
24 }
```

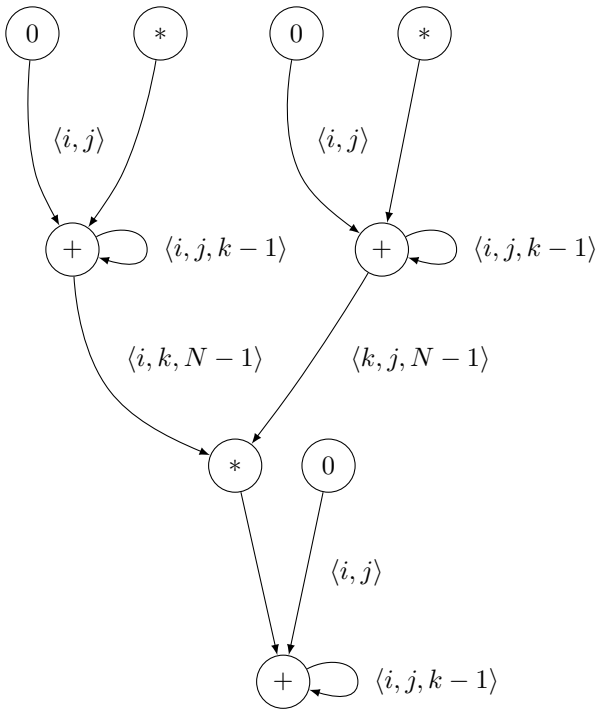


FIGURE 6.3 – PRDG à dépendances fines du triple produit matriciel.

Dans notre cas, l'objectif est de déterminer des groupes d'opérations qui seront déportés sur une extension matérielle. La nature des opérations effectuées dans les instructions du PRDG est donc importante. La méthode que nous proposons s'appuie sur une représentation plus fine des dépendances de données qu'un PRDG usuel, puisqu'un nœud y représente une instruction primitive de calcul (addition, multiplication, etc.). Les dépendances de données correspondent, quant à elles, aux

3. Static Control Part

opérandes possibles de chaque opération. Elles peuvent donc être fonction du contexte d'exécution de l'opération.

Si une dépendance ne dépend pas du contexte d'exécution de l'opération qui consomme la donnée, on parlera, dans la suite, de *dépendance de données totale* (définition 1).

Définition 1 (Dépendance de données totale) *Soit $G(N, E)$ un PRDG et $S_1, S_2 \in N$. Une dépendance $e_{S_1 \to S_2} \in E$ est une dépendance de données totale si et seulement si $De_{S_1 \to S_2}$ est le domaine univers (aucune contrainte).*

La figure 6.3 décrit la représentation fine du PRDG du triple produit matriciel. Les dépendances des additions sur les multiplications sont totales alors que les additions dépendent d'elles mêmes à l'itération $\langle i, j, k-1 \rangle$ si $N \geq k \geq 1$. Si la source et la destination d'une dépendance de données se trouvent à la même itération, on parle alors de dépendance de données directe (définition 2). Dans l'exemple, seules les dépendances totales sont des dépendances directes. Elles sont ici issues de l'expansion du PRDG standard en sa forme fine.

Définition 2 (Dépendance de données directe) *Soit $G(N, E)$ un PRDG et $S_1, S_2 \in N$. Une dépendance $e_{S_1 \to S_2} \in E$ est une dépendance de données directe si et seulement si :*

$$\mathcal{D}_{e_{S_1 \to S_2}} : \{i^1_{S_1}, ..., i^k_{S_1}, i^1_{S_2}, ..., i^k_{S_2} | i^1_{S_2} = i^1_{S_1}, ..., i^k_{S_2} = i^k_{S_1}\} \tag{6.22}$$

Dans la suite de ce chapitre, quand l'on parle de PRDG, on sous-entend qu'il s'agit de sa représentation fine où toutes les instructions primitives sont détaillées.

6.3.2 Sélection et ordonnancement affine d'instructions spécialisées

La plupart des approches de sélection d'instructions spécialisées s'appuient sur un graphe acyclique (DAG) issu d'un bloc de base de la représentation intermédiaire du compilateur. Dans cette représentation, les nœuds sont des opérations de calcul supportées par le processeur. Sélectionner une instruction spécialisée consiste alors à sélectionner une occurrence d'un motif de calcul dans le graphe de l'application. Il s'en suit que l'ensemble des nœuds qui y sont contenus est alors exécuté sur l'extension matérielle du processeur. Le PRDG fournit beaucoup plus d'informations qu'un simple DAG : chaque nœud est associé à un domaine polyédrique de définition et les liens identifient les instructions produisant les données ainsi que leurs positions dans l'espace d'itération. L'expressivité du PRDG nécessite cependant de faire la distinction entre la notion de motif et celle d'instruction spécialisée.

Dans notre approche, nous considérons qu'une occurrence de motif sera mise œuvre par un *macrobloc* matériel (cf. figure 6.4) contenant le chemin de données d'une ou plusieurs instructions spécialisées communiquant entre elles par l'intermédiaire de mémoires embarquées dans le macrobloc. De plus, un des objectifs de l'approche est de sélectionner des instructions spécialisées vectorisables. Nous chercherons donc également à déterminer un ordonnancement affine respectant cette contrainte.

6.3.2.1 Sélection d'instruction spécialisée dans un PRDG

Un motif dans un PRDG est constitué de nœuds pouvant être définis sur des domaines d'itérations différents, ils n'auront alors pas les mêmes points dans l'espace multidimensionnel de leurs nids

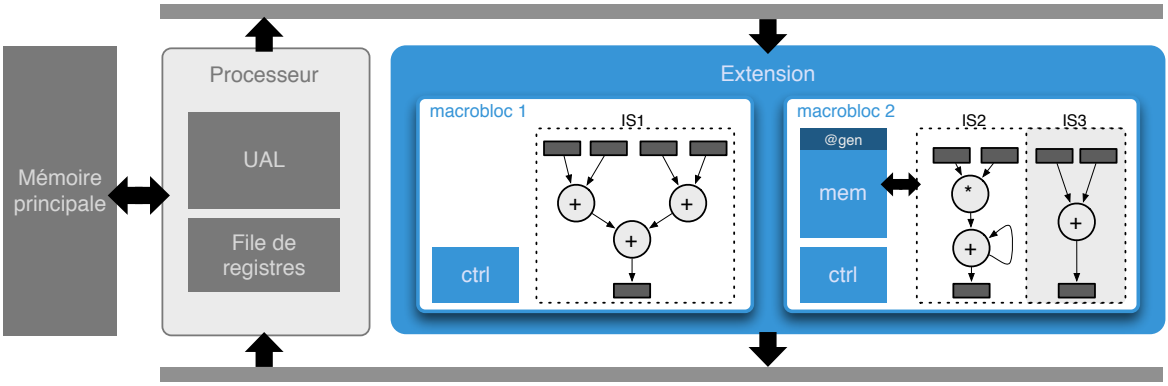


FIGURE 6.4 – Schéma fonctionnel d'un processeur couplé à une extension comportant les composants matériels de deux macroblocs. Le deuxième macrobloc utilise une mémoire pour enregistrer des résultats intermédiaires des instructions spécialisées IS_1 ou IS_2.

de boucles respectifs. De plus, les liens entre chacun des nœuds d'un motif représentent une dépendance de données dont la source et la destination sont susceptibles d'être exécutées à des itérations différentes.

Dans ce contexte, un motif est susceptible de ne pas correspondre à une unique instruction spécialisée. En effet, si l'on choisit de regrouper, par exemple, la multiplication du dernier nid de boucles du triple produit matriciel (cf. figure 6.3) avec l'addition du second, la dépendance de données indique que l'itération source est $\langle i, k, N-1 \rangle$. Il ne sera donc pas possible de regrouper ces deux nœuds en une seule instruction spécialisée puisqu'une donnée produite par l'addition ne sera utilisable qu'à la dernière itération de k.

Une première approche envisageable est de considérer uniquement les occurrences de motifs dont toutes les dépendances de données sont directes et totales. Ainsi, de même que pour un DAG, une occurrence de motif identifiera une unique instruction spécialisée. Cependant, cette approche ne cherche pas à transformer le code original pour y faire apparaître de nouvelles instructions spécialisées et on retombe alors sur les travers des techniques existantes puisqu'on ne sélectionnera alors principalement que des instructions spécialisées contenues dans le même bloc de base.

D'autre part, un des intérêts du PRDG est de fournir une connaissance exacte des itérations où sont produites les données. Cette information offre des possibilités très intéressantes quant au déport de dépendances de données dans des ressources de mémorisation (e.g., registres ou mémoires) embarquées sur l'extension matérielle. Cependant, il n'est évidemment pas raisonnable de matérialiser toutes les dépendances d'un PRDG comme des mémoires de l'extension. Un choix explicite des dépendances mémorisées sur l'extension constitue un autre paramètre d'optimisation difficile à évaluer et complique considérablement le problème.

La solution que nous adoptons consiste à considérer qu'**une occurrence de motif est une zone de calcul et de mémorisation intégralement déportée sur l'extension matérielle**. À la différence de l'approche précédente, cette zone peut éventuellement contenir des dépendances pour lesquelles les itérations de la source et de la destination sont différentes. Dans ce contexte, une occurrence de motif contient potentiellement plusieurs instructions spécialisées (définition 3). Une dépendance de données indirecte, interne à l'occurrence, y utilisera alors directement une mémoire de

l'extension et évitera une communication coûteuse avec le processeur et sa hiérarchie mémoire.

Définition 3 (Instruction spécialisée atomique) *Soit $G(N, E)$ un PRDG. Une instruction spécialisée est constituée d'un ensemble de nœuds $I \subset N$ dont les domaines de définitions sont identiques et qui sont reliés uniquement par des dépendances directes.*

La sélection d'une occurrence de motif répond à la fois au problème de partitionnement des nœuds et à celui du choix des dépendances déportées sur les mémoires de l'extension. En effet, seules les occurrences de motifs de taille unitaire sont exécutées sur le processeur, les autres constituent des ensembles d'instructions spécialisées. De plus, les données de toutes les dépendances non directes d'une occurrence sont mémorisées sur l'extension. Réciproquement, toutes les dépendances sortantes d'une occurrence de motif n'utilisent pas la mémoire de l'extension, les données produites sont communiquées au processeur.

Afin d'éviter une confusion entre la notion de motif d'un DAG et celle d'un PRDG, une occurrence de motif dans un PRDG est appelée *macrobloc* (définition 4).

Définition 4 (Macrobloc spécifique) *Soit $G(N, E)$ un PRDG et P une bibliothèque de motifs. Un macrobloc spécifique est une instance M d'un motif $P_k \in P$ dans G exécuté intégralement sur l'extension matérielle. Il est constitué d'un ensemble d'instructions spécialisées atomiques reliées par des dépendances nécessitant une mémorisation.*

6.3.2.2 Ordonnancement affine des instructions spécialisées

La représentation fine du PRDG nous permet d'identifier des instructions spécialisées qui se situent pourtant dans des boucles différentes. La figure 6.5 illustre un exemple où les nids de boucles peuvent être fusionnés pour détecter une instruction spécialisée vectorisable. En effet, si l'on dispose d'un motif contenant trois additions, l'intégralité du PRDG peut être couverte par un seul macrobloc ne contenant qu'une unique instruction spécialisée puisque toutes les dépendances de données sont directes. De plus, si l'on considère que seul le tableau G est utile (*liveout*), l'instruction spécialisée sélectionnée élimine toute mémorisation temporaire inutile dans les tableaux E et F. Si dans cet exemple, il est évident que le code source aurait pu être optimisé (en fusionnant les boucles et les calculs), d'autres cas sont beaucoup plus difficiles à détecter et à optimiser en vue d'une approche d'extension de jeu d'instructions.

Un des objectifs de notre approche est d'identifier des macroblocs qui respectent des contraintes spécifiques. En effet, les instructions spécialisées doivent être vectorisables et l'on souhaite également avoir la possibilité d'ajouter des contraintes supplémentaires (i.e. limiter la taille des mémoires utilisées pour temporiser les données produites dans chaque macrobloc). Les opérations qui ne peuvent être couvertes par des macroblocs respectant toutes ces contraintes sont exécutées sur le processeur. Elles pourront néanmoins être la cible d'un flot standard de spécialisation, analysant chaque bloc de base du code source regénéré à partir du PRDG couvert.

Naturellement, le respect de toutes ces contraintes nécessite souvent de transformer le code original d'un programme. Les ordonnancements affines nous permettent d'explorer l'espace des transformations de boucles de manière à faire apparaître des macroblocs respectant à la fois les contraintes de légalité et les contraintes additionnelles. Cependant, la sélection et l'ordonnancement de différents

```
1  for(i=0;i<N;i++){
2    for(j=0;j<N;j++){
3      E[i][j] = A[i][j] + B[i][j];
4    }
5  }
6  for(i=0;i<N;i++){
7    for(j=0;j<N;j++){
8      F[i][j] = C[i][j] + D[i][j];
9    }
10 }
11 for(i=0;i<N;i++){
12   for(j=0;j<N;j++){
13     //G: liveout array
14     G[i][j] = E[i][j] + F[i][j];
15   }
16 }
```

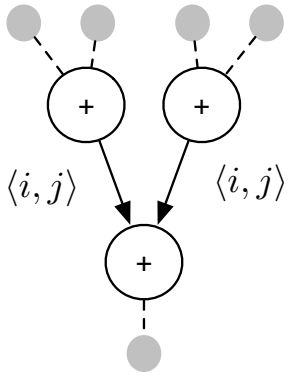


FIGURE 6.5 – Calcul de la somme de quatre matrices : $G = A + B + C + D$. Il est évident ici que les nids de boucles peuvent être fusionnés. La sélection et l'ordonnancement d'un macrobloc permettent d'identifier une unique instruction spécialisée vectorisable.

macroblocs peuvent être exclusifs. Le cas le plus évident est quand un même nœud est couvert par deux macroblocs potentiels : un seul macrobloc sera sélectionné. Deux macroblocs disjoints peuvent également être exclusifs. Dans ce cas, la sélection du premier macrobloc imposera des contraintes sur l'ordonnancement global du PRDG qui ne pourront être satisfaites par le second.

Nous proposons d'utiliser les résultats de l'ordonnancement affine modulaire d'un réseau de processus (cf. section 6.2), pour énoncer dans un cadre unifié (programmation par contraintes), un problème conjoint de couverture et d'ordonnancement du PRDG. Le principe est d'assimiler chaque macrobloc à un processus. Les contraintes évoquées précédemment sont alors exprimées de manière modulaire et ne dépendent que des coefficients d'ordonnancement des communications externes du macrobloc. La modularité des contraintes de chaque macrobloc est nécessaire à la gestion de leurs éventuelles relations d'exclusivité dans la couverture et l'ordonnancement du PRDG. Ainsi, si un macrobloc n'est pas sélectionné, ses contraintes d'ordonnancement n'ont pas à être respectées. Réciproquement, si les contraintes d'ordonnancement d'un macrobloc ne sont pas respectées alors il ne sera pas sélectionné dans la couverture du PRDG. L'algorithme conjoint d'ordonnancement et de couverture est détaillé dans la section 6.4. Cet algorithme consiste principalement à résoudre un problème de satisfaction de contraintes mêlant les contraintes non linéaires de couverture d'un graphe (cf. sous-section 3.3.1) au problème d'ordonnancement affine structuré.

6.3.3 Génération de code pour l'architecture cible

Dans les paragraphes précédents, nous nous sommes intéressés à la notion de motifs de calcul dans un PRDG. Les ordonnancements modulaires sont utilisés pour formuler un problème conjoint d'ordonnancement affine et de couverture du PRDG, dont le résultat est la sélection d'un ensemble de macroblocs exécutés sur l'extension matérielle d'un processeur extensible. Nous présentons maintenant la manière d'exploiter le résultat de cette couverture sur l'architecture cible.

6.3.3.1 Exploitation de l'extension matérielle

L'architecture cible est un processeur extensible (e.g., NiosII) fortement couplé à une extension matérielle. Tout comme l'approche présentée dans le chapitre 3, cette extension matérielle bénéficie d'un accès direct à la file de registres du processeur et dispose d'un nombre limité de bus d'entrée et de sortie pour communiquer avec le processeur. La différence porte sur le fait que les motifs sélectionnés dans le graphe correspondent cette fois à des macroblocs pouvant contenir *plusieurs* instructions spécialisées (l'instruction spécialisée k est notée IS_k).

La figure 6.4 présente le fonctionnement général de l'architecture cible. Par souci de simplicité, on considère qu'à chaque macrobloc sélectionné correspond un composant matériel dédié sur l'extension. Il existe clairement des possibilités de réutilisation du matériel de l'extension, que ce soit au niveau intra ou inter macroblocs. Ainsi, l'information sur les domaines de définition des instructions spécialisées pourrait être avantageusement exploitée pour fusionner des opérateurs (i.e., l'addition de l'IS2 et de l'IS3) dont l'exécution est exclusive sur une même ressource matérielle. Toutefois, la synthèse et l'optimisation du matériel de l'extension sont des problèmes complexes qui n'ont pas été abordés dans le cadre de cette thèse. De plus, nous ne proposons pas d'approche permettant de traiter l'intégralité du problème de vectorisation. En effet, nous nous contentons de sélectionner des instructions qui sont potentiellement vectorisables. Les problématiques de synthèse d'architecture et de gestion de la mémoire que pose la vectorisation ne sont, pour l'instant, pas traitées et constituent autant de perspectives intéressantes.

Chaque composant d'un macrobloc dispose d'un contrôleur chargé de configurer le chemin de données en fonction du code de l'opération (*opcode*) transmis par le processeur. En effet, il est souvent nécessaire de décomposer en plusieurs étapes le comportement de chaque instruction spécialisée d'un macrobloc. Chaque étape correspond alors à une *instruction spécialisée primitive* qui est identifiée de manière unique par un *opcode*. Cette décomposition est une conséquence des contraintes architecturales qui limitent, par exemple, le nombre d'opérandes d'une instruction exécutée sur l'extension (cf. paragraphe 3.2.2.4, page 51). Ainsi, le premier macrobloc du schéma contient l'unique instruction spécialisée (IS_1) qui a été sélectionnée dans le calcul de la somme de quatre matrices (cf. figure 6.5). Le calcul à effectuer comporte quatre opérandes et si l'extension ne dispose que de deux bus d'entrée (cas du NiosII), il est alors nécessaire de décomposer le motif en deux instructions spécialisées primitives. La première transmet deux opérandes à mémoriser dans les registres de l'extension. La seconde instruction spécialisée primitive transmet les deux opérandes restants, lance l'exécution du calcul et transmet le résultat au processeur.

Un macrobloc qui contient plusieurs instructions spécialisées utilise une mémoire pour temporiser les données qui sont produites par une instruction spécialisée et seront utilisées, lors d'une itération ultérieure, par une instruction spécialisée du même macrobloc. C'est le cas, par exemple, du deuxième macrobloc du schéma qui contient un MAC [4] (IS_2) et une opération d'addition (IS_3) également déportée sur l'extension. Une mémoire est donc ajoutée au composant matériel du macrobloc. Un générateur d'adresse est chargé de calculer les positions des différentes lectures et écritures dans cette mémoire en fonction de l'*opcode* et de l'itération courante. La mise en œuvre de ce générateur d'adresse dépend du modèle de mémorisation envisagé. Si l'on peut utiliser un *buffer* circulaire, la mise en œuvre est triviale : une simple machine à état effectue l'adressage modulo la taille du *buffer*.

4. Multiplication et accumulation

```
for(i=0;i<N;i++){ //Parallel dimension
  for(j=0;j<N;j++){ //Parallel dimension
    custom_add3(A[i][j],B[i][j],C[i][j],D[i][j],&G[i][j]);
  }
}
```

```
void custom_add3(int v1, int v2, int v3, int v4, int *result){
  asm volatile("
    custom 1, %0, %1;  /*Load two operands*/
    custom 2, %2, %3, %4;"/*Load remaining operands, launch execution and store the result*/
    :"r"(v1)
    :"r"(v2)
    :"r"(v3)
    :"r"(v4)
    :"=r"(result)
  );
}
```

FIGURE 6.6 – Utilisation séquentielle de la nouvelle instruction spécialisée sur un NIOSII . Les instructions en assembleur correspondent à l'envoi des opérandes et au lancement de l'exécution sur l'extension.

6.3.3.2 Génération de code

Le moyen le plus simple d'exploiter les résultats de la couverture et de l'ordonnancement du PRDG est de générer une nouvelle version du code source de l'application qui contient les appels explicites des instructions spécialisées sélectionnées. Le compilateur natif du processeur sera ensuite chargé de produire le code binaire.

Cependant, à la différence d'un flot standard d'extension de jeu d'instructions, la représentation intermédiaire détermine également la structure du programme (domaines d'itérations des nœuds). L'ordonnancement affine du PRDG risque d'avoir modifié cette structure et un outil tel que CLOOG [18] est indispensable pour générer un parcours (sous forme de nids de boucles et de conditions affines) de l'ensemble des points des domaines ordonnancés. Lors de cette étape, les instructions standard et spécialisées sont réparties dans différentes boucles. Celles-ci font notamment apparaître les fusions ou fissions issues des éventuelles dimensions scalaires des ordonnancements.

La syntaxe des appels aux instructions spécialisées dépend du processeur dont le jeu d'instruction est étendu. Dans le cas du NIOSII, il est possible d'utiliser des *Macros* ou encore des instructions en assembleur pour chaque instruction spécialisée primitive. La figure 6.6 présente un exemple de code généré et qui utilise des instructions assembleurs. Le PRDG couvert et ordonnancé dans cet exemple est celui de la somme de quatre matrices (cf. figure 6.5) où une seule instruction spécialisée a été sélectionnée. Celle-ci correspond dans le code à la procédure `custom_add3` chargée d'initialiser et de lancer l'exécution sur l'extension. La zone de code spécifique est délimitée par la construction `asm volatile` de GCC qui définit une séquence d'instructions en assembleur ne pouvant être optimisées par le compilateur. Dans cette zone, les paramètres des instructions spécialisées primitives sont des registres du processeur, mais leur allocation est laissée au soin du compilateur et ce sont des symboles qui sont explicitement associés aux paramètres des instructions `custom`.

6.4 Algorithme d'ordonnancement affine et de sélection d'instructions spécialisées

6.4.1 Contraintes d'un macro-bloc

Pour que l'ordonnancement affine d'un PRDG couvert soit légal, les macroblocs sélectionnés et ordonnancés doivent tout d'abord respecter les dépendances de données inter-instructions spécialisées. De plus, nous imposons que la dimension la plus interne de chaque instruction spécialisée soit parallèle (condition suffisante à une vectorisation ultérieure).

6.4.1.1 Identification des instructions spécialisées atomiques

Les contraintes internes d'un macrobloc portent toutes sur la notion d'instruction spécialisée. Une instruction spécialisée (cf. définition 3), est constituée d'un groupe de nœuds ayant le même domaine de définition et qui sont liés par des dépendances de donnée directes. Cette notion d'instruction spécialisée pose un problème quant au choix du moment où les identifier.

La première solution est d'analyser chaque macrobloc candidat et d'y identifier les instructions spécialisées avant de résoudre le problème d'ordonnancement. Cependant, dans ce cas, il se peut que la modification des ordonnancements des nœuds fasse apparaître de nouvelles dépendances de donnée directes qui sont autant de nouvelles possibilités de regroupements intéressants. Il est important en effet de garder à l'esprit que plus une instruction spécialisée contient de nœuds, plus elle a de chance d'accélérer l'application en utilisant un chemin de données optimisé sur le matériel.

Une autre possibilité consiste à identifier les instructions spécialisées une fois que le PRDG a été couvert et ordonnancé. Il se pose alors le dual du problème précédent : la modification des ordonnancements risque d'avoir fait disparaître des regroupements potentiels de nœuds et donc des opportunités d'instructions spécialisées plus efficaces.

Néanmoins, le fait d'ajouter des contraintes supplémentaires aux instructions spécialisées est beaucoup plus simple à mettre en œuvre si les instructions spécialisées de chaque macrobloc sont identifiées en amont de l'ordonnancement (première solution). En effet, dans le cas contraire de nombreux choix explicites deviennent autant de paramètres supplémentaires à résoudre lors de l'étape conjointe d'ordonnancement et de couverture : comment modéliser à la fois l'espace des regroupements de nœuds au sein d'un macrobloc et celui de leurs ordonnancements parallèles ?

Une possibilité serait de chercher à sélectionner directement des instructions spécialisées au lieu des macroblocs lors de l'étape de couverture. Cependant, outre les problèmes importants que ce type d'approche pose (développés dans la sous-section 6.3.2), il est alors impossible de regrouper des nœuds qui n'ont pas un domaine de définition identique. En effet, au sein d'un même macrobloc, il est possible que des nœuds soient liés par des dépendances directes, mais sans être sur le même domaine. C'est le cas par exemple dans le macrobloc de la figure 6.7. Les nœuds $n2$, $n3$ et $n4$ ne peuvent constituer une instruction spécialisée puisque leurs dépendances n'ont pas les mêmes conditions et que leurs domaines sont différents.

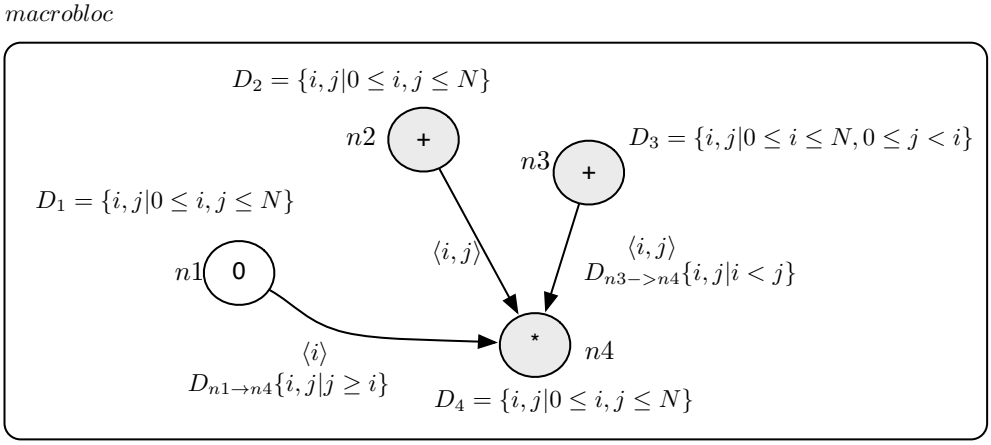


FIGURE 6.7 – Exemple de macrobloc où se trouve une instruction spécialisée à extraire. Les dépendances $n2 \to n4$ et $n3 \to n4$ sont directes, mais les domaines D_3 et $\mathcal{D}_{n3\to n4}$ sont contenus dans D_2 ou D_4.

Afin d'extraire des instructions spécialisées de tels macroblocs, un algorithme itératif (cf. algorithme 5) transforme le PRDG interne à un macrobloc. Le PRDG original n'est évidemment pas modifié pour conserver la modularité : le macrobloc ne sera peut-être pas sélectionné. Le principe est de produire, à chaque itération de l'algorithme, une nouvelle instruction spécialisée en réalisant l'intersection des différents domaines d'un groupe de nœuds dont les dépendances de données sont directes.

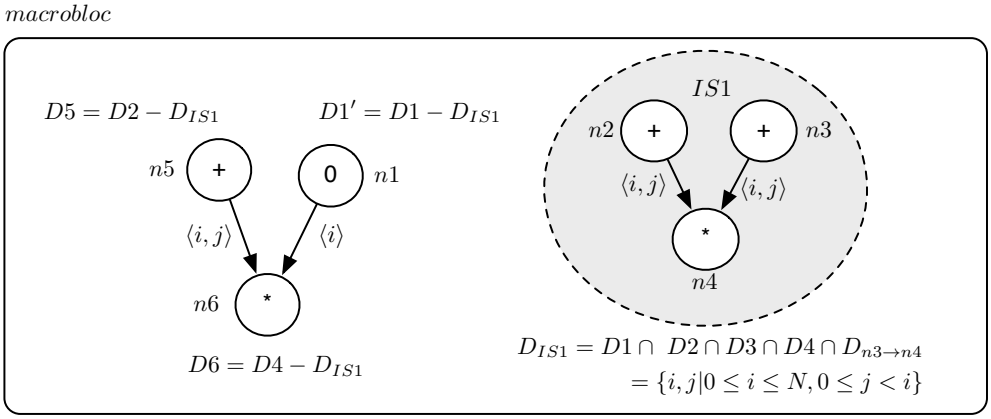


FIGURE 6.8 – Identification de l'instruction spécialisée $IS1$. Les nœuds $n5$ et $n6$ ont été extraits respectivement à partir de $n2$ et de $n4$ (première itération de l'algorithme).

La première étape est d'identifier $IS(N,E)$ (ligne 3), le plus grand groupe de nœuds (N) dont les dépendances de données (E) sont directes. Dans l'exemple de la figure 6.7, le groupe formé par les nœuds $n2$, $n3$ et $n4$ est le seul présent. Si $\mathcal{D}_g$ l'intersection des domaines des nœuds et des liens du groupe n'est pas vide (ligne 6) alors une nouvelle instruction spécialisée sera identifiée. Cependant, certains nœuds ont un domaine $\mathcal{D}_n$ qui inclue le domaine commun ($\mathcal{D}_g$) à tous les éléments de la future instruction spécialisée. Dans ce cas, il est nécessaire d'extraire le nœud pour le reste de son domaine de définition ($\mathcal{D}'_n$). Cette extraction (ligne 11) produit un nœud n' dont le domaine est $\mathcal{D}'_n$

Algorithme 5 Identification des instructions spécialisées dans un macrobloc

```
IdentificationIS (macrobloc)
1  instructions ← ∅
2  nodes ← macrobloc.nodes
3  IS(N, E) ← prochainGroupe(nodes)
4  tant que IS ≠ ∅ faire
5     D_g ← (⋂_{n∈N} D_n) ∩ (⋂_{e∈E} D_e)
6     si D_g ≠ ∅ alors
7        pour ∀n ∈ N faire
8           D_IS ← D_n ∩ D_g
9           D'_n ← D_n − D_g
10          si D'_n ≠ ∅ alors
11             n' ← extraction(n, D'_n)
12             nodes ← nodes ∪ {n'}
13             D_n ← D_IS
14          fin si
15          nodes ← nodes − {n}
16       fin pour
17       instructions ← instructions ∪ instruction(g)
18    sinon
19    fin si
20    IS(N, E) ← prochainGroupe(nodes)
21 fin tant que
22 retourner instructions
```

et qui est ajouté à l'ensemble des nœuds du macrobloc.

Une fois que chaque nœud du groupe a été traité, ils sont tous définis dans le même domaine polyédrique et une instruction spécialisée est identifiée. Ainsi, dans la figure 6.8, l'identification de l'instruction spécialisée $IS1$ donne lieu à l'extraction des nœuds $n5$ et $n6$. Les nœuds extraits sont ajoutés à la liste des nœuds (*nodes*) à analyser par une prochaine itération de l'algorithme. Dans l'exemple de la figure 6.9, une autre instruction spécialisée ($IS2$) est formée à partir des nœuds $n5$ et $n6$ extraits à l'itération précédente. Il n'y a alors plus aucune instruction spécialisée potentielle, l'algorithme est terminé et a identifié deux instructions spécialisées ($IS1$ et $IS2$) dans le macrobloc.

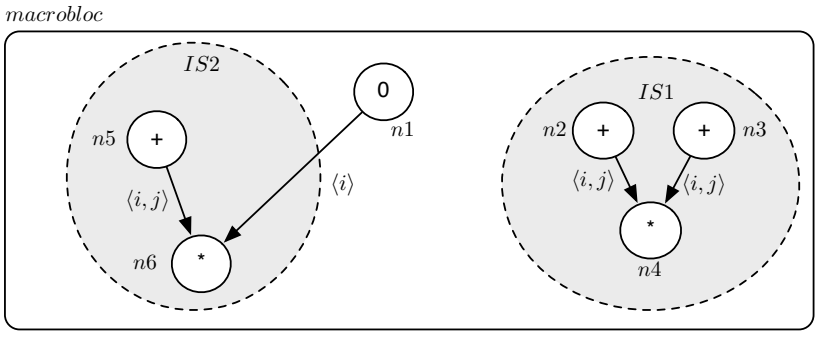


FIGURE 6.9 – Deux instructions spécialisées ont été identifiées dans le macrobloc (deuxième itération de l'algorithme).

Une fois les instructions spécialisées d'un macrobloc identifiées, chacune est vue comme un unique nœud à ordonnancer si le macrobloc est sélectionné.

6.4.1.2 Contraintes de légalité

Les contraintes de légalité d'un macrobloc expriment les conditions dans lesquelles son ordonnancement est légal. De la même manière que pour les ordonnancements affines modulaires, ces contraintes sont issues de chaque dépendance interne et des canaux de communication du macrobloc (cf. sous-section 6.2.1 et sous-section 6.2.3).

6.4.1.3 Instructions spécialisées vectorisables

Pour qu'une instruction spécialisée soit vectorisable, il est suffisant qu'elle exhibe du parallélisme sur sa dimension la plus interne.

Dans le cas de domaines paramétrés, il suffit de s'assurer que la fonction d'ordonnancement, pour cette dimension, ne dépend que des paramètres ou d'une constante. Ainsi, pour une instruction $S1$ définie dans le domaine $\mathcal{D}_{S1} = \{i, j | 0 \leq i, j \leq N\}$, la fonction d'ordonnancement $\theta_{S1}(i, j) = (i, 0)$ implique que la seconde dimension est parallèle. En effet, quelle que soit la valeur des indices, la deuxième dimension de la date d'ordonnancement ne change pas : tous les points (i, j) de l'espace original peuvent être exécutés à la même itération i.

On ajoute donc des contraintes supplémentaires sur les coefficients d'ordonnancement du nœud de chaque instruction spécialisée identifiée dans un macrobloc. Soit S une instruction spécialisée d'un macrobloc et définie sur un domaine D_S disposant de n paramètres, le prototype de son ordonnancement à la dimension la plus interne $k = dim(D_S)$ est :

$$\theta_S^k(\vec{i}) = \mu_{S_0}^k + \mu_{S_1}^k.i_1 + \ldots + \mu_{S_k}^k.i_k + \mu_{S_{k+1}}^k.p_1 + \ldots + \mu_{S_{k+m}}^k.p_m$$

Pour garantir que l'ordonnancement de S est parallèle à la dimension k, on impose les contraintes suivantes sur les coefficients d'ordonnancement :

$$\forall i_n \in \vec{i}, \mu_{S_n}^k = 0 \tag{6.23}$$

6.4.2 Couverture du PRDG

L'objectif de la couverture du PRDG est de sélectionner les macroblocs et d'obtenir un ordonnancement légal de leurs communications externes. La technique de couverture de graphe (cf. sous-section 3.3.1, page 57), basée sur la programmation par contraintes, est adaptable au problème de sélection et d'ordonnancement des macroblocs. Il s'agit alors de construire un CSP (algorithme 6) contenant les contraintes de couverture associées à l'ensemble des contraintes des macroblocs (cf. sous-section 6.4.1).

Algorithme 6 Construction du CSP de couverture et d'ordonnancement des macroblocs.

```
ContraintesSélectionMacroblocs (prdg, motifs)
1  C ← ContraintesCouvertureGraphe(prdg, motifs)
2  pour ∀m ∈ occurrences(prdg, motifs) faire
3     DM_m ← contraintesModulaires(m)
4     C ← polyCP(D_m ∪ m_sel = 0)
5  fin pour
6  retourner C
```

Contraintes sur les occurrences de motifs Une occurrence de motif peut être exécutée sur l'extension uniquement si elle constitue un macrobloc ordonnançable de plus d'un nœud. On associe alors à une occurrence m (ligne 2), un domaine polyédrique $\mathcal{D}_m$ ainsi que DM_m son domaine modulaire après élimination des coefficients internes du macrobloc (ligne 3). Ces contraintes n'ont pas à être impérativement respectées si l'occurrence n'est pas sélectionnée. Or, dans le CSP de couverture (cf. **??**) et pour chaque occurrence, il existe une variable m_{sel} qui indique si l'occurrence est sélectionnée (i.e., $m_{sel} = 1$). Dans le cas contraire, cette variable est nulle. Pour modéliser le fait que les contraintes d'un macrobloc ne doivent être respectées que si celui-ci est sélectionné, on ajoute au problème (ligne 4) la contrainte polyédrique (cf. chapitre 5) suivante :

$$polyCP(DM_m \cup m_{sel} = 0) \tag{6.24}$$

Résolution du problème Une fois que le problème conjoint de couverture et d'ordonnancement est énoncé, il est résolu avec un algorithme standard de résolution de CSP (cf. chapitre 2). Il se pose alors le problème du critère d'optimisation.

Si l'approche de couverture de graphe détaillée dans les chapitres précédents a pour objectif de minimiser la durée totale de l'exécution d'un graphe couvert, il est ici beaucoup plus difficile de l'évaluer. Tout d'abord, dans un PRDG, les opérations sont définies dans des espaces d'itération multidimensionnels. Or, le nombre de points d'un domaine paramétré est le résultat d'une fonction quasi polynomiale [17, 188]. Il est donc impossible de pondérer la sélection d'un macrobloc par une durée constante, comme c'est le cas dans la couverture des opérations d'un bloc de base d'une application. De plus, un PRDG fait parfois apparaître des cycles directs ou indirects entre ses nœuds. Il est donc impossible de formuler une fonction d'optimisation simple qui minimise la durée d'exécution totale du PRDG.

Cependant, il est raisonnable de considérer qu'une couverture déportant le plus de traitement possible sur l'extension matérielle permettra d'obtenir de bonnes performances d'exécution. Tout d'abord, plus la taille d'une instruction spécialisée est importante, plus elle a de possibilités matérielles pour améliorer la durée totale de l'exécution de ses instructions primitives. De plus, les instructions spécialisées d'un même macrobloc communiquent par des mémoires embarquées sur l'extension matérielle. Dès lors, le fait de minimiser le nombre total d'occurrences dans la couverture du PRDG réduit à la fois le nombre d'accès à la mémoire du processeur et le nombre de communications entre le processeur et l'extension. Enfin, le fait de minimiser l'utilisation du processeur offre d'autant plus de possibilités de parallélisation, puisque les instructions spécialisées sélectionnées sont vectorisables.

Toutes ces caractéristiques permettent de formuler une fonction d'optimisation simple qui favorise les performances de l'application. Celle-ci consiste à minimiser le nombre d'occurrences (M) sélectionnées :

$$nbOccurrences = \sum_{m \in M} sel_m \tag{6.25}$$

Il est important de rappeler que l'on ne cherche pas à minimiser une fonction de coût liée à l'ordonnancement affine des instructions spécialisées (e.g., distance des dépendances, nombre de dimensions, etc.). L'objectif est ici de trouver une couverture intéressante et qui respecte un ensemble de contraintes sur les ordonnancements affines. L'ajout de contraintes supplémentaires pour chaque macrobloc permet de répondre à des exigences plus spécifiques, mais ne change rien à l'algorithme.

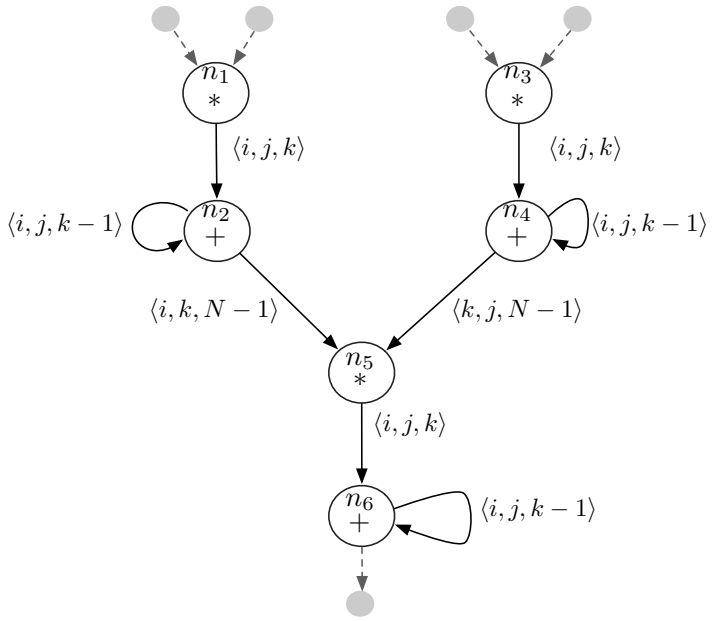


FIGURE 6.10 – PRDG du triple produit matriciel (les constantes sont supprimées pour simplifier l'exemple).

6.4.3 Ordonnancement des occurrences sélectionnées

Après résolution du problème de couverture, on obtient un ensemble de macrobloc sélectionnés et les ordonnancements de chacune des communications utilisées. Les nœuds du PRDG qui ne sont pas couverts par des macroblocs sont exécutés sur le processeur, les autres sont des instructions spécialisées déportées sur l'extension matérielle.

La dernière étape est d'utiliser l'information issue des ordonnancements des communications pour contraindre l'espace d'ordonnancement de chacun des nœuds non couverts et des macroblocs sélectionnés.

Les différents éléments sont ordonnancés séparément en utilisant un algorithme existant tel que l'algorithme d'ordonnancement multidimensionnel (cf. sous-section 4.3.3) ou encore PLUTO qui permettra alors d'obtenir un code pavable dans chaque macrobloc (cf. sous-section 4.3.5). Dans tous les cas, l'élimination des coefficients d'ordonnancement des canaux (par projection) fournit de nouvelles contraintes sur les coefficients d'ordonnancement des nœuds. Celles-ci doivent donc être ajoutées comme contexte additionnel de l'algorithme utilisé.

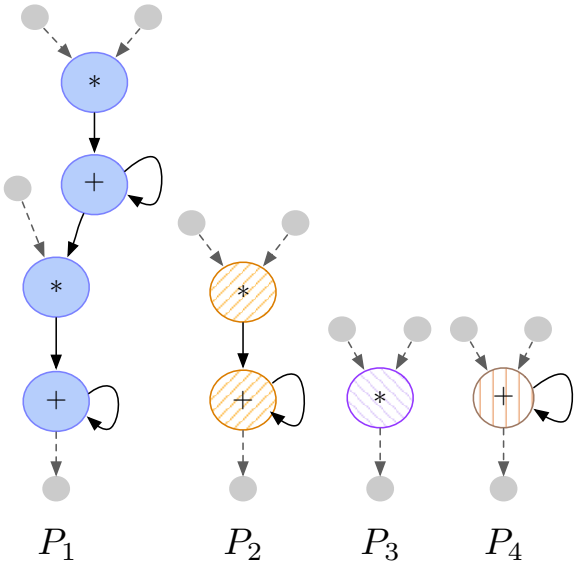
6.5 Exemple complet

Cette section illustre chaque étape de l'algorithme précédent au travers de l'exemple du triple produit matriciel. Après avoir identifié toutes les occurrences d'un ensemble de motifs dans le PRDG, on formule les contraintes de chaque macrobloc. Puis, la résolution du problème de couverture sélectionne deux macroblocs qui sont ordonnancés indépendamment. Le résultat final est un nouveau code C qui utilise les instructions spécialisées sélectionnées.

6.5.1 Identification et contraintes des macroblocs

On dispose en entrée de l'algorithme d'un ensemble de quatre motifs (cf. figure 6.11), ayant tous des occurrences dans le PRDG. Pour la clarté de l'exemple, le nombre de motifs est volontairement restreint et les constantes du PRDG original (cf. figure 6.3) ne sont pas prises en compte.

Les différentes occurrences de motifs dans le PRDG sont détaillées dans le tableau de la figure 6.11. Ainsi, le motif P_1 dispose de deux occurrences ($m_1 = \{n_1, n_2, n_5, n_6\}$ et $m_2 = \{n_3, n_4, n_5, n_6\}$). Ces occurrences sont exclusives puisque les nœuds $n5$ et $n6$ leur sont communs. D'autre part, les occurrences de m_6 à m_{11} sont des occurrences de taille unitaire et seront donc exécutées sur le processeur si elles sont sélectionnées.



	n_1	n_2	n_3	n_4	n_5	n_6
$m_1(P_1)$	*	*			*	*
$m_2(P_1)$			*	*	*	*
$m_3(P_2)$	*	*				
$m_4(P_2)$			*	*		
$m_5(P_2)$					*	*
$m_6(P_3)$	*					
$m_7(P_3)$			*			
$m_8(P_3)$					*	
$m_9(P_4)$		*				
$m_{10}(P_4)$				*		
$m_{11}(P_4)$						*

FIGURE 6.11 – Motifs disponibles et leurs occurrences dans le PRDG du triple produit matriciel.

Dans les paragraphes suivants, on détaille les contraintes associées à chaque occurrence de motif et le lecteur est invité à passer directement à la sous-section 6.5.2 s'il ne désire pas connaître les détails des contraintes.

Malgré l'existence d'un ordonnancement monodimensionnel légal, on cherche tout de même un ordonnancement multidimensionnel (trois dimensions) pour illustrer notamment les contraintes de vectorisation.

6.5.1.1 Contraintes des occurrences m_1 et m_2

Dans l'occurrence m_1, on identifie deux instructions spécialisées MAC[5]. En effet, les nœuds $n1$ et n_2 ainsi que les nœuds n_5 et n_6 forment deux groupes de nœuds qui contiennent une unique dépendance de données directe et ont le même domaine de définition ($\mathcal{D} = \{i, j, k | 0 \leq i, j, k \leq N\}$).

Les prototypes d'ordonnancement du canal $A_{n_5 \to n_4}$ et des instructions spécialisées IS_1 et IS_2 à la dimension p sont :

$$\theta(A, i, j, k)^p = \mu^p_{A1} i + \mu^p_{A2} j + \mu^p_{A3} k + \mu^p_{A4} N + \mu^p_{A5}$$

5. Multiplication Acumulation

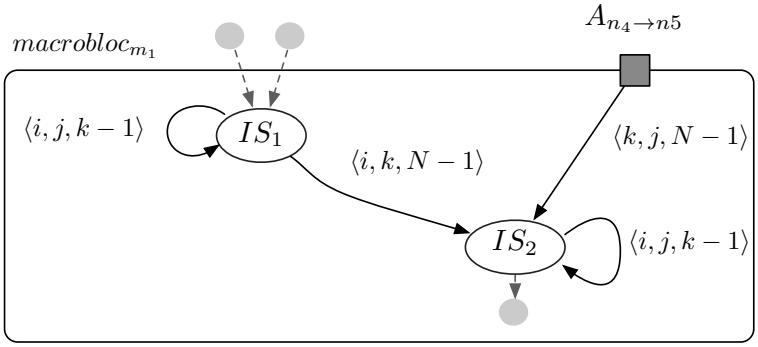


FIGURE 6.12 – Macrobloc de l'occurrence m_1.

$$\theta(is1, i, j, k)^p = \mu^p_{is11} i + \mu^p_{is12} j + \mu^p_{is13} k + \mu^p_{is14} N + \mu^p_{is15}$$
$$\theta(is2, i, j, k)^p = \mu^p_{is21} i + \mu^p_{is22} j + \mu^p_{is23} k + \mu^p_{is24} N + \mu^p_{is25}$$

Lecture dans le canal A On utilise la généralisation aux cas multidimensionnels de l'ordonnancement affine structuré pour énoncer les contraintes multidimensionnelles de la lecture de IS_2 dans le canal A :

1. La causalité de la communication pour la première dimension ($p = 1$) s'exprime sous la forme :

$$i\mu^1_{is21} + j(\mu^1_{is22} - \mu^1_{A2}) + k(\mu^1_{is23} - \mu^1_{A1}) + N(\mu^1_{is24} - \mu^1_{A3} - \mu^1_{A4}) + (\mu^1_{A3} + \mu^1_{is25} - \mu^1_{A5}) \geq \delta^1_0$$

On exprime le domaine de la dépendance sous forme matricielle pour identifier les multiplieurs de Farkas :

$$\begin{array}{l} \\ -i + N \geq 0 :: \lambda_1 \\ i \geq 0 :: \lambda_2 \\ -j + N \geq 0 :: \lambda_3 \\ j \geq 0 :: \lambda_4 \\ -k + N \geq 0 :: \lambda_5 \\ k \geq 0 :: \lambda_6 \end{array} \begin{array}{c} \begin{array}{cccccc} e & i & j & k & N & 1 \end{array} \\ \begin{pmatrix} 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

On en déduit le système :

$$\begin{cases} \mu^1_{is21} &= -\lambda_1 + \lambda_2 \\ \mu^1_{is22} - \mu^1_{A2} &= -\lambda_3 + \lambda_4 \\ \mu^1_{is23} - \mu^1_{A1} &= -\lambda_5 + \lambda_6 \\ \mu^1_{is24} - \mu^1_{A3} - \mu^1_{A4} &= \lambda_1 + \lambda_3 + \lambda_5 \\ \mu^1_{A3} + \mu^1_{is25} - \delta^1_0 - \mu^1_{A5} &= \lambda_0 \end{cases}$$

Après l'élimination des multiplieurs de Farkas, on obtient les contraintes :

$$\mathcal{D}^1_{A\to IS_2}\begin{cases} \mu^1_{A3}-\mu^1_{A5}+\mu^1_{is25}-\delta^1_0 \geq 0 \\ -\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A1}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is23}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A1}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is21}+\mu^1_{is23}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is21}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A2}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is21}+\mu^1_{is22}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A1}-\mu^1_{A2}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is21}+\mu^1_{is22}+\mu^1_{is23}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A1}-\mu^1_{A2}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is22}+\mu^1_{is23}+\mu^1_{is24} \geq 0 \\ -\mu^1_{A2}-\mu^1_{A3}-\mu^1_{A4}+\mu^1_{is22}+\mu^1_{is24} \geq 0 \end{cases}$$

2. La causalité de la communication pour la deuxième dimension ($p=2$) s'exprime sous la forme :

$$i\mu^2_{is21}+j(\mu^2_{is22}-\mu^2_{A2})+k(\mu^2_{is23}-\mu^2_{A1})+\mu^2_{is24}-\mu^2_{A3}-\mu^2_{A4})+(\mu^2_{A3}+\mu^2_{is25}-\mu^2_{A5}) \geq \delta^2_0-\delta^1_0(KN+K)$$

Après l'application de la forme affine du lemme de Farkas et élimination des multiplieurs, on obtient :

$$\mathcal{D}^2_{A\to IS_2}\begin{cases} \mu^2_{A3}-\mu^2_{A5}+\mu^2_{is25}+K\delta^1_0-\delta^2_0 \geq 0 \\ -\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A1}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is23}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A1}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is21}+\mu^2_{is23}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is21}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A2}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is21}+\mu^2_{is22}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A1}-\mu^2_{A2}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is21}+\mu^2_{is22}+\mu^2_{is23}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A1}-\mu^2_{A2}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is22}+\mu^2_{is23}+\mu^2_{is24}+K\delta^1_0 \geq 0 \\ -\mu^2_{A2}-\mu^2_{A3}-\mu^2_{A4}+\mu^2_{is22}+\mu^2_{is24}+K\delta^1_0 \geq 0 \end{cases}$$

3. La causalité de la communication pour la troisième dimension ($p=3$) s'exprime sous la forme :

$$\begin{aligned} i\mu^3_{is21}+j(\mu^3_{is22}-\mu^3_{A2})+k\mu^3_{is23}+N(\mu^3_{is24}-\mu^3_{A1}-\mu^3_{A3}-\mu^3_{A4}) \\ +\mu^3_{A1}+\mu^3_{A3}+\mu^3_{is25}-\mu^3_{A5}) \geq \delta^3_0-(\delta^1_0+\delta^2_0)(KN+K) \end{aligned}$$

Après l'application de la forme affine du lemme de Farkas et élimination des multiplieurs on obtient :

$$\mathcal{D}^3_{A\to IS_2}\begin{cases} \mu^3_{A1}+\mu^3_{A3}-\mu^3_{A5}+\mu^3_{is25}+K\delta^1_0+K\delta^2_0-\delta^3_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is23}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is21}+\mu^3_{is23}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is21}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A2}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is21}+\mu^3_{is22}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A2}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is21}+\mu^3_{is22}+\mu^3_{is23}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A2}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is22}+\mu^3_{is23}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0\\ -\mu^3_{A1}-\mu^3_{A2}-\mu^3_{A3}-\mu^3_{A4}+\mu^3_{is22}+\mu^3_{is24}+K\delta^1_0+K\delta^2_0 \geq 0 \end{cases}$$

4. Il n'y a qu'une seule dimension qui satisfait fortement la dépendance :

$$\delta^1_0+\delta^2_0+\delta^3_0=1$$

Dépendance interne $e_{IS_1\to IS_1}$ Après avoir appliqué la forme affine du lemme de Farkas à la causalité de la dépendance $e_{IS_1\to IS_1}$ pour les deux dimensions, on obtient :

$$\mathcal{D}^1_{IS_1\to IS_1}:\ \mu^1_{is13}-\delta^1_1 \geq 0$$

$$\mathcal{D}^2_{IS_1\to IS_1}:\ \mu^2_{is13}+K\delta^1_1-\delta^2_1 \geq 0$$

$$\mathcal{D}^3_{IS_1\to IS_1}:\ \mu^3_{is13}+K\delta^1_1+K\delta^2_1-\delta^3_1 \geq 0$$

Dépendance interne $e_{IS_2\to IS_2}$ On procède de même pour la causalité de la dépendance $e_{IS_2\to IS_2}$:

$$\mathcal{D}^1_{IS_2\to IS_2}:\ \mu^1_{is23}-\delta^1_3 \geq 0$$

$$\mathcal{D}^2_{IS_2\to IS_2}:\ \mu^2_{is23}+K\delta^1_3-\delta^2_3 \geq 0$$

$$\mathcal{D}^3_{IS_2\to IS_2}:\ \mu^3_{is23}+K\delta^1_3+K\delta^2_3-\delta^3_3 \geq 0$$

Dépendance interne $e_{IS_1\to IS_2}$ La dépendance $e_{IS_1\to IS_2}$ est associée à une mémorisation sur l'extension matérielle des résultats produits par IS_1. Ils sont utilisés par l'instruction spécialisée IS_2 sans nécessiter d'accès à la mémoire du processeur. Les contraintes du respect de la causalité de la

dépendance pour un ordonnancement multidimensionnel sont :

$$\mathcal{D}^1_{IS_1\to IS_2} : \begin{cases} \mu^1_{is13} - \mu^1_{is15} + \mu^1_{is25} - \delta^1_2 \geq 0 \\ -\mu^1_{is13} - \mu^1_{is14} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is12} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is23} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is11} - \mu^1_{is12} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is21} + \mu^1_{is23} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is11} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is21} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is11} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is21} + \mu^1_{is22} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is11} - \mu^1_{is12} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is21} + \mu^1_{is22} + \mu^1_{is23} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is12} - \mu^1_{is13} - \mu^1_{is14} + \mu^1_{is22} + \mu^1_{is23} + \mu^1_{is24} \geq 0 \\ -\mu^1_{is13} - \mu^1_{is14} + \mu^1_{is22} + \mu^1_{is24} \geq 0 \end{cases}$$

$$\mathcal{D}^2_{IS_1\to IS_2} : \begin{cases} \mu^2_{is13} - \mu^2_{is15} + \mu^2_{is25} + K\delta^1_2 - \delta^2_2 \geq 0 \\ -\mu^2_{is13} - \mu^2_{is14} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is12} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is23} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is11} - \mu^2_{is12} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is21} + \mu^2_{is23} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is11} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is21} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is11} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is21} + \mu^2_{is22} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is11} - \mu^2_{is12} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is21} + \mu^2_{is22} + \mu^2_{is23} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is12} - \mu^2_{is13} - \mu^2_{is14} + \mu^2_{is22} + \mu^2_{is23} + \mu^2_{is24} + K\delta^1_2 \geq 0 \\ -\mu^2_{is13} - \mu^2_{is14} + \mu^2_{is22} + \mu^2_{is24} + K\delta^1_2 \geq 0 \end{cases}$$

$$\mathcal{D}^3_{IS_1\to IS_2} : \begin{cases} \mu^3_{is12} + \mu^3_{is13} - \mu^3_{is15} + \mu^3_{is25} + K\delta^1_2 + K\delta^2_2 - \delta^3_2 \geq 0 \\ -\mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is23} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is11} - \mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is21} + \mu^3_{is23} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is11} - \mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is21} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is11} - \mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is21} + \mu^3_{is22} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is11} - \mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is21} + \mu^3_{is22} + \mu^3_{is23} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is22} + \mu^3_{is23} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \\ -\mu^3_{is12} - \mu^3_{is13} - \mu^3_{is14} + \mu^3_{is22} + \mu^3_{is24} + K\delta^1_2 + K\delta^2_2 \geq 0 \end{cases}$$

Instructions spécialisées vectorisables Les instructions spécialisées sont vectorisables si la fonction d'ordonnancement de leur dimension la plus interne ne dépend pas des indices de leur espace d'itérations respectifs. Les contraintes de vectorisation pour IS_1 et IS_2 sont donc :

$$\mathcal{DV}_{IS_1} : \mu^3_{is11} = \mu^3_{is12} = \mu^3_{is13} = 0$$

$$\mathcal{DV}_{IS_2} : \mu^3_{is21} = \mu^3_{is22} = \mu^3_{is23} = 0$$

Intersection de toutes les contraintes de m_1 L'ensemble des contraintes du macrobloc m_1 constitue le polyèdre $\mathcal{D}_{m_1}$:

$$\mathcal{D}_{m_1} = \mathcal{DV}_{IS_1} \cap \mathcal{DV}_{IS_2} \bigcap_{p\in[1,3]} (\mathcal{D}^p_{A\to IS_2} \cap \mathcal{D}^p_{IS_1\to IS_1} \cap \mathcal{D}^p_{IS_2\to IS_2} \cap \mathcal{D}^p_{IS_1\to IS_2})$$

Contraintes modulaires de m_1 Après avoir éliminé les coefficients internes du macrobloc m_1, on obtient les contraintes modulaires suivantes :

$$\mathcal{DM}_{m_1} : \begin{cases} -1 + \delta^1_0 + \delta^2_0 + \delta^3_0 = 0 \\ 10 + K\delta^1_0 - \mu^2_{A1} - \mu^2_{A3} - \mu^2_{A4} \geq 0 \\ 10 + K\delta^1_0 + \mu^2_{A1} + \mu^2_{A3} - \mu^2_{A5} - \delta^2_0 \geq 0 \\ 10 + \mu^1_{A1} + \mu^1_{A3} - \mu^1_{A5} - \delta^1_0 \geq 0 \\ 9 + K\delta^1_0 + K\delta^2_0 + \mu^3_{A1} + \mu^3_{A3} - \mu^3_{A5} \geq 0 \\ 10 + K\delta^1_0 + K\delta^2_0 - \mu^3_{A1} - \mu^3_{A2} - \mu^3_{A3} - \mu^3_{A4} \geq 0 \end{cases}$$

Les constantes 9 et 10 proviennent des bornes des coefficients d'ordonnancement utilisées ($0 \leq \mu_k \leq 10$). En effet, pour pouvoir utiliser la nullification, il est nécessaire de borner les coefficients d'ordonnancement. Ces bornes ne sont pas présentées dans les contraintes précédentes pour des raisons de lisibilité.

D'autre part, la construction des contraintes modulaires du macrobloc m_2 est similaire et n'est pas détaillée ici.

6.5.1.2 Contraintes des occurrences m_3 et m_4

Les macroblocs m_3 et m_4 (cf. figure 6.14) ont le même comportement. Ils contiennent une unique instruction spécialisée qui écrit dans un canal ($B_{n_4\to n_5}$ dans le cas de m_4).

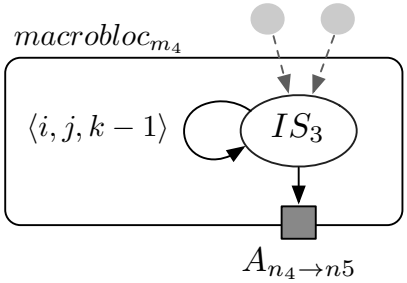


FIGURE 6.13 – Macrobloc de l'occurrence m_4.

Écriture dans le canal A Il n'y a pas de causalité dans le cas d'une écriture dans un canal (cf. contrainte 6.2), aucune nullification n'est donc nécessaire. Après la projection des multiplieurs de

Farkas et quelle que soit la dimension $p \in [1,3]$ on a :

$$\begin{cases} -\mu^p_{is35} + \mu^p_{A5} \geq 0 \\ -\mu^p_{is34} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is33} - \mu^p_{is34} + \mu^p_{A3} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is31} - \mu^p_{is33} - \mu^p_{is34} + \mu^p_{A1} + \mu^p_{A3} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is31} - \mu^p_{is34} + \mu^p_{A1} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is31} - \mu^p_{is32} - \mu^p_{is34} + \mu^p_{A1} + \mu^p_{A2} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is31} - \mu^p_{is32} - \mu^p_{is33} - \mu^p_{is34} + \mu^p_{A1} + \mu^p_{A2} + \mu^p_{A3} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is32} - \mu^p_{is33} - \mu^p_{is34} + \mu^p_{A2} + \mu^p_{A3} + \mu^p_{A4} \geq 0 \\ -\mu^p_{is32} - \mu^p_{is34} + \mu^p_{A2} + \mu^p_{A4} \geq 0 \end{cases}$$

Dépendance interne $e_{IS_3 \to IS_3}$ Les contraintes de cette dépendance sont similaires à celles de $e_{IS_1 \to IS_1}$ et $e_{IS_2 \to IS_2}$ détaillées pour le macrobloc m_1, seuls les coefficients changent.

Instructions spécialisées vectorisables Les contraintes de vectorisation pour IS_3 sont :

$$\mathcal{DV}_{IS_3} : \mu^3_{is31} = \mu^3_{is32} = \mu^3_{is33} = 0$$

Intersection de toutes les contraintes de m_4 L'ensemble des contraintes du macrobloc m_4 constitue le polyèdre $\mathcal{D}_{m_4}$:

$$\mathcal{D}_{m_4} = \mathcal{DV}_{IS_1 - 3} \bigcap_{p \in [1,3]} (\mathcal{D}^p_{IS_3 \to A} \cap \mathcal{D}^p_{IS_3 \to IS_3})$$

Contraintes modulaires de m_4 Après avoir éliminé les coefficients internes du macrobloc m_4, on obtient la contrainte modulaire suivante :

$$\mathcal{DM}_{m_4} : \quad -1 + K\mu^1_{A3} + K\mu^1_{A4} + \mu^2_{A3} + \mu^2_{A4} \geq 0$$

6.5.1.3 Contraintes de l'occurrence m_5

Dans l'occurrence m_5, on identifie une unique instruction spécialisée (IS_4). Celle-ci lit dans deux canaux de communication ($A_{n_4 \to n_5}$ et $B_{n_2 \to n_5}$). Le macrobloc contient également une dépendance interne cyclique.

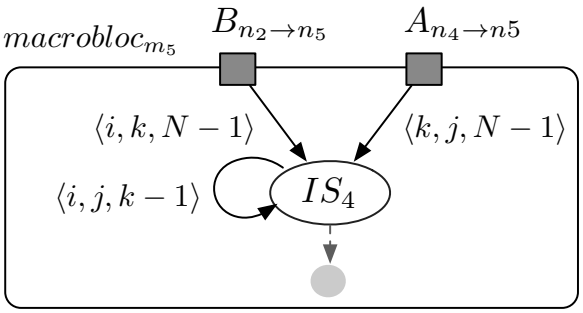


FIGURE 6.14 – Macrobloc de l'occurrence m_5.

Lecture dans le canal A On utilise la généralisation aux cas multidimensionnels de l'ordonnancement affine structuré pour énoncer les contraintes multidimensionnelles de la lecture de IS_3 dans le canal A. Les contraintes obtenues sont identiques à celles de la lecture de A dans le macrobloc m_1, il suffit de changer les coefficients de IS_1 en ceux de IS_4.

Lecture dans le canal B En suivant la même méthode que pour la lecture dans A de m_1, on obtient les contraintes suivantes :

$$\mathcal{D}^1_{B\to IS_4}\begin{cases} \mu^1_{B3}-\mu^1_{B5}+\mu^1_{is45}-\delta^1_2 \geq 0\\ -\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is44} \geq 0\\ -\mu^1_{B2}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is43}+\mu^1_{is44} \geq 0\\ -\mu^1_{B1}-\mu^1_{B2}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is41}+\mu^1_{is43}+\mu^1_{is44} \geq 0\\ -\mu^1_{B1}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is41}+\mu^1_{is44} \geq 0\\ -\mu^1_{B1}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is41}+\mu^1_{is42}+\mu^1_{is44} \geq 0\\ -\mu^1_{B1}-\mu^1_{B2}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is41}+\mu^1_{is42}+\mu^1_{is43}+\mu^1_{is44} \geq 0\\ -\mu^1_{B2}-\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is42}+\mu^1_{is43}+\mu^1_{is44} \geq 0\\ -\mu^1_{B3}-\mu^1_{B4}+\mu^1_{is42}+\mu^1_{is44} \geq 0 \end{cases}$$

$$\mathcal{D}^2_{B\to IS_4}\begin{cases} \mu^2_{B3}-\mu^2_{B5}+\mu^2_{is45}+K\delta^1_2-\delta^2_2 \geq 0\\ -\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B2}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is43}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B1}-\mu^2_{B2}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is41}+\mu^2_{is43}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B1}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is41}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B1}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is41}+\mu^2_{is42}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B1}-\mu^2_{B2}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is41}+\mu^2_{is42}+\mu^2_{is43}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B2}-\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is42}+\mu^2_{is43}+\mu^2_{is44}+K\delta^1_2 \geq 0\\ -\mu^2_{B3}-\mu^2_{B4}+\mu^2_{is42}+\mu^2_{is44}+K\delta^1_2 \geq 0 \end{cases}$$

$$\mathcal{D}^3_{B\to IS_4}\begin{cases} \mu^3_{B3}-\mu^3_{B5}+\mu^3_{is45}+K\delta^1_2+K\delta^2_2-\delta^3_2 \geq 0\\ -\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B2}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is43}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B1}-\mu^3_{B2}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is41}+\mu^3_{is43}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B1}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is41}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B1}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is41}+\mu^3_{is42}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B1}-\mu^3_{B2}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is41}+\mu^3_{is42}+\mu^3_{is43}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B2}-\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is42}+\mu^3_{is43}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0\\ -\mu^3_{B3}-\mu^3_{B4}+\mu^3_{is42}+\mu^3_{is44}+K\delta^1_2+K\delta^2_2 \geq 0 \end{cases}$$

Dépendance interne $e_{IS_4\to IS_4}$ Les contraintes de cette dépendance sont similaires à celles de $e_{IS_1\to IS_1}$, $e_{IS_2\to IS_2}$ et $e_{IS_2\to IS_3}$ détaillées précédemment, seuls les coefficients changent.

Instructions spécialisées vectorisables Les contraintes de vectorisation pour IS_4 sont :

$$\mathcal{DV}_{IS_4} : \mu^3_{is41}=\mu^3_{is42}=\mu^3_{is43}=0$$

Intersection de toutes les contraintes de m_5 L'ensemble des contraintes du macrobloc m_5 constitue le polyèdre $\mathcal{D}_{m_5}$:

$$\mathcal{D}_{m_5} = \mathcal{DV}_{IS_4} \bigcap_{p\in[1,3]} (\mathcal{D}^p_{A\to IS_4}\mathcal{D}^p_{B\to IS_4} \cap \mathcal{D}^p_{IS_4\to IS_4})$$

Contraintes modulaires de m_5 Après avoir éliminé les coefficients internes du macrobloc m_5, on obtient les contraintes modulaires suivantes :

$$\mathcal{DM}_{m_5} : \begin{cases} -1+\delta^1_2+\delta^2_2+\delta^3_2 = 0 \\ -1+\delta^1_0+\delta^2_0+\delta^3_0 = 0 \\ 10-\mu^1_{A3}-\mu^1_{A4} \geq 0 \\ 1-\delta^1_2-\delta^2_2 \geq 0 \\ 10+K\delta^1_0+K\delta^2_0-\mu^3_{A1}-\mu^3_{A2}-\mu^3_{A3}-\mu^3_{A4} \geq 0 \\ 10+\mu^1_{A3}-\mu^1_{A5}-\delta^1_0 \geq 0 \\ 10+K\delta^1_0-\mu^2_{A3}-\mu^2_{A4} \geq 0 \\ 9+K\delta^1_0+K\delta^2_0+\mu^3_{A3}-\mu^3_{A5} \geq 0 \\ 10-\mu^1_{B3}-\mu^1_{B4} \geq 0 \\ 10+\mu^1_{B3}-\mu^1_{B5}-\delta^1_2 \geq 0 \\ 10+K\delta^1_2+K\delta^2_2-\mu^3_{B1}-\mu^3_{B2}-\mu^3_{B3}-\mu^3_{B4} \geq 0 \\ 10+K\delta^1_0+\mu^2_{A3}-\mu^2_{A5}-\delta^2_0 \geq 0 \\ 9+K\delta^1_2+K\delta^2_2+\mu^3_{B3}-\mu^3_{B5} \geq 0 \\ 10+K\delta^1_2+\mu^2_{B3}-\mu^2_{B5}-\delta^2_2 \geq 0 \\ 1-\delta^1_0-\delta^2_0 \geq 0 \\ 10+K\delta^1_2-\mu^2_{B3}-\mu^2_{B4} \geq 0 \end{cases}$$

De même que pour m_1, les constantes 10 et 9 proviennent de la borne maximale des coefficients d'ordonnancement utilisée ($0 \leq \mu_k \leq 10$). Ces bornes ne sont pas présentées dans les contraintes précédentes pour des raisons de lisibilité.

6.5.1.4 Contraintes des occurrences de m_6 à m_{11}

Ces occurrences sont issues des motifs de taille unitaire et seront donc exécutées sur le processeur : aucune contrainte de vectorisation n'est imposée. Mis à part cette particularité, la construction des contraintes de leurs processus respectifs est identique aux occurrences précédentes.

6.5.2 Couverture et ordonnancement du PRDG

Une fois les contraintes affines modulaires de chaque occurrence construites, on énonce un CSP contenant les contraintes de couverture ainsi qu'une contrainte polyédrique pour chaque occurrence :

$$\forall m_k \in M = \{m_1, \ldots, m_{16}\}, PolyCP(\mathcal{DM}_{m_k} \cup sel_{m_k} = 0)$$

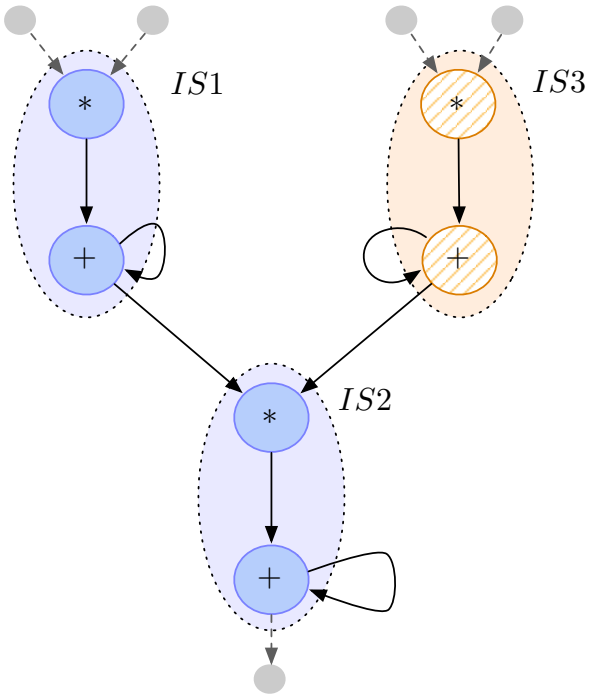


FIGURE 6.15 – Couverture du PRDG du triple produit matriciel : deux macroblocs sont sélectionnés. De plus, trois instructions spécialisées ont été identifiées et ordonnancées.

On résout ensuite le CSP en minimisant le nombre d'occurrences sélectionnées :

$$nbOccurrences = \sum_{k=1}^{k=16} sel_{m_k}$$

Dans cet exemple, il est évident que seulement deux occurrences suffisent pour couvrir le PRDG. La figure 6.15 illustre une des deux couvertures possibles (c-à-d., m_1, m_4 ou m_2, m_3) qui minimisent le nombre d'occurrences sélectionnées. Après résolution du CSP, on obtient une solution qui sélectionne, par exemple, les macroblocs m_1 et m_4. La solution détermine également un ordonnancement légal du canal A : $\mu^1_{A4} = \delta^1_0 = 1$. Tous les autres coefficients sont nuls.

La couverture sélectionnée forme un réseau de deux macroblocs communiquant par le canal A. Il suffit d'injecter l'ordonnancement de A (appelé $Dcom_A$), obtenu lors de la résolution du CSP précédent, pour ordonnancer séparément chaque macrobloc :

1. Macrobloc m_1. Le domaine indépendant d'ordonnancement de m_1 est obtenu à partir des contraintes construites avant d'éliminer les coefficients des communications :

$$\mathcal{DS}_{m_1} = \mathcal{D}_{m_1} \cap Dcom_A$$

L'élimination des coefficients d'ordonnancement du canal A forme un ensemble de contraintes supplémentaires pour un algorithme d'ordonnancement multidimensionnel standard. Une solution possible est :

$$\mu^1_{is23} = \mu^1_{is24} = \mu^1_{is25} = 1,\ \mu^1_{is13} = 1,\ \delta^1_1 = \delta^1_2 = \delta^1_3 = 1$$

Le reste des coefficients étant nuls, on en déduit les ordonnancements des instructions spécialisées contenues dans le macrobloc m_1 :

$$\theta_{IS_1}(i,j,k) = (k,0,0)$$

$$\theta_{IS_2}(i,j,k) = (k+N+1,0,0)$$

2. Macrobloc m_4. Le domaine indépendant d'ordonnancement de m_4 est :

$$DS_{m_4} = \mathcal{D}_{m_4} \cap Dcom_A$$

Une solution possible est :

$$\mu^1_{is33} = 1,\ \delta^1_5 = 1$$

Le reste des coefficients étant nul, on en déduit l'ordonnancement de l'unique instruction spécialisée contenue dans le macrobloc m_4 :

$$\theta_{IS_3}(i,j,k) = (k,0,0)$$

6.5.3 Génération du code spécialisé

Les instructions spécialisées sélectionnées et ordonnancées sont utilisées par la dernière étape de génération de code. Le code C produit en sortie est présenté dans la figure 6.16. À chaque instruction spécialisée correspond une fonction dont le comportement est une séquence d'instructions spécialisées primitives analysable par le compilateur natif du processeur extensible.

L'instruction spécialisée IS_1 réduit la pression sur la mémoire du processeur en utilisant la mémoire de l'extension. À l'inverse, l'instruction IS_3 n'étant pas dans le même macrobloc que l'instruction IS_2, elle requiert une communication avec le processeur. Ainsi, à chaque itération, le résultat produit par IS_2 est enregistré dans le tableau F.

```
1  for (c1=0;c1<N;c1++) {
2     for (i=0;i<N;i++) { /*parallel*/
3       for (j=0;j<N;j++) { /*parallel*/
4         custom_IS1(A[i][c1],B[c1][j]);
5         custom_IS3(D[i][c1],E[c1][j],&F[i][j]);
6       }
7     }
8   }
9   for (c1=N+1;c1<2*N+1;c1++) {
10    for (i=0;i<N;i++) { /*parallel*/
11      for (j=0;j<N;j++) { /*parallel*/
12        custom_IS2(F[c1-N-1][j],&G[i][j]);
13      }
14    }
15  }
16 }
```

FIGURE 6.16 – Code C du triple produit matriciel utilisant les trois instructions spécialisées sélectionnées.

D'autre part, les trois instructions spécialisées sont exécutées dans des nids de boucles dont les deux dimensions internes sont parallèles. Le code original a été transformé pour qu'elles soient toutes vectorisables.

6.5.4 Validation expérimentale

Afin d'étudier expérimentalement l'impact de la spécialisation et de la transformation du code pour le triple produit matriciel, nous avons simulé le comportement de différentes possibilités de programmes à l'aide de la plateforme de prototypage virtuel SoCLib [6] [169] (simulation précise au cycle et au bit près).

- programme original : aucune modification n'est faite sur le programme.
- instructions MAC : utilisation, dans chaque nid de boucles, d'une ISE réalisant une multiplication et une accumulation en un seul cycle. Cette version du programme correspond au résultat produit par un flot classique d'extension de jeu d'instructions.
- instructions MAC + mémoire : résultat produit par notre approche conjointe de transformation de code et d'extension de jeux d'instructions. La dimension la plus interne de chaque nid de boucles est parallèle et les ISE pourraient donc être vectorisées. Cependant, nous nous contentons ici de mesurer l'impact de la mémoire déportée sur l'extension qui temporise les données produites par l'IS1 et consommées par l'IS2.

La plateforme de simulation est un système complet composé d'un processeur NiosII (version *Fast*) avec un cache d'instructions séparé du cache de données, d'une mémoire ainsi que des périphériques d'entrées-sorties. L'extension matérielle du NiosII supporte l'instruction MAC et utilise éventuellement une mémoire (cas de IS1 et IS2). Chacune des versions du programme est compilée pour le processeur NiosII (gcc nios2 4.4.4 -O2) couplé à une extension matérielle. L'exploration consiste à faire varier la taille du cache de données entre 1 ko et 2048 ko. L'objectif est alors d'étudier le comportement de chacune des versions du programme en fonction de la taille du cache et de la taille des matrices.

La figure 6.17 résume les accélérations obtenues pour les deux versions de programmes utilisant des instructions spécialisées. Il y apparaît que la version utilisant une mémoire interne est généralement deux fois plus rapide que celle qui n'utilise que des instructions MAC. Une autre observation intéressante est que l'accélération obtenue par les instructions MAC est finalement négligeable dans le cas où les caches ont une taille importante (e.g., 32 ko et 2048 ko) et ce quelle que soit la taille des matrices. Dans de telles conditions, ce sont les lectures dans le cache qui pénalisent le plus les performances, l'écriture dans la mémoire qui est évitée par l'accumulation ne suffit pas à accélérer notablement l'application puisque le gain est d'environ 1,2 par rapport à la version normale du programme.

6. SoCLib est une bibliothèque de modèles de simulation de composants matériels écrits en SystemC

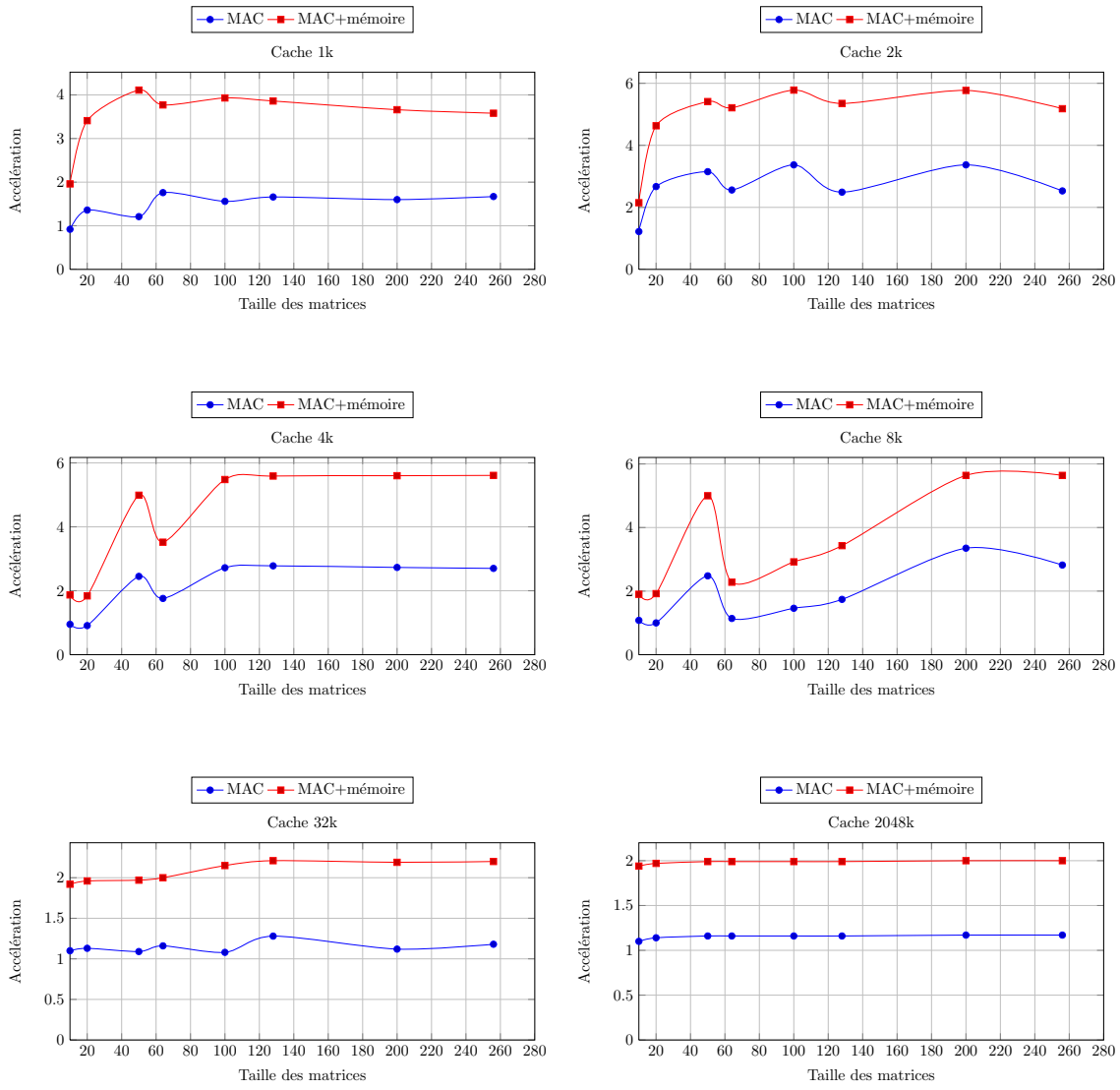


FIGURE 6.17 – Accélération obtenue en fonction de la taille des matrices.

Les figures 6.18, 6.19 et 6.20 illustrent l'influence de la taille du cache pour, respectivement, la version originale du programme, celle qui utilise les MAC et celle qui utilise les MAC ainsi qu'une mémorisation sur l'extension. Chaque barre indique l'accélération obtenue par rapport à un cache de 1 ko pour une taille de cache et une taille de matrices données. Dans le cas du programme original (cf. figure 6.18), le dimensionnement du cache a un impact important sur les performances puisque, par exemple, pour une taille de matrice égale à 50, les performances sont multipliées par 2,5 pour un cache supérieur ou égal à 32 ko. L'influence de la taille du cache reste importante pour la version utilisant des MAC (cf. figure 6.19) et peut également améliorer les performances d'un facteur 2,5 par rapport à un cache de 1 ko. Par contre, il est intéressant de noter que pour le code produit par notre approche, l'impact du dimensionnement du cache est négligeable et ce quelle que soit la taille des matrices. La pression mémoire sur le cache du processeur est donc considérablement diminuée par l'utilisation de la mémoire de l'extension et par le nouvel ordonnancement affine des accès mémoires.

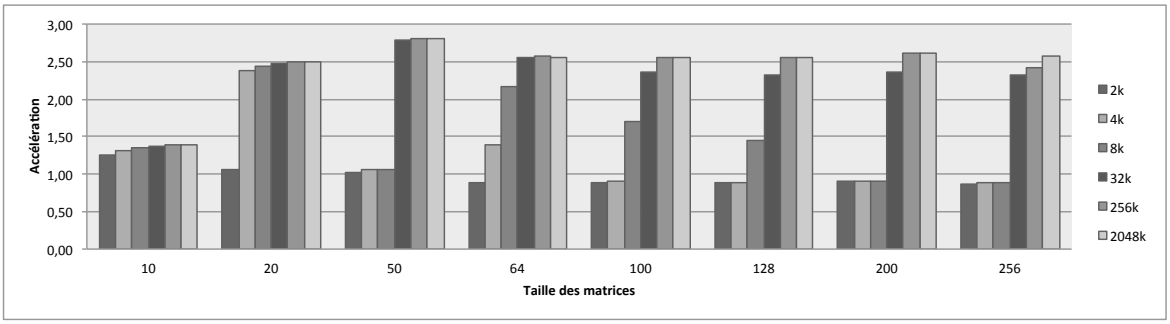


FIGURE 6.18 – Accélération relative par rapport à un cache 1 ko (programme original).

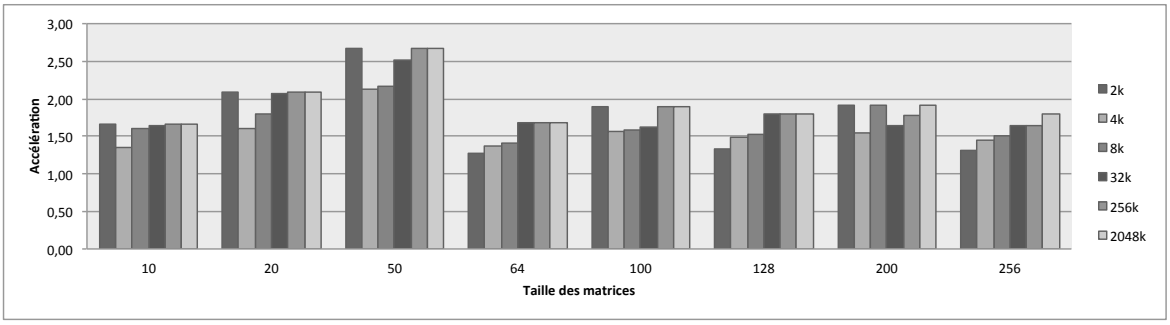


FIGURE 6.19 – Accélération relative par rapport à un cache 1 ko (instructions MAC).

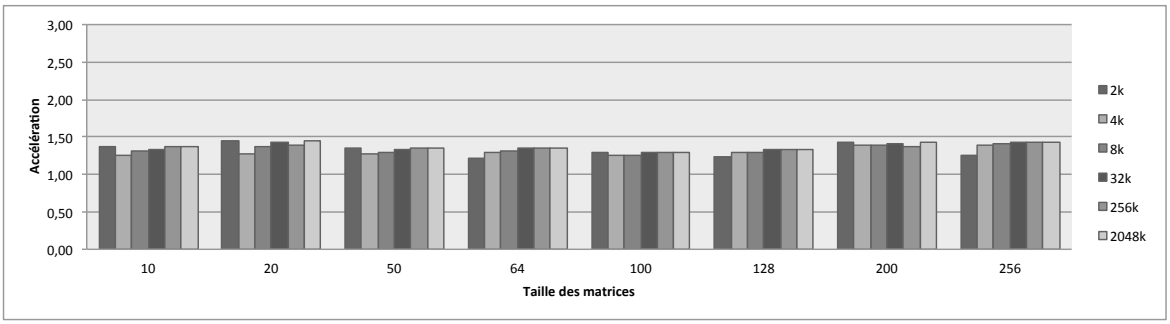


FIGURE 6.20 – Accélération relative par rapport à un cache 1 ko (instructions MAC + mémoire).

6.6 Travaux liés

Dans la majorité des approches existantes sur l'extension de jeu d'instructions, la sélection d'instructions spécialisées est la dernière étape du flot d'optimisation d'un compilateur. Quelques travaux s'intéressent néanmoins à l'impact de ces optimisations sur la qualité des instructions spécialisées sélectionnées. Ainsi, Bonzini et Pozzi [27] explorent différentes possibilités de déroulage de boucles et de *if-conversion* pour exposer une forme intermédiaire sujette à de meilleures instructions spécialisées. Les résultats expérimentaux obtenus justifient le fait que les stratégies d'optimisation d'un compilateur gagnent à être modifiées pour améliorer la qualité des instructions spécialisées sélectionnées. Ce constat a notamment motivé les travaux de Benett [21] qui utilise un algorithme probabiliste [61]

combiné à un apprentissage pour explorer de multiples compositions de transformations allant jusqu'à cinquante étapes successives.

Notre approche se démarque de ces travaux de la même manière que les transformations dans le modèle polyédrique se différencient des flots itératifs d'optimisation. En effet, les ordonnancements affines nous permettent de modéliser de multiples compositions complexes de transformations de boucle par la simple modification des coefficients d'ordonnancement d'instructions spécialisées candidates.

L'approche proposée par Trifunovic et. al [182] transforme automatiquement un nid de boucle afin de faire apparaître des instructions vectorisées (pour un facteur de vectorisation donné) et minimise la durée d'exécution totale en tenant compte du surcoût des alignements mémoire issus de la vectorisation. L'algorithme est itératif et utilise les transformations affines pour faire apparaître les vectorisations qui minimisent la fonction de coût. L'approche n'est applicable qu'à un code où les tailles des tableaux sont connues et où les espaces d'itération ne sont pas paramétrés. Cette approche n'est pas à proprement parlé liée à l'extension de jeu d'instructions mais constitue une piste intéressante pour guider finement la vectorisation dans notre approche.

L'exploration itérative de l'espace des ordonnancements affines légaux d'un programme, proposé dans la méthode de Pouchet et. al [148, 147] (cf. sections 4.3.2 et 4.3.3), se rapproche également de notre problématique. En effet, nous explorons également l'espace des ordonnancements légaux. Cependant, dans notre cas, cette exploration est faite par l'algorithme de résolution du CSP car la fonction de coût est connue : il s'agit de favoriser les performances en maximisant l'utilisation de l'extension tout en respectant des contraintes supplémentaires (e.g., pour la vectorisation).

6.7 Conclusion et perspectives

Dans ce chapitre, nous avons proposé une approche originale qui utilise l'expressivité du modèle polyédrique pour coupler les transformations de boucles à l'extension de jeux d'instructions. Contrairement aux techniques existantes, c'est la sélection des instructions spécialisées qui guide la transformation du code. Il est ainsi possible de sélectionner des regroupements d'opérations se trouvant pourtant, à l'origine, dans des blocs de base différents.

La généralisation des ordonnancements affines modulaires aux cas multidimensionnels nous permet d'exprimer les contraintes de légalité pour chaque occurrence candidate à la sélection. Cet aspect modulaire offre la possibilité d'ajouter des contraintes supplémentaires afin de répondre à des exigences de performance (i.e., instructions vectorisables) ou encore à des limitations architecturales.

L'extensibilité de notre algorithme constitue un terrain propice à de futurs travaux. Pour le moment, nous imposons que toutes les instructions spécialisées contenues dans un macrobloc soient vectorisables. De plus, on considère que les instructions d'un macrobloc communiquent entre elles par une mémoire embarquée sur l'extension matérielle. Cependant, la gestion d'une taille restreinte pour ces mémoires embarquées correspond à résoudre un problème conjoint d'ordonnancement affine et de contraction mémoire.

Il s'agit d'un problème extrêmement difficile et qui reste, à notre connaissance, ouvert pour des ordonnancements multidimensionnels. Il existe néanmoins des solutions dans le cas monodimensionnel. Ainsi, Thies [180] se base sur la notion de vecteur d'occupation pour contraindre les ordonnancements légaux en fonction d'une allocation mémoire donnée. Cependant, il n'est pas évident que cette ap-

proche soit compatible avec la modularité requise par notre approche. Feautrier [57] propose, dans le cadre d'un ordonnancement modulaire monodimensionnel, d'effectuer une contraction de la première dimension d'un tableau dans une mémoire dont la taille est bornée. Le modèle de mémoire utilisé est un *buffer* circulaire qui permet de lier la date d'allocation d'une donnée dans la mémoire et celle de sa destruction. Si cette approche borne simplement la taille de la mémoire utilisée par un canal de communication, nous n'avons pas réussi à la généraliser aux cas multidimensionnels.

D'autre part, nous n'avons pas encore pu mesurer expérimentalement la pertinence de notre approche. En effet, les gains attendus se situent principalement au niveau de la localité des données et de l'exploitation du parallélisme des instructions spécialisées. Les perspectives à plus court terme de ces travaux sont donc de générer une description matérielle de l'extension et de mettre en œuvre une étape de vectorisation.

Troisième partie

Intégration de méthodologies logicielles dans la conception d'outils pour la compilation optimisante

Chapitre 7

Compilation et ingénierie dirigée par les modèles

Comme évoqué dans les chapitres 1 et 3, l'extension du jeu d'instructions soulève de nombreux défis qui placent le compilateur au cœur du processus de conception. De plus, dans ce contexte, la compilation est caractérisée par différentes préoccupations (e.g., sélection du jeu d'instructions, génération de code, ordonnancement, synthèse matérielle, etc.) qui favorisent la multiplication des représentations intermédiaires. Dans ce chapitre, nous nous intéressons aux moyens d'alléger les différentes tâches logicielles d'un compilateur optimisant par l'intégration de méthodologies issues de l'ingénierie dirigée par les modèles.

Sommaire

7.1 Introduction

Les recherches sur l'ingénierie dirigée par les modèles (IDM) ont pour principal objectif de réduire la complexité apparaissant *accidentellement* lors du développement de systèmes logiciels complexes [60]. Elles reposent sur une description logicielle abstraite associée à des technologies d'analyse rigoureuses de transformation en implémentations concrètes [166]. Au cœur de l'IDM, se trouve la notion de *modèle* et de *métamodèle*. Un modèle est une abstraction d'une entité concrète réalisée dans une intention particulière. Un modèle ne représente donc pas l'intégralité d'une entité, mais uniquement un sous ensemble de ses caractéristiques et comportements permettant de répondre à un problème spécifique. Un métamodèle est un langage de modélisation offrant une abstraction commune pour l'ensemble des modèles répondant à la même intention. Les métamodèles sont décrits dans un unique métalangage tel que le *Meta-Object Facility*(MOF) de l'OMG. Les concepteurs peuvent utiliser ces langages de modélisation pour décrire des logiciels complexes à différents niveaux d'abstraction et pour de multiples perspectives. L'IDM s'intéresse principalement à la transformation des descriptions d'artefacts logiciels en d'autres formes plus adaptées à chaque besoin spécifique. Par exemple, les techniques de l'IDM peuvent être utilisées pour transformer un modèle décrit dans le langage de modélisation unifié (UML) [179] en un programme Java compilable et exécutable. Un autre exemple consiste à transformer la description abstraite d'un logiciel en un modèle permettant d'évaluer les performances et la qualité de conception de ce logiciel.

De prime abord, les chercheurs de l'IDM et de la compilation optimisante s'intéressent à des problèmes radicalement différents. Cependant, les technologies matures de l'IDM offrent des facilités particulièrement pertinentes pour des environnements de compilation axés sur la recherche. Ceux-ci utilisent des représentations intermédiaires raffinées, par des transformations successives, en des formes plus efficaces pour l'architecture matérielle ciblée. Ils sont donc, par essence, proches des problématiques de l'IDM. La structure d'une représentation intermédiaire (RI) peut alors être exprimée sous la forme d'un métamodèle dont les instances (modèles) correspondent aux représentations intermédiaires de différentes versions, éventuellement optimisées, du programme d'entrée du compilateur. D'autre part, les chercheurs en compilation optimisante ont besoin d'outils facilitant le développement rapide d'un compilateur en constante évolution pour prototyper et évaluer de nouvelles idées. Pour ces raisons, il est intéressant et utile de s'intéresser à l'intégration de l'IDM au sein de la conception d'un compilateur optimisant dans un environnement de recherche.

Dans ce chapitre, nous illustrons le rôle que peuvent jouer les techniques IDM dans la conception d'un compilateur de recherche. GeCoS [1] est un compilateur principalement orienté source à source qui offre un haut niveau d'extensibilité. Il est notamment utilisé dans l'expérimentation des différentes approches ASIP présentées dans les chapitres précédents. Nous identifions les principales tâches réalisées au cours du développement d'un tel compilateur et indiquons comment l'IDM aide à réduire significativement leurs difficultés de mise en œuvre. Les technologies de l'IDM ne concernent pas le métier des logiciels et, dans le cas particulier d'un compilateur, elles ne masquent pas la complexité élevée des algorithmes d'optimisation. Des attentes irréalistes sur ce que l'IDM peut apporter risquent d'amener à une utilisation inefficace et frustrante de ces techniques. Afin d'éviter cette situation, nous discutons de l'applicabilité de l'IDM dans le domaine spécifique de la compilation optimisante.

La suite de ce chapitre est organisée de la manière suivante. La première section identifie les

1. http://gecos.gforge.inria.fr

principaux défis issus de la conception d'un compilateur optimisant. La deuxième section se base sur l'expérience acquise dans l'intégration de l'IDM au sein de GeCoS pour détailler les différents éléments de réponse aux défis précédents. Enfin, la dernière section cadre l'applicabilité de l'IDM dans les compilateurs en clarifiant ses objectifs et prérequis.

7.2 Les défis d'un compilateur optimisant

Les recherches sur la compilation optimisante visent à obtenir de hautes performances par l'analyse et l'optimisation de programmes au niveau du compilateur. Elles répondent à des problèmes très spécifiques (parallélisation automatique, sélection d'instructions, etc.) qui constituent les briques d'un compilateur complet. Par conséquent, les phases de prototypage des analyses et des transformations sont au cœur du développement d'une infrastructure de compilation. Il est important d'avoir une infrastructure permettant de valider rapidement les idées et prototypes d'optimisation. Ce haut niveau de réactivité implique de fréquentes évolutions de l'infrastructure qui restent acceptables dans un contexte de recherche où un compilateur n'est pas attendu comme étant aussi stable et robuste qu'un compilateur de production. De plus, les performances du compilateur ne sont pas réellement critiques à partir du moment où la sortie produite amène à un gain de performance sur l'architecture cible. Les caractéristiques spécifiques à un compilateur axé sur la recherche induisent de nombreux défis pour les concepteurs, ceux-ci sont détaillés dans les prochains paragraphes.

7.2.1 Maintenabilité et pérennité du code

Un des défis fondamentaux dans notre contexte est la pérennité et la maintenabilité du code. Les compilateurs de recherche sont des infrastructures logicielles très complexes développées par plusieurs générations d'étudiants et d'experts de domaines spécifiques. Ce taux élevé de renouvellement requiert la mise en œuvre d'un processus de développement incrémental reposant notamment sur une homogénéisation du style de programmation. Cette tâche est d'autant plus difficile que les contributeurs n'ont, d'une part, pas nécessairement une expérience significative en ingénierie logicielle et sont, d'autre part, parfois issus de domaines aux dogmes radicalement différents. Par exemple, un expert matériel avec une formation avancée en électronique n'associe pas généralement les mêmes préoccupations à un outil qu'un concepteur logiciel soucieux des possibilités d'évolution de cet outil.

De manière générale, dans un logiciel pour la recherche, les intervenants ont un double rôle : ils sont à la fois les clients et les concepteurs du logiciel. Cette ambiguïté n'est pas spécifique aux compilateurs, mais son impact est d'autant plus important dans une infrastructure de compilation où les domaines impliqués constituent un éventail très large de compétences, allant de l'interprétation abstraite aux descriptions matérielles de circuits. Un acteur du développement de l'infrastructure est généralement expert dans l'un de ces domaines métiers et n'a pas nécessairement les compétences ni le temps pour appréhender la conception de l'outil dans son ensemble. La multiplicité de ces besoins métiers associée aux exigences d'une infrastructure flexible et pérenne constitue un problème de conception logiciel complexe dont la solution est un compromis difficile à atteindre.

7.2.2 Validation structurelle des représentations intermédiaires

Lors de l'écriture d'une transformation optimisante, une des tâches les plus répétitives consiste à s'assurer que les contraintes sur la structure de données de la représentation intermédiaire soient respectées après transformation. Il existe de nombreuses règles structurelles à respecter. Par exemple, dans la plupart des programmes impératifs, il faut assurer qu'une instruction qui utilise une variable se trouve après la déclaration de cette variable. De telles validations sont généralement effectuées après l'analyse lexicale et syntaxique du programme. Dans un compilateur de recherche, il peut être intéressant d'effectuer ces vérifications après l'exécution d'une transformation afin de détecter le plus tôt possible une incohérence évidente dans le flot d'optimisation. L'écriture de ces vérifications peut s'avérer coûteuse en termes de temps, car elles requièrent de nombreuses opérations de navigation dans la représentation intermédiaire et s'appuient souvent sur la gestion d'un historique des états courants du parcours. Ces opérations sont fastidieuses à décrire et fortement sujettes aux erreurs.

D'autre part, les compilateurs optimisants utilisent parfois des langages expérimentaux qui sont plus souvent étendus et modifiés que les langages généralistes. Chacune de ces modifications ou extensions risque d'amener les concepteurs à réécrire une partie ou l'intégralité de chacune des analyses. Ce coût de maintenance supplémentaire peut rapidement devenir inacceptable et conduire à des vérifications incohérentes. Garantir un certain niveau de robustesse dans un compilateur travaillant sur des langages spécifiques expérimentaux pose donc un réel défi du fait de sa spécification mouvante, qui rend très difficile le maintien de la cohérence des vérifications structurelles de la représentation intermédiaire.

7.2.3 Requêtes complexes sur les représentations intermédiaires

Dans un compilateur, une passe d'optimisation n'est généralement applicable qu'à un sous-ensemble de constructions du langage pour lequel un ensemble de préconditions sont respectées. Retrouver ces constructions et vérifier que les préconditions sont respectées requièrent de nombreuses analyses au sein de la représentation intermédiaire. La plupart des analyses ou requêtes peuvent être exprimées par des opérations de recherche de motifs plus ou moins complexes. Par exemple, avant de dérouler une boucle, il faut s'assurer que les bornes et le pas de cette boucle sont constants. Une passe de déroulage de boucles identifie donc toutes les boucles respectant ces conditions et vérifie que le corps de chaque boucle ne produit pas d'effets de bord sur son indice.

De même que pour les opérations de validation structurelle, les requêtes complexes s'appuient sur des parcours de la représentation intermédiaire. Elles sont donc sujettes aux mêmes difficultés quant au coût du maintien de la cohérence avec les évolutions de la représentation intermédiaire. Ces opérations de navigation sont généralement mises en œuvre efficacement par un patron de conception de type *visiteur*. Cependant, une requête complexe induit un code difficile à comprendre et à maintenir. Il est important de noter que, bien souvent, cette difficulté n'est en fait qu'un *bruit* technique provenant du fossé artificiel séparant l'intention de la requête de sa mise en œuvre dans l'implémentation concrète de la représentation intermédiaire.

7.2.4 Interaction avec des outils externes

Les compilateurs expérimentaux s'appuient sur de multiples logiciels externes pour répondre à des besoins très spécifiques. Par exemple, les solveurs de satisfaction booléenne et de programmation linéaire sont souvent utilisés pour exprimer et résoudre des problèmes complexes d'optimisation. Ces outils peuvent être codés dans de multiples langages et requièrent donc la définition d'interfaces si le compilateur est écrit dans un langage différent. Ces interfaces peuvent se décliner en différents niveaux de couplage avec le compilateur. Si l'utilisation du format d'entrée standard d'un outil est le moyen le plus simple de procéder, il est souvent plus intéressant de s'interfacer directement avec la représentation intermédiaire de cet outil, si cela est possible. La construction et le maintien de ces interfaces constituent une charge de travail qui peut s'avérer suffisamment lourde pour dissuader les concepteurs de suivre l'évolution ou même d'utiliser un outil externe. Cet aspect pose un réel problème dans le cadre d'une infrastructure expérimentale où l'aspect exploratoire des différentes solutions existantes joue un rôle important.

7.2.5 Transformations préservant la sémantique

La caractéristique fondamentale d'un compilateur est d'assurer que la sortie conserve la sémantique originale du programme. Cette caractéristique fondamentale est paradoxalement la plus difficile à assurer. Cette difficulté conduit à un intérêt croissant pour la notion de prouvabilité, comme dans le cas du compilateur prouvé CompCert [116]. Dans le contexte d'un compilateur optimisant, un défi de taille est de prouver qu'une transformation conserve la sémantique tout en garantissant que sa mise en œuvre préserve l'intention originale du concepteur. Idéalement, chaque transformation est associée à une preuve de validité et à des outils garantissant que la mise en œuvre de la transformation préserve cette preuve. Ce fonctionnement est très utile pour identifier rapidement des incohérences entre la théorie et la pratique qui, trop souvent, ne tient pas compte de tous les cas particuliers.

7.2.6 Capturer les connaissances spécifiques aux domaines

Une des raisons du manque d'efficacité des compilateurs est qu'il est très difficile d'identifier toutes les opportunités d'optimisation pour une architecture cible. Le fossé entre un langage généraliste et les capacités réelles d'une architecture (parallélisme, mémorisation, etc.) est souvent trop important pour que des techniques automatiques réussissent à en tirer pleinement parti.

Pour combler les lacunes des langages généralistes, une approche possible consiste à définir des dialectes et/ou des extensions de langages. L'objectif est de réduire le fossé entre l'algorithme et l'architecture en exprimant explicitement des informations liées à la cible matérielle. La programmation parallèle s'appuie sur cette connaissance spécifique explicite pour atteindre un niveau de performance en adéquation avec l'architecture [204, 36, 34]. D'autres approches sont partisanes de langages alternatifs et plus spécifiques pour mieux répondre aux objectifs qui leurs sont propres. Les environnements MMAlpha [129] et plus récemment AlphaZ [2] proposent, par exemple, un langage où les calculs sont exprimés sous forme d'équations mathématiques, pour favoriser la séparation des préoccupations : ce qui est calculé devrait être indépendant d'autres choix tels que l'allocation mémoire.

2. https ://www.cs.colostate.edu/AlphaZ/

La connaissance spécifique d'un domaine est également utilisée par les concepteurs de compilateurs eux-mêmes. Par exemple, la plupart des compilateurs repose sur une description formelle du jeu d'instructions du processeur et de sa microarchitecture. Cette description est utilisée pour automatiser l'adaptation d'un compilateur à une nouvelle cible matérielle.

Cependant, recourir à des langages spécifiques aux domaines (DSL [3]) implique de concevoir des outils dédiés dont les coûts de développement et de maintenance sont élevés. A titre d'exemple, bien que des outils existent pour faciliter les tâches d'analyse syntaxique et lexicale (e.g., ANTLR, Yacc, etc.), ceux-ci ne permettent pas de s'interfacer directement avec la représentation intermédiaire du compilateur. Cet aspect constitue un réel problème dans un contexte de recherche où les DSL sont généralement conçus de manière incrémentale, et où chaque nouvelle fonctionnalité du DSL est alors associée à un effort de développement significatif. Évidemment, ces problèmes sont encore plus importants quand il s'agit d'étendre des langages généralistes avec des DSL embarqués. Les langages généralistes sont en effet généralement très complexes (e.g., C/C++) et, à ce jour, aucun compilateur de compilateur ne permet de gérer les problèmes issus d'une telle intégration. La solution pragmatique actuelle consiste à décorer le code source de l'application par des annotations exprimant explicitement des directives guidant le compilateur.

7.2.7 Génération de code

Un compilateur peut cibler de multiples architectures ou de multiples langages. La dernière étape du compilateur consiste à traduire la représentation intermédiaire en une forme compréhensible par la cible.

Si la cible est également un langage de haut niveau, on parle de compilateur source à source (e.g., [94, 15, 155, 115, 135]). L'objectif d'un compilateur source à source est de produire un code source optimisé qui exprime implicitement ou explicitement une sémantique guidant le compilateur cible vers un programme compilé plus efficace. Ceci est particulièrement intéressant dans le cas d'un compilateur optimisant de recherche dont l'aspect exploratoire évalue notamment les possibilités de multiples architectures parallèles émergentes (e.g., IBM/SONY/Toshiba Cell BE, GPGPU, Intel Larrabee). La complexité de ces architectures associée à l'expertise déployée par les concepteurs dans leurs compilateurs natifs, amène à considérer l'approche source à source comme la plus adaptée à des expérimentations rapides et robustes. Ce constat peut être développé, particulièrement pour des cibles hétérogènes, jusqu'à envisager le langage C comme une représentation intermédiaire de haut niveau permettant de s'interfacer avec la majorité des outils intervenant dans un flot de compilation optimisant [79].

Dans le cadre de notre flot de conception et de compilation ASIP, la sortie de l'outil est composée d'un programme exécutable sur l'architecture et d'une description matérielle de l'extension. L'utilisation d'un code source de haut niveau associé à des blocs d'instructions de type assembleur permet de décrire finement le comportement du processeur pour les zones exploitant l'extension matérielle. Le générateur de code du compilateur source à source se doit donc d'être extensible pour permettre de traiter les comportements spécifiques des opérations déportées sur l'extension. D'autre part, la définition de l'extension s'appuie sur une phase de génération de la description matérielle via des langages de type VHDL, Verilog ou SystemC. Cette étape se situe en périphérie du flot classique

3. Domain Specific Language

de compilation et se rapproche des problématiques de la synthèse de haut niveau, dont l'objectif est d'obtenir une description matérielle à partir d'un programme écrit dans un langage généraliste.

Développer des générateurs de code pour chaque cible matérielle ou langage requiert un effort important qui peut être réduit significativement par des outils permettant de réutiliser et de personnaliser ces générateurs.

7.3 Utilisation de l'IDM dans les compilateurs

Les représentations intermédiaires d'un compilateur sont des abstractions de programmes et donc, par essence, des modèles. Une instance d'une représentation intermédiaire correspond à l'abstraction d'un code source donné. Dans ce contexte, la grammaire du langage source ou, plus couramment, la structure de la représentation intermédiaire devient le métamodèle. Dans cette section, nous présentons les réponses partielles de l'IDM aux défis d'un compilateur optimisant axé sur la recherche. Cette description est issue de notre expérience autour de l'utilisation des techniques de l'IDM au sein du compilateur GeCoS, utilisé notamment dans notre flot de compilation ASIP.

7.3.1 Les bénéfices directs de l'IDM

L'infrastructure de compilation GeCoS est issue de travaux [121] proposant un environnement pour la conception et la compilation pour des *soft-core* sur FPGA. L'objectif de cette infrastructure était alors d'utiliser Eclipse pour profiter d'un environnement hautement modulaire, facilitant l'évaluation rapide d'approches expérimentales. Cette flexibilité a rapidement fait naître le besoin de formaliser et de standardiser le processus de développement de l'infrastructure. L'IDM constitue une réponse directe à ce besoin. GeCoS étant fortement couplé à Eclipse, nous nous sommes naturellement intéressés aux outils de métamodélisation fournis par EMF [4].

Le modèle constitue une documentation Un bénéfice immédiat de l'IDM est que l'intégralité des informations clés réside dans la spécification du métamodèle. Celui-ci ne concerne qu'un domaine spécifique d'un problème, il est donc exempt des informations parasites d'une mise en œuvre concrète. Cette abstraction, qui est un prérequis, permet à de nouveaux acteurs du développement de s'approprier rapidement les différentes structures de données, y compris dans les cas, fréquents, où la documentation est manquante. D'autre part, le travail de modélisation effectué lors de la création du métamodèle évite la mise en œuvre « frénétique » et donc source d'erreurs d'une approche expérimentale.

Générateur de code : du modèle à sa réalisation L'homogénéisation du code et l'utilisation de bonnes pratiques logicielles sont une conséquence directe de l'IDM. Les générateurs de code (e.g., générateur de code Java EMF) produisent une implémentation respectant des interfaces standard et assurant la consistance structurelle des modèles. Le code généré offre notamment une réflexivité avancée (e.g., informations sur la composition et les attributs des objets) qui facilite l'écriture de fonctions utilitaires sans nécessiter une instrumentation du métamodèle, fastidieuse et éloignée des préoccupations du problème modélisé.

4. http://www.eclipse.org/modeling/emf/

Accès à des outils génériques Les outils génériques, applicables à n'importe quel métamodèle, offrent une valeur ajoutée significative pour un très faible coût de mise en œuvre. Ces outils participent à améliorer la robustesse des compilateurs. Tout d'abord, un modèle valide doit respecter les propriétés structurelles de son métamodèle. Ces propriétés peuvent être facilement vérifiées, par exemple lors de la sérialisation d'un modèle. Cette sérialisation, fonctionnalité de base fournie par EMF, analyse les arités des références et s'assure de leur cohérence avec le métamodèle. De plus, un modèle sérialisé est visualisable dans un éditeur arborescent généré automatiquement par EMF. L'arborescence provient de l'analyse des informations de composition décrites dans le métamodèle. La figure 7.1 illustre l'éditeur légèrement personnalisé de la représentation intermédiaire standard de GeCoS . Celui a été généré directement à partir de son métamodèle. La racine de l'éditeur correspond à l'ensemble des procédures déclarées dans le fichier C. L'arborescence de cette racine fait notamment apparaître la hiérarchie des blocs du programme et les instructions qui sont contenues dans les blocs de base. Cette facilité d'observation permet de détecter rapidement des erreurs apparaissant dans le résultat d'une transformation.

```
1  void MM(float **a, float **b, float
       **c, int M){
2    int i,j,k;
3    for (i=0;i<M;i=i+1){
4      for (j=0;j<M;j=j+1){
5        c[i][j]=0.0;
6        for (k=0;k<M;k=k+1){
7          c[i][j]+=a[i][k]*b[k][j];
8        }
9      }
10   }
11 }
```

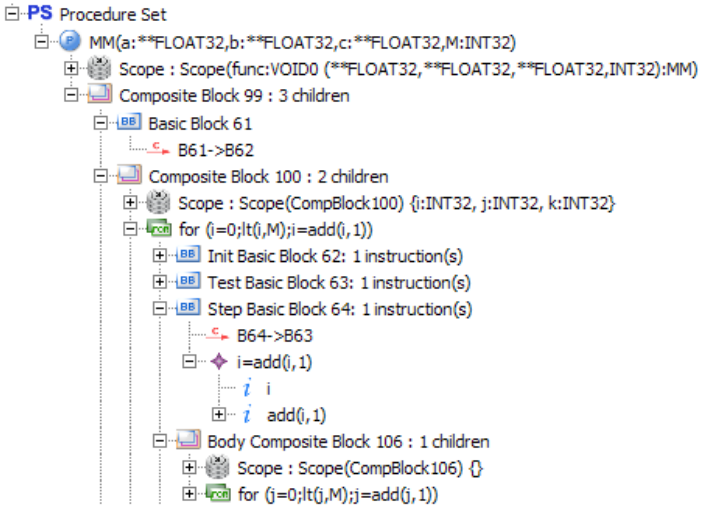


FIGURE 7.1 – Visualisation de la représentation intermédiaire arborescente de GeCoS pour une multiplication de matrices.

Le langage de contraintes OCL [5] peut être utilisé pour exprimer des règles additionnelles de validation des modèles. Il offre la possibilité d'exprimer des requêtes complexes sur la structure d'un modèle à partir des informations issues du métamodèle. Le langage OCL dispose d'un ensemble d'opérations sur les ensembles qui facilite particulièrement la mise en œuvre des requêtes sur des représentations intermédiaires arborescentes. Cela s'avère utile dans un flot de compilation optimisant constitué d'une séquence complexe de transformations pour assurer, à chaque étape, que les pré et post conditions d'une transformation sont respectées. Par exemple, une passe d'évaluation des constantes d'un programme est plus efficace sur une représentation intermédiaire sous une forme à assignation unique (SSA [6]). Une requête OCL simple (cf. figure 7.2) garantissant que chaque variable n'est affectée qu'une seule fois, correspond alors à une pré-condition de l'optimisation.

Généraliser ces vérifications additionnelles permet d'accroître la robustesse globale du compilateur

5. Object Constraint Language
6. Static Single Assignment

```
let affectations : Collection(SymbolInstruction) =
  self.basicBlocks.instructions
    ->select(i | i.oclIsTypeOf(SetInstruction))
    ->collect(set | set.oclAsType(SetInstruction).dest)
in
self.scope.allSymbols
  ->forAll(s | affectations->select(e | e.symbol=s)->size()<=1)
```

FIGURE 7.2 – Vérification OCL de la forme SSA d'une procédure GeCoS.

tout en facilitant l'appréhension d'un flot de compilation complexe en identifiant clairement les interactions des différentes transformations. L'intégration d'un interpréteur OCL dans les éditeurs générés automatiquement par EMF est une fonctionnalité standard qui permet de prototyper rapidement des contraintes OCL sur des modèles concrets de représentations intermédiaires.

7.3.2 Utilisation des métaoutils existants

L'IDM améliore significativement l'efficacité des développements par des outils génériques dédiés à des tâches spécifiques et complexes. Ces outils ont en commun qu'ils utilisent tous la même représentation intermédiaire correspondant à la description structurelle des métamodèles. On parle alors de métaoutils.

Outils pour la définition de DSL Comme évoqué dans la section 7.2, les compilateurs peuvent tirer parti des connaissances spécifiques à un domaine pour guider les optimisations. Des outils tels que Xtext [7] ou EMFText [8] permettent de décrire la syntaxe textuelle d'un DSL et de générer automatiquement l'environnement associé. Cet environnement dispose d'une coloration syntaxique, d'une autocomplétion contextuelle ainsi que d'une infrastructure d'analyse statique des erreurs et d'assistance à la correction dont le comportement est relativement simple à personnaliser. Dans le cas de Xtext, le métamodèle de la structure de données est inféré à partir de la description de la syntaxe du DSL. Il est donc plus simple de faire évoluer un langage, puisque les répercutions sur la structure de données sont immédiates.

Outils pour la transformation de modèles Les multiples représentations intermédiaires d'un compilateur optimisant amènent les concepteurs à définir des transformations pour chaque représentation et les ponts logiciels permettant de passer d'une représentation à l'autre. Dans les deux cas, il s'agit de transformer un modèle en un autre modèle. Si les deux modèles partagent le même métamodèle, il s'agit d'une transformation de la même représentation intermédiaire. Si les métamodèles source et destination sont différents, il s'agit d'un pont entre deux représentations intermédiaires.

Des outils de modèle à modèle (M2M) existent pour alléger le coût de mise en oeuvre de telles opérations. Certains de ces outils (e.g., ATL [9] et ETL [10]) proposent de définir de simples règles de correspondance entre le modèle source et le modèle destination. L'outil est ensuite chargé d'identifier et d'appliquer automatiquement les occurrences de toutes ces règles dans le modèle source. Si ce

7. http://www.eclipse.org/Xtext/
8. http://www.emftext.org/
9. http://www.eclipse.org/atl/
10. http://www.eclipse.org/gmt/epsilon/doc/etl/

```
1 // Code generation dispatch for a symbol instruction
2 def dispatch generate(SymbolInstruction s){
3    '''« s.symbol.name »'''
4 }
5
6 // Code generation dispatch for a set instruction
7 def dispatch generate(SetInstruction s) {
8    '''« generate(s.dest) »=« generate(s.source) » ;'''
9 }
```

FIGURE 7.3 – Extrait de générateur Xtend2 pour le code C des instructions GeCoS .

fonctionnement est très intéressant pour des transformations simples, les règles s'avèrent rapidement difficiles à exprimer lorsque les différences entre le modèle source et destination sont trop importantes. Dans de telles situations, il est préférable d'utiliser un langage plus généraliste pour exprimer le comportement de ces transformations.

Kermeta [11] est un environnement de métaprogrammation construit autour d'un langage capable d'exprimer à la fois la structure et le comportement d'un métamodèle. Il constitue une alternative intéressante, car la nature de son langage permet de définir des transformations complexes tout en bénéficiant de fonctionnalités logicielles avancées (e.g., programmation par aspect) et en conservant une indépendance vis-à-vis de l'implémentation concrète du métamodèle.

Outils pour la génération de code Les outils axés sur la génération d'une sortie textuelle à partir d'un modèle (M2T [12]) tel que Xpand/Xtend [13] et, plus récemment, Xtend2 [14] offrent une spécification flexible et modulaire du code généré à travers la gestion d'imports et d'aspects. De plus, les règles de génération de chaque entité du modèle supportent le *polymorphic dispatch*. Il s'agit d'une extension du patron de conception *visiteur* permettant à un objet de visiter la fonction adaptée à son type. Dans le cas du *polymorphic dispatch*, et à l'inverse du *visiteur*, aucun artefact intrusif n'est nécessaire dans le code du modèle pour obtenir ce comportement. Ce sont les méthodes visitées elles-mêmes qui définissent le type d'objet qu'elles supportent. Ceci est particulièrement utile dans un compilateur où une représentation intermédiaire est souvent décrite par un arbre de syntaxe abstraite dont les nœuds sont des spécialisations d'une unique définition abstraite.

La figure 7.3 illustre un extrait d'un générateur de code C écrit avec Xtend2. Les méthodes dites de *dispatch* spécialisent le comportement du générateur pour chaque type d'instruction. L'instruction *SetInstruction* est une affectation définie par une source et une destination. Le mot clé *dispatch* assure que la fonction *generate* (ligne 7) est appelée lors de la génération d'une *SetInstruction*, les codes associés à la source et à la destination seront séparés par l'opérateur C d'une affectation. Une *SymbolInstruction* représente un accès à une variable scalaire, la ligne 3 indique que le rendu de cette instruction correspond au nom de la variable référencée. D'autre part, les triples *quote* de chaque fonction identifient des zones de chaînes de caractères particulières, favorisant la lisibilité du code généré à travers la gestion des tabulations et des retours à la lignes.

11. http://www.kermeta.org/
12. Model-to-Text
13. http://www.eclipse.org/modeling/m2t/?project=xpand
14. http://www.eclipse.org/Xtext/xtend/

Les outils M2T associent donc des fonctionnalités logicielles avancées à des facilités dédiées au domaine de la génération de code (e.g., concaténation de chaînes de caractères, gestion des tabulations, etc.). Ils simplifient la conception de générateurs de code aisément adaptables à de multiples variations de la représentation intermédiaire ou de la cible textuelle. Ces outils sont donc particulièrement intéressants dans le contexte d'un compilateur optimisant s'il est amené à cibler de multiples dialectes ou plateformes matérielles.

7.3.3 Définition de nouveaux métaoutils

Tous les métamodèles sont décrits par le même modèle (appelé *métamétamodèle*). Les métaoutils analysent et manipulent les métamodèles en travaillant au niveau de ce métamétamodèle. La diversité des représentations intermédiaires au sein d'un compilateur a pour conséquence d'alourdir la conception globale par la récurrence de certaines tâches parfois complexes. La capitalisation des algorithmes de ces tâches est facilitée par la définition de nouveaux métaoutils génériques issus de besoins spécifiques. Ces outils peuvent être décrits de manière générative ou encore par interprétation pour un prototypage rapide. Dans les deux cas, un environnement dédié offre un langage qui se base sur le métamétamodèle pour exprimer les comportements des tâches à capitaliser.

Approches génératives Les métaoutils génératifs automatisent la réutilisation de transformations en générant leurs comportements instanciés pour différents métamodèles cibles. Certaines tâches peuvent être entièrement automatisées via des transformations M2M (e.g., ATL ou Kermeta), d'autres peuvent être guidées par des DSL.

Les patrons structurels de conception logicielle constituent des exemples parfaits de concepts génériques. Les exprimer au niveau métamétamodèle offre une puissante boîte à outils aux développeurs, qui peuvent alors appliquer ces patrons à tous leurs métamodèles. Dans un compilateur, l'analyse et la transformation d'une représentation intermédiaire s'appuient souvent sur le patron de conception *visiteur* associé à un algorithme de parcours (e.g., en profondeur ou en largeur). Ce patron est intrusif puisqu'il nécessite d'ajouter une méthode dans chaque type d'objet visité. Sa fastidieuse mise en œuvre pour chaque représentation peut être automatisée par une simple transformation M2M des métamodèles. Les algorithmes de parcours classiques peuvent être inférés en analysant les informations structurelles du métamodèle et choisir, par exemple, de visiter les objets contenus par chaque objet visité.

Certaines tâches répétitives sont plus complexes et leur capitalisation nécessite parfois une connaissance explicite d'une partie de leur comportement instancié. Définir des DSL simples pour exprimer ces informations permet d'automatiser des tâches complexes comme celle, notamment, de s'interfacer avec des outils externes. Nous présentons dans la suite de ce paragraphe, deux métaoutils mis en œuvre au cours de cette thèse et qui nous ont permis d'accroître significativement notre productivité pour des taches répétitives et coûteuses.

Exemple 1 : *Graph Mapper* Les optimisations d'un compilateur reposent souvent sur une représentation intermédiaire sous forme de graphe. Pour éviter de coder les algorithmes, souvent complexes, de la théorie des graphes, il est préférable d'utiliser une bibliothèque externe.

Deux possibilités s'offrent alors au concepteur. La première consiste à considérer la représentation

```
1  import "platform:/resource/fr.irisa.cairn.gecos.model/model/gecos.cdfg.ecore";
2  GraphAdapter adapt DAGInstruction {
3    directed <= true;
4    initialize {
5      adapted.nodes.foreach {n | nodeBuilder (n)};
6      adapted.edges.foreach {e | edgeBuilder (e)};
7    }
8  }
9
10 NodeAdapter adapt DAGNode {
11   initialize {
12     adapted.inputs.foreach {p | inportBuilder (p)};
13     adapted.outputs.foreach {p | outportBuilder (p)};
14   }
15 }
16
17 PortAdapter adapt DAGPort {}
18
19 EdgeAdapter adapt DAGEdge {
20   source <= adapted.src;
21   destination <= adapted.dest;
22 }
```

FIGURE 7.4 – Typage explicite de la forme DAG d'une instruction GeCoS à travers un DSL. Cette description permet de générer un adaptateur entre le métamodèle du DAG et une structure externe de graphe.

intermédiaire comme une spécialisation de la structure du graphe de la bibliothèque externe. Elle expose un couplage important entre la représentation intermédiaire et la bibliothèque. Ce couplage peut être problématique pour de multiples raisons (e.g., évolutions de la représentation intermédiaire ou de la bibliothèque externe, licences incompatibles, etc.). Dans ce contexte, il est préférable de construire une interface entre la représentation intermédiaire et la structure externe du graphe. Cependant, la description d'une nouvelle interface pour chaque représentation intermédiaire s'avère rapidement coûteuse et répétitive. Automatiser ce processus est difficile, puisque les concepts intrinsèques à un graphe ne sont pas nécessairement explicites dans un métamodèle. Utiliser un DSL permet de décrire simplement la manière dont sont exprimés les concepts de graphe, de nœuds et de liens dans n'importe quel métamodèle.

Le code de la figure 7.4 illustre comment les instructions d'un bloc de base d'un CDFG, mises sous forme de graphe acyclique (cf. métamodèle DAG, figure 7.5), sont explicitement associées à un type abstrait de graphe. À chaque concept, on associe une entité (cf. lignes 2, 10, 17 et 19) du métamodèle importé et la manière d'initialiser l'adaptateur correspondant. Par exemple, à la ligne 5 on indique que pour chaque élément référencé par l'attribut *nodes* d'une *DAGInstruction*, on appelle une fonction (*nodeBuilder*) chargée de construire l'adaptateur d'un nœud. L'ordre de construction des adaptateurs est donc déterminé par le comportement de chacune de ces zones d'initialisation (e.g., lignes 4-7). Le choix des éléments du métamodèle référencés par les expressions d'initialisation est simplifié par l'auto-complétion de l'éditeur du DSL qui fournit uniquement la liste des références légales dans le contexte courant. Par exemple, à la ligne 5, les seules références valides sont celles portant sur les attributs *nodes* et *edges* de la classe *DAGInstruction* qui a été sélectionnée comme

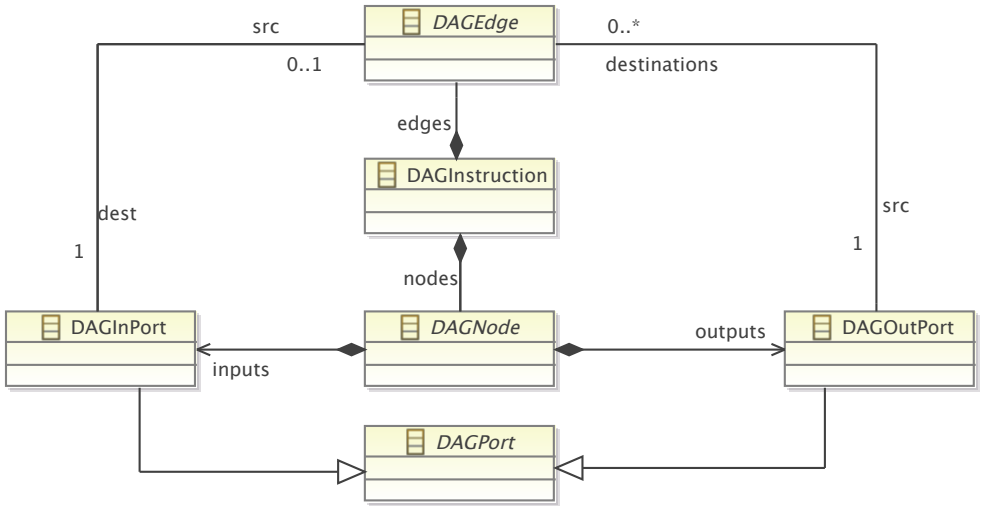


FIGURE 7.5 – Extrait du diagramme de classe du métamodèle correspondant aux instructions DAG de GeCoS .

étant la cible de l'adaptateur de graphe.

Dans cet exemple simple, le métamodèle est très proche de la structure du graphe externe et chaque référence à une entité du métamodèle correspond à une référence dans le graphe cible. Par exemple, le port source d'un lien (cf. ligne 20) correspond à l'adaptateur du port référencé par *src* dans la classe *DAGEdge*. Dans des cas plus complexes (e.g., un métamodèle de graphe défini uniquement par une liste de nœuds connaissant leurs successeurs), il peut être nécessaire de fournir des paramètres supplémentaires aux fonctions de construction des adaptateurs. De plus, les expressions initialisant les attributs de chaque adaptateur peuvent faire appel à des fonctions utilitaires dont les comportements sont décrits en Java et fournis sous forme de chaînes de caractères dans le DSL.

À partir de cette description, GraphMapper génère le code Java permettant de construire automatiquement un adaptateur entre la représentation intermédiaire et l'implémentation externe du graphe. Ce métaoutil a été conçu en collaboration avec Jérémie Guidoux, pour, notamment, connecter notre flot d'optimisation à une représentation intermédiaire fournie par la société STMicroelectronics dans le cadre du projet *RecMotifs* (NANO2012).

Exemple 2 : *TOM Mapper* Tom/Gom [15] est un outil de manipulation de structures arborescentes. Il offre un langage dédié à l'élaboration de règles complexes sur ces structures. Les règles peuvent être utilisées comme des motifs à identifier ou encore comme une transformation à appliquer. GeCoS utilise Tom pour exprimer les transformations qui ne sont que des réécritures de la représentation intermédiaire (e.g., opérations arithmétiques, etc.) et pour détecter des motifs difficiles à exprimer en OCL ou dans un langage généraliste. Par exemple, les expressions d'un programme tel que $2i + 2i$ peuvent être simplifiées en $4i$ en utilisant la règle de réécriture suivante :

$$\texttt{add}(\texttt{term}(\texttt{c1}, \texttt{var}), \texttt{term}(\texttt{c2}, \texttt{var})) \rightarrow \texttt{term}(\texttt{c1} + \texttt{c2}, \texttt{var})$$

15. http://tom.loria.fr/

```
1 %op Inst mul (children : InstL){
2   //Test d'applicabilit de l'oprateur
3   is_fsym(t) {$t instanceof GenericInstruction && ((GenericInstruction)$t).getName().equals("
        mul")}
4   //Accesseur pour les instructions filles
5   get_slot(children,t) {((GenericInstruction)$t).getChildren()}
6   //Construction de l'oprateur
7   make(_children) {GecosTomFactory.createMul($_children)}
8 }
```

FIGURE 7.6 – Description Tom de l'opérateur *mul* (cf. figure 7.7) correspondant à une instruction de multiplication.

elle simplifie l'addition de deux termes linéaires si leurs variables respectives sont identiques. Ces règles sont plus faciles à exprimer et à appréhender qu'une transformation basée sur un *visiteur* par exemple. Cependant, pour utiliser Tom/Gom, il est nécessaire de déclarer les terminaux (i.e. les feuilles de l'arbre manipulé) et la manière de détecter et de construire les différentes expressions supportées dans les règles. S'il est possible d'inférer ces éléments en analysant l'information structurelle d'un métamodèle, le résultat est souvent, dans le cas de métamodèles relativement complexes, difficile à exploiter directement. Pour apporter une réponse à ce problème, nous avons conçu le métaoutil TomMapper, qui guide l'inférence en indiquant explicitement les terminaux traités et éventuellement des règles personnalisées (nom de la règle, ordre des paramètres, etc.) dans un DSL.

La figure 7.7 illustre un extrait de la description dans TomMapper des terminaux et opérateurs des instructions de GeCoS. Le terminal *Inst* (ligne 7) référence la classe abstraite de toutes les instructions décrites dans le métamodèle importé (ligne 3) et *InstL* correspond à une liste de ces terminaux. Les opérateurs personnalisés sont définis dans des modules, ils offrent une syntaxe concise pour exprimer des motifs arborescents dans une instruction. Par exemple, l'opérateur *add* (ligne 14) représente une addition. Les paramètres de l'opérateur sont issus des attributs de la classe *GenericInstruction* qui modélise une instruction disposant d'un nom et contenant une liste d'instructions filles. Dans le cas d'une addition, le nom de l'instruction doit être égal à *add* et les instructions filles sont des paramètres du constructeur de l'opérateur Tom (cf. ligne 7 de la figure 7.6). Afin de simplifier l'expression des motifs Tom, le type d'une instruction est ignoré dans l'appel d'un opérateur. La perte de cette information ne pose pas de problème puisque les types sont inférés lors de la construction des instructions.

La description des différents opérateurs est ensuite analysée pour identifier tous les autres opérateurs générés par défaut (un opérateur par classe qui hérite d'un terminal et qui n'a pas été personnalisée) et générer les interfaces avec la syntaxe Tom/Gom. La figure 7.6 illustre la syntaxe d'un fichier Tom généré par TomMapper pour l'opérateur *add* (ligne 14). Elle correspond aux trois primitives nécessaires à la recherche et à la création de cet opérateur : identification, accesseurs et constructeur.

Les deux exemples précédents (GraphMapper et TomMapper) sont des cas particuliers d'un problème abordé dans les recherches sur l'IDM sous le nom de *Model Mapping*. Ce thème s'intéresse aux adaptateurs entre différentes structures de données existantes avec une contrainte forte d'indépendance. Il existe des langages dédiés pour exprimer les concepts communs à tous les mécanismes d'adaptateurs. Par exemple, Clavreul et al. [42] proposent un processus de génération d'adaptateur

```
1  TomMapping gecos;
2  prefix "fr.irisa.cairn.gecos.model.tom";
3  import "platform:/resource/fr.irisa.cairn.gecos.model/model/gecos.cdfg.ecore";
4
5  terminals {
6    define {
7      Inst : gecos.instrs.Instruction,  //Nom du terminal pour les instructions
8      InstL : gecos.instrs.Instruction[], //Nom du terminal pour les listes d'instructions
9    }
10 }
11 //Declaration des operateurs specifiques aux instructions arithmetiques
12 module arithmetic {
13   operators {
14     op add::gecos.instrs.GenericInstruction(name="add", children, ignore type);
15     op sub::gecos.instrs.GenericInstruction(name="sub", children, ignore type);
16     op mul::gecos.instrs.GenericInstruction(name="mul", children, ignore type);
17     op div::gecos.instrs.GenericInstruction(name="div", children, ignore type);
18     op shr::gecos.instrs.GenericInstruction(name="shr", children, ignore type);
19     op shl::gecos.instrs.GenericInstruction(name="shl", children, ignore type);
20   }
21 }
```

FIGURE 7.7 – Extrait de la description des opérateurs TOM dans TomMapper pour les instructions arborescentes de GeCoS .

basé sur différents niveaux d'automatisation des règles de correspondances bidirectionnelles entre deux métamodèles. Ils définissent quatre types de stratégie allant d'une simple copie de tous les attributs, dans le cas d'une correspondance parfaite, à une stratégie complètement personnalisée exprimée en Kermeta. Le choix de ces stratégies est laissé au concepteur de l'adaptateur à travers un DSL. Ce type d'approche pourrait être utilisé dans la conception d'un compilateur pour faciliter le recours aux outils externes et le passage d'une représentation intermédiaire à une autre (où l'aspect bidirectionnel est particulièrement intéressant).

Instrumentation des métamodèles Il est possible de définir des métaoutils qui ajoutent un comportement exécutable à un métamodèle ou à une famille de métamodèles. Cette instrumentation facilite notamment l'expression de tâches récurrentes. Elle requiert un environnement capable d'interpréter la description des comportements ajoutés.

Kermeta permet d'ajouter ou de modifier des comportements via la notion d'aspect. La programmation par aspect est un moyen élégant de séparer les préoccupations tout en favorisant la réutilisation de code. L'idée de base est d'exprimer les attributs et comportements communs à chaque intention spécifique. Un problème modélisé est alors énoncé par un *tissage* explicite des différents aspects correspondants à la composition de ses intentions. L'instrumentation par des aspects permet donc de simplifier l'expression de transformations complexes en « décorant » les entités des métamodèles par des comportements dédiés à l'intention de chaque transformation.

Adapter une représentation intermédiaire arborescente en un graphe acyclique (DAG) est un exemple concret de transformation simplifiée par une instrumentation de leurs métamodèles respectifs. Il est important de noter que transformer un arbre en graphe est un problème classique et que les aspects utilitaires instrumentant les métamodèles sont réutilisables dans de multiples autres contextes.

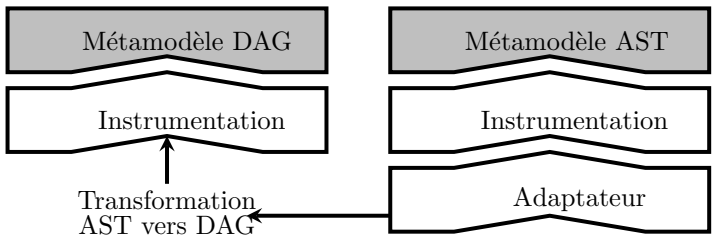


FIGURE 7.8 – Transformation M2M d'un AST en DAG en instrumentant les métamodèles. Les couches d'aspects favorisent la capitalisation des comportements.

Afin de favoriser cette réutilisation, il est préférable de diviser l'instrumentation en différentes couches d'aspects, comme illustré par la figure 7.8. Les aspects utilitaires correspondent aux fonctionnalités utiles à d'autres transformations. L'aspect adaptateur ajoute une référence explicite qui évite le recours à une table de correspondance coûteuse pour associer les éléments de l'AST à ceux du DAG.

L'instrumentation des métamodèles introduit des concepts puissants, qui permettent de simplifier l'expression de transformations en favorisant la capitalisation et l'efficacité de comportements. Cependant, cette instrumentation induit un surcoût important en termes de durée d'exécution. Ce faible niveau de performance est issu des environnements de métaprogrammation tels que Kermeta qui peinent, actuellement, à ouvrir et transformer des modèles contenant plusieurs dizaines de milliers d'éléments. Si nous sommes convaincus que l'instrumentation des métamodèles améliore significativement la flexibilité et la maintenabilité des transformations, les problèmes de passage à l'échelle constituent aujourd'hui un frein important à la démocratisation de cette instrumentation au sein d'un compilateur.

7.3.4 Synthèse des réponses de l'IDM aux défis d'un compilateur optimisant

L'IDM offre des solutions avancées aux transformations M2M et M2T, qui en font une technologie très attractive pour la conception des composants d'un compilateur optimisant. Le schéma de la figure 7.9 cartographie les principales contributions de l'IDM au sein de l'infrastructure de compilation GeCoS et distingue les outils mis en œuvre dans le contexte de cette thèse. Le cercle central correspond à l'intégration des outils existants de l'IDM au coeur du flot de compilation, le second cercle représente les principaux métaoutils définis pour assister les concepteurs dans des tâches inhérentes aux compilateurs. Nous résumons ici brièvement les différentes réponses de l'IDM aux défis décrits dans la section 7.2.

Les approches génératives basées sur la spécification d'un métamodèle mènent à un code homogène et conforme aux bonnes pratiques logicielles (défi 7.2.1). L'IDM contribue également à la validation des représentations intermédiaires et des transformations en assurant le respect de propriétés structurelles simples et plus complexes, via des requêtes OCL. Elle offre donc une réponse au défi 7.2.2 et une aide significative pour répondre aux défis 7.2.3 et 7.2.5.

L'utilisation et la définition de métaoutils, par génération ou instrumentation des métamodèles, simplifient la conception de transformations et de requêtes complexes sur les représentations intermédiaires d'un compilateur (défi 7.2.3). Ces métaoutils permettent également d'automatiser la création

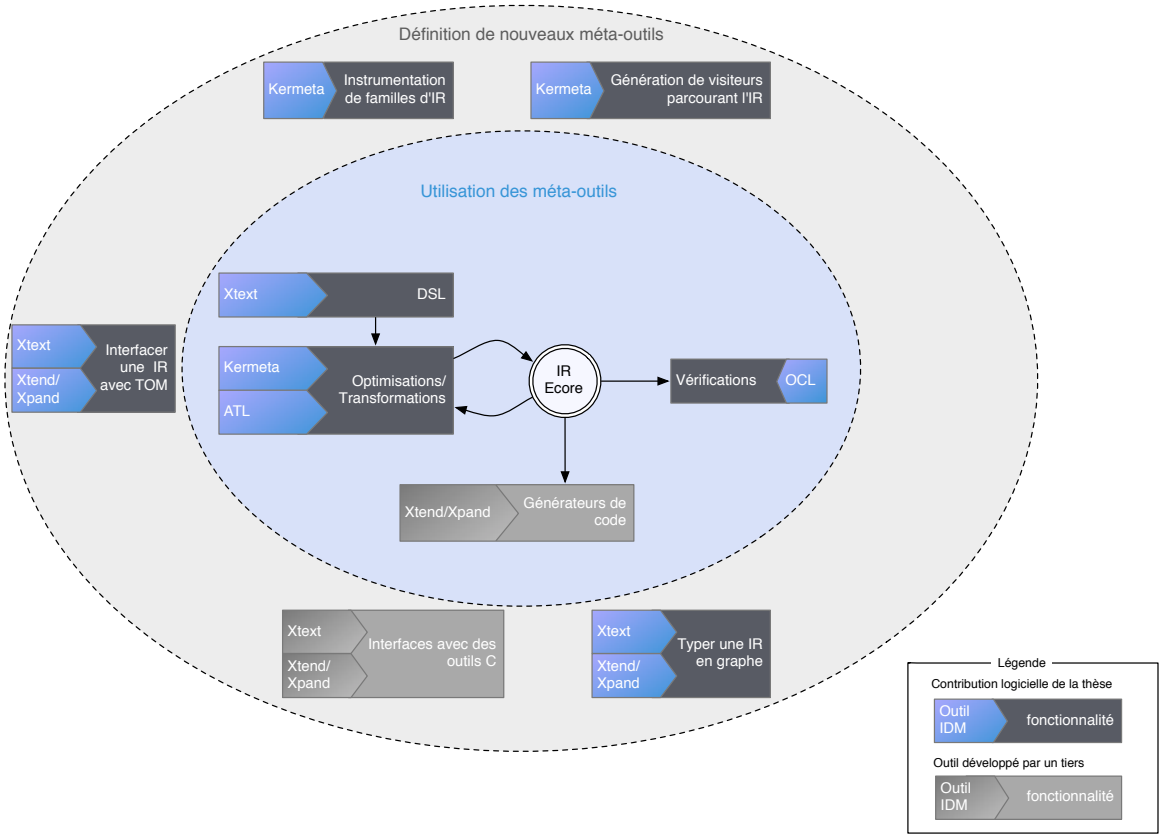


FIGURE 7.9 – Intégration d'outils de l'IDM dans un flot de compilation GeCoS.

d'interfaces avec des outils externes (défi 7.2.4) par des descriptions simples, effectuées dans des langages dédiés. La définition de ces langages dédiés profite des facilités offertes par l'IDM dans la description de DSL et de génération de code qui répond partiellement aux défis 7.2.6 and 7.2.7.

7.4 Applicabilité de l'IDM

S'il est intéressant d'utiliser l'IDM lors de la conception d'une infrastructure de compilation optimisante, il est important de comprendre que des attentes irréalistes sur l'IDM risquent de conduire à des frustrations et à un échec. Certaines de ces frustrations peuvent également venir d'une mauvaise appréhension des compétences logicielles requises pour utiliser efficacement l'IDM.

7.4.1 Clarification des objectifs

L'IDM est une solution très attractive pour de nombreux problèmes inhérents à la conception de logiciels complexes. Dans de tels systèmes, les modèles peuvent être utilisés pour décrire les logiciels à différents niveaux d'abstraction. Les technologies de l'IDM permettent, quant à elles, de créer et de manipuler ces modèles. Les modèles permettent donc de décrire des solutions aux problèmes. Cependant, les compilateurs, comme beaucoup d'infrastructures logicielles complexes, sont bien plus que de simples systèmes d'information. De nombreux problèmes traités impliquent des algorithmes combinatoires très complexes qui sont au-delà des objectifs ciblés par les technologies de l'IDM.

De plus, ces algorithmes reposent souvent sur des fondements théoriques avancés, qui requièrent un haut degré d'expertise de la part du concepteur. Par exemple, modéliser le jeu d'instructions d'un processeur ne suffit pas à compiler efficacement un programme. Autrement dit, modéliser le problème n'est pas résoudre le problème.

Une idée clé pour une utilisation efficace de l'IDM est de comprendre que les modèles servent des objectifs spécifiques et qu'un bon modèle est un modèle qui sert efficacement ses objectifs. Par exemple, dans un compilateur, la bonne représentation intermédiaire est celle qui décrit le programme dans la forme la plus efficacement analysable par des étapes particulières du flot de compilation.

Les concepteurs doivent être conscients que les outils de l'IDM ne garantissent pas que les modèles créés sont adaptés à leurs objectifs. Ceci est le rôle de la créativité humaine et les technologies de l'IDM sont conçues pour *améliorer* et non *remplacer* cette créativité. L'IDM offre les outils aux concepteurs pour décrire et analyser leurs modèles dans l'optique de convaincre que ces modèles sont effectivement les plus adaptés. Dans le cadre de la compilation optimisante, cela signifie que les technologies de l'IDM ne permettent pas de produire de meilleurs modèles de programmes.

7.4.2 Prérequis

Il est important que les membres de l'équipe soient aptes à travailler à un niveau d'abstraction élevé. Plus précisément, ils doivent avoir de solides compétences de modélisation. Modélisation est ici à prendre au sens large du terme, c'est-à-dire allant de la modélisation mathématique aux approches type UML. De plus, les concepteurs doivent être à l'aise avec les concepts de la programmation orientée objet. Cette condition n'est pas toujours respectée par de jeunes ingénieurs en électronique qui, malgré une formation sur ce paradigme de programmation, ont souvent une connaissance trop approximative de ses concepts et de ses bonnes pratiques tels que le polymorphisme et les patrons de conception.

Pour obtenir un code exécutable, le développeur IDM doit tout d'abord modéliser son logiciel avant de le générer. Cette dissociation des étapes, même si elle est la plupart du temps souhaitable, introduit un surcoût lié à l'utilisation des outils qui peut ralentir les premiers développements. De plus, si la plupart des outils de l'IDM ont un faible coût d'entrée pour une utilisation standard, le prix à payer pour un outil flexible est souvent une importante complexité de l'infrastructure. Si les technologies de l'IDM permettent d'obtenir très rapidement des prototypes, atteindre un niveau de maturité industriel requiert un investissement supplémentaire important pour maitrîser chaque technologie.

Les développeurs ayant une expérience dans la conception de logiciels complexes seront rapidement intéressés par l'IDM. Ces derniers sont conscients, en raison de leurs expériences précédentes, que la qualité de la conception d'un logiciel est critique et que le temps passé à modéliser un logiciel peut être plus long que celui passé à le mettre en œuvre. Ce type d'utilisateur sera rapidement convaincu de l'intérêt de travailler au niveau modèle. Cet intérêt n'est pas toujours aussi évident pour des développeurs moins expérimentés dont les réalisations sont souvent des applications à faible niveau de flexibilité et de capitalisation. Ce constat peut être étendu à des experts de domaines spécifiques qui, s'ils ont connaissance du faible niveau de qualité logicielle de leurs prototypes, seront parfois réticents à s'investir dans une approche plus qualitative, mais éloignée de leurs objectifs de recherche. D'après notre expérience, la démocratisation de l'IDM au sein d'une équipe travaillant sur la conception d'un

compilateur gagne énormément à profiter d'une ou plusieurs personnes « locomotives ». Celles-ci évaluent le panel des technologies disponibles avant d'influencer, de guider et de former le reste de l'équipe aux solutions les plus adaptées.

7.5 Conclusion

Dans ce chapitre, nous avons montré que la recherche sur la compilation optimisante peut aisément profiter de l'IDM en raison des aspects de modélisation inhérents aux compilateurs. Nous avons illustré les bénéfices de l'IDM en nous basant sur l'expérience acquise au cours du développement de l'infrastructure de compilation GeCoS .

Le bénéfice le plus évident est une homogénéisation des pratiques de développement pourtant difficile à atteindre dans un cadre académique. Les métamodèles offrent également une représentation abstraite d'un logiciel, ils documentent de nombreux choix de conception. Cet aspect est particulièrement intéressant dans un contexte de recherche, où les mises en œuvre restent souvent au stade de prototype non documenté. De plus, les métaoutils offrent une aide importante dans l'automatisation de tâches de développement coûteuses et sources d'erreurs. Enfin, nous avons observé que les approches génératives constituent de puissants facteurs de créativité, car ils facilitent le prototypage rapide et l'évaluation de nombreuses idées.

Si l'IDM est adaptée à la résolution de nombreux défis issus de la conception d'un compilateur optimisant, ce contexte d'utilisation apporte également de multiples axes de recherche pour la communauté de l'IDM. Tout d'abord, les optimisations d'un compilateur, mises en œuvre sous forme de transformations M2M, se doivent d'assurer le respect des propriétés sémantiques de la représentation intermédiaire transformée. Par conséquent, le test et la vérification des transformations de modèles nous apparaissent comme des défis très intéressants pour la communauté IDM.

De plus, si l'IDM offre maintenant des outils simplifiant significativement la définition de DSL, il devient urgent de prendre en compte la croissance rapide de leur nombre. Cette explosion est une conséquence directe de la spécificité intrinsèque à un DSL et aux facilités existantes pour le définir. Identifier des familles de langages permettrait de capitaliser les outils associés à chaque DSL (simulateur, vérification et générateurs) et donc de simplifier la gestion de la multiplicité des langages dédiés. De même, la capitalisation des transformations sur des familles de métamodèles est un thème important. Les approches développées dans le *model typing* [173] et le *model mapping* [42] constituent des perspectives intéressantes pour la capitalisation autour de la multiplicité des représentations intermédiaires et des DSL. Ces thématiques ne sont pas propres aux compilateurs, elles y sont néanmoins caractéristiques.

Enfin, il s'avère que l'utilisation de l'IDM au sein d'un compilateur peut poser des problèmes de passage à l'échelle. Dans des codes de production, la taille des modèles manipulés peut être de l'ordre de dizaines de milliers d'éléments, ce qui n'est pas toujours bien supporté par les outils de l'IDM (e.g., Kermeta). Ce constat, acceptable dans le contexte d'un compilateur optimisant axé recherche (où les algorithmes d'optimisation constituent généralement le coût critique), amène à considérer certaines techniques de l'IDM comme étant inadaptées à des compilateurs industriels.

Nous espérons que ces trois sujets : transformations préservant la sémantique, capitalisation des transformations et passage à l'échelle des outils motiveront de futures recherches.

Chapitre 8

ARCAdE : Un environnement orienté aspect pour la modélisation modulaire de problèmes de satisfaction de contraintes

Dans le chapitre précédent, nous nous sommes intéressés à l'utilisation de l'ingénierie dirigée par les modèles (IDM) dans le cadre du développement d'un compilateur optimisant. Une des conclusions de ce chapitre (et qui dépasse le cadre des compilateurs optimisants) est que les outils de l'IDM constituent de puissants facteurs de créativité en réduisant considérablement la difficulté de tâches complexes. L'outil présenté dans ce chapitre n'aurait pas été réalisable (ni même envisagé) dans le cadre de cette thèse sans les facilités offertes par l'IDM.

Cet outil est issu d'un réel besoin de capitalisation des différents problèmes d'optimisation énoncés dans le chapitre 3. Il garantit la mise en œuvre de bonnes pratiques (i.e., indépendance de la modélisation du problème et extensibilité) qui facilitent la conception, l'utilisation et la communication des différents modèles de contraintes.

Sommaire

8.1 Introduction

La programmation par contraintes (PPC [1], cf. chapitre 2) offre un formalisme dont le niveau d'abstraction est suffisamment élevé pour exprimer de manière compacte de complexes problèmes d'optimisation. Dans ces problèmes, les variables à déterminer sont associées à des domaines finis dont les valeurs possibles sont restreintes par un ensemble de contraintes.

La conception d'un CSP s'apparente à une étape de modélisation. Cependant, contrairement à la modélisation d'un programme, l'objectif n'est pas d'abstraire un comportement exécutable, mais plutôt d'identifier les contraintes issues des interactions entre les différentes entités du problème. Cette différence entre la programmation par contraintes et la programmation standard constitue un point particulièrement difficile à appréhender pour les nouveaux utilisateurs. À cette difficulté inhérente au paradigme PPC, s'ajoute la multiplicité des solveurs de contraintes (e.g. JaCoP [177], Gecode [168], ECLiPSe [10], ILOG [93] et Choco [101]) et des concepts théoriques associés qui compliquent significativement la tâche de modélisation d'un concepteur non expert du domaine. Ce constat a favorisé l'émergence d'environnements dédiés à la modélisation de CSP (e.g., MiniZinc [139] et s-COMMA [39]). Ces environnements sont indépendants des solveurs et s'appuient sur des langages spécifiques qui facilitent la description d'un CSP par une syntaxe simple et réduite.

Les travaux présentés dans ce chapitre s'inscrivent dans la continuité de ces environnements de modélisation en s'intéressant plus particulièrement à la capitalisation des modèles de contraintes. En effet, si, conceptuellement, la programmation par contraintes offre un cadre dans lequel plusieurs problèmes peuvent être facilement combinés (il suffit de fusionner les contraintes), aucun environnement de modélisation existant ne permet de le faire simplement. Afin de répondre à ce problème, nous avons conçu ARCAdE [2], un environnement permettant de modéliser des CSP modulaires en se basant sur la notion d'aspects [107]. Les aspects permettront, intuitivement, de composer un problème complexe en assemblant des pièces de *puzzle* décrites dans les sous-problèmes. Les contributions d'ARCAdE sont les suivantes.

1. Nous proposons un nouveau langage permettant de modéliser de manière « naturelle »et modulaire un CSP. Chaque entité d'un problème est un objet contenant un ensemble de variables et de règles définissant ses interactions avec les autres entités du problème. Les entités modélisées supportent l'héritage multiple et la notion d'aspect afin de composer très simplement des problèmes complexes à partir de sous-problèmes.

2. Afin de conserver l'indépendance vis-à-vis du solveur PPC, nous générons le code spécifique à un solveur à partir du CSP modélisé. Le flot de génération est aisément adaptable à n'importe quel solveur Java et utilise les outils de l'IDM pour générer la structure et le comportement du code.

3. Un environnement intégré à Eclipse [3] pour assister l'utilisateur dans la modélisation des entités et dans la description des contraintes. La simplicité d'utilisation est difficilement quantifiable, elle est néanmoins critique dans le processus d'appropriation d'un outil par un utilisateur. Le DSL d'ARCAdE est conçu avec Xtext [4] (cf. chapitre 7, page 169) et bénéficie donc de nombreuses

1. Programmation par constraintes
2. Aspect oriented constraint programming environement
3. http://www.eclipse.org
4. http://www.eclipse.org/Xtext/

facilités d'édition et d'analyse simplifiant considérablement son utilisation.

La prochaine section de ce chapitre positionne plus précisément notre approche vis-à-vis des travaux existants sur la modélisation de CSP. Ensuite, nous nous intéresserons à la modélisation modulaire de CSP en faisant notamment apparaître que la démarche de modélisation avec ARCAdE est proche d'un raisonnement naturel. Les différents concepts et éléments de syntaxe sont également détaillés. Nous poursuivrons par l'étude de différents problèmes d'ordonnancement dont la modularité est avantageusement exploitée pour composer un problème complexe de gestion de projet. Enfin, nous présenterons le flot de génération de code ainsi que ses facilités d'adaptation à un solveur existant.

8.2 Travaux liés

MiniZinc [139] est un langage de modélisation dissocié du solveur utilisé. De plus, le CSP modélisé est associé à un jeu de données décrit dans un fichier séparé. Leur association forme une instance d'une représentation intermédiaire (FlatZinc) utilisée en entrée des solveurs. MiniZinc offre donc une indépendance vis-à-vis des solveurs et des données initialisant les CSP modélisés. Il constitue actuellement un langage de référence permettant notamment de comparer l'efficacité (*MiniZinc Challenge*) des différents solveurs existants. La syntaxe de MiniZinc est fortement inspirée d'OPL [185]. Elle dispose en effet de structures de contrôle (boucles, conditions, etc.) pour exprimer, de manière programmative, un ensemble de contraintes sur des variables typées. À la différence d'OPL, MiniZinc permet d'ajouter de nouveaux types à ceux supportés nativement (entiers, flottants, ensembles). Un type personnalisé est une composition de variables éventuellement nommées et s'apparente donc intuitivement à une structure en langage C. De plus, chaque type personnalisé peut être soumis à des contraintes sur ses variables internes. La modélisation de problèmes complexes est donc facilitée par la description de nouveaux types contraints, spécifiques à une famille de problèmes. D'autre part, toujours dans cette optique de capitalisation et de lisibilité, il est possible de définir des prédicats et des fonctions qui correspondent éventuellement à des contraintes globales supportées par un sous-ensemble des solveurs cibles. Cependant, la notion de type n'est pas orientée objet, il n'est donc pas possible de spécialiser un type, ni d'y ajouter du comportement (en dehors de ses contraintes primitives).

D'autres langages de modélisation s'intéressent à cette vision objet qui simplifie la description et l'appréhension de problèmes complexes. L'environnement autour du langage s-COMMA [39] est basé sur l'IDM et propose de modéliser un CSP dans un éditeur graphique. Les entités d'un problème sont des objets dont les interactions sont contraintes par des fonctions décrites dans un langage proche de MiniZinc. Les données d'une instance de problème proviennent également d'un fichier externe et des transformations M2M (écrites en ATL) produisent une représentation intermédiaire pour chaque solveur. Il est alors possible de spécialiser des entités d'un problème pour modéliser un problème plus particulier impliquant de nouvelles entités, variables et contraintes. Dans la même optique, COB [100] utilise la notion d'objets contraints et propose notamment CUML, une extension d'UML [179] pour décrire des objets constitués d'attributs, de contraintes et de prédicats. Cependant COB est uniquement axé sur la programmation logique par contraintes et est dépendant du solveur CLP(R) [98].

Dans cette logique d'abstraction, ARCAdE propose d'aller plus loin dans la capitalisation des modèles de contraintes en se basant sur la notion d'aspects [107]. L'objectif est d'identifier et de

modéliser des problèmes indépendants qui peuvent être composés en des problèmes plus complexes. De plus, à la différence des autres approches de modélisation PPC, les instances de problèmes ne sont pas construites à partir d'un fichier externe de données. Elles reposent sur un typage explicite des entités concrètes d'une instance de problème. Autrement dit, l'utilisateur associe à chaque type modélisé (appelé *acteur*), une classe concrète issue de l'implémentation qu'il souhaite soumettre au CSP. Ce lien direct évite une étape de sérialisation des données et facilite l'analyse et la correction éventuelle d'erreurs apparaissant lors de l'initialisation du problème modélisé.

D'autre part, les langages MiniZinc et s-COMMA ne disposent pas, à notre connaissance, d'environnement d'édition pour assister l'utilisateur dans la saisie des contraintes du problème modélisé. Pourtant, un éditeur disposant d'une analyse des erreurs et d'autocomplétion contextuelle offre un environnement d'expérimentation intéressant pour un utilisateur non expert du domaine. ARCAdE est un environnement basé sur l'IDM et intégré à Eclipse qui simplifie la modélisation modulaire de CSP en profitant des facilités offertes par Xtext (cf. paragraphe 7.3.2), il s'adresse à des utilisateurs habitués à la programmation-objet, mais qui ne sont pas nécessairement experts en PPC.

8.3 Modélisation modulaire de CSP

Les problèmes de satisfaction de contraintes ont souvent une nature modulaire. Ainsi, un problème d'ordonnancement peut être étendu par différentes préoccupations (e.g., allocation de ressources, répartition de charge). Ces préoccupations sont indépendantes et leurs différentes combinaisons constituent autant de nouveaux problèmes. Dans cette section, nous nous intéressons à la manière de modéliser et de résoudre un CSP modulaire avec ARCAdE.

8.3.1 Identification, modélisation et instanciation des acteurs d'un problème

La programmation par rôle [209] est un paradigme de programmation qui repose sur l'identification des différents rôles, définis par leurs interfaces et leurs fonctionnalités, qu'un même objet peut assumer dans un système logiciel. Les fonctionnalités du système sont alors réparties en plusieurs rôles et chaque objet assumera un ou plusieurs rôles au cours de sa durée de vie.

Cette notion de rôle est finalement très proche d'un raisonnement d'élaboration d'un CSP. En effet, formuler un CSP consiste à identifier les différentes entités du problème (e.g., dans un problème d'ordonnancement simple, la seule entité est une tâche dont la date de début est à déterminer) et à caractériser leurs interactions et contraintes (e.g., tâches dépendantes). Ces entités peuvent être assimilées à des rôles qui seront assumés par des objets issus de l'implémentation concrète analysée (e.g., nœuds d'un graphe). Une fois les rôles identifiés, les contraintes du problème correspondent aux propriétés de ces rôles ainsi qu'à leurs interactions. Un des objectifs d'ARCAdE est d'offrir un environnement de modélisation très proche d'un raisonnement naturel : un problème est décrit par un ensemble d'acteurs dont les interactions correspondent à des ensembles de contraintes.

Chaque rôle d'un problème est décrit par un type abstrait appelé *acteur* (définition 21). Un acteur est constitué d'un ensemble de variables entières à domaine fini et de *règles* (définition 22) exprimant des restrictions sur les valeurs possibles de ces variables.

```
1  actor Task {
2    var start : "Start time of a task";
3    var duration : "Total duration of a task";
4    var end : "End of a task";
5
6    where {
7      end = start + duration;
8    }
9
10   rule dependency(Task previous) {
11     start >= previous.start + previous.duration;
12   }
13 }
```

FIGURE 8.1 – Modélisation ARCAdE d'une tâche caractérisée par trois variables *start*, *duration* et *end* qui déterminent respectivement le début, la durée et la fin d'une tâche.

Définition 21 (Acteur) *Un acteur est un type capturant les contraintes sémantiques d'un ensemble d'entités concrètes répondant à la même intention pour un CSP donné.*

Définition 22 (Règle) *Une règle est une fonction qui produit l'ensemble des contraintes d'une interaction particulière entre l'acteur de cette règle et d'autres acteurs fournis en paramètre.*

La figure 8.1 est un extrait d'un fichier ARCAdE modélisant une tâche d'ordonnancement. Celle-ci est décrite dans un acteur comportant notamment trois variables à évaluer lors de la recherche d'une solution. Le domaine par défaut de chacune de ces variables est $\mathbb{N}$ [5], il s'avère en effet le plus répandu en pratique. Les valeurs possibles de ces variables sont restreintes par deux règles. La première règle (ligne 6) est propre à la tâche, elle contraint sa date de fin en fonction de son début et de sa durée. La seconde règle (ligne 10) correspond aux contraintes imposées quand une tâche doit être précédée par une autre fournie en paramètre.

Les acteurs définissent des types supportant des fonctionnalités logicielles avancées (e.g., aspects, héritage multiple, surcharge et redéfinition des règles) afin de factoriser les modèles de contraintes déployés dans de multiples contextes. La mise en œuvre de ces fonctionnalités est facilitée par la nature d'un CSP qui élimine notamment le problème du diamant issu de l'héritage multiple.

La figure 8.2 illustre le problème du diamant à travers une classe D qui hérite de deux autres classes (B et C) définissant le même prototype de méthode m. Dans ce cas, le comportement de m dans D requiert un choix explicite entre celui défini dans B ou C. Or, par définition, un CSP est constitué d'un ensemble de variables et d'un ensemble de contraintes. L'ordre dans lequel sont imposées ces contraintes n'a aucune importance et si une contrainte s'avère être exclusive avec le reste du problème alors celui-ci reste dans une forme cohérente, mais n'a pas de solution. Ces caractéristiques lèvent toute ambiguïté puisque les règles définies dans chaque acteur produisent toutes des ensembles de contraintes. Ainsi, une règle m de D qui est définie à la fois dans l'acteur B et l'acteur C, produit à la fois les contraintes de m dans B et celles de m dans C. La sémantique spécifique aux CSP ne requiert donc aucun choix explicite de la part du concepteur : si les contraintes de B et C sont incompatibles, l'instance du CSP final n'aura pas de solution. Autrement dit, les comportements issus de l'héritage

5. Abus de notation par souci de lisibilité : les domaines étant finis, la borne supérieure est une constante entière représentant l'infini positif.

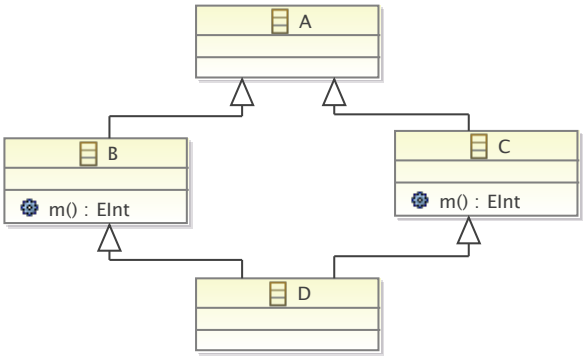


FIGURE 8.2 – Problème du diamant : quel est le comportement de m dans D ?

multiple sont implicites et l'analyse de leur cohérence est déportée sur le solveur, ce processus est donc transparent à l'utilisateur.

Une autre caractéristique d'ARCAdE est d'offrir un lien direct entre une implémentation concrète et les modèles de contraintes utilisés pour résoudre un problème de satisfaction ou d'optimisation. Ce lien est mis en œuvre à travers le polymorphisme paramétrique des acteurs : les classes concrètes d'un problème modélisé sont des paramètres des acteurs. Le polymorphisme paramétrique (ou généricité) consiste à définir des classes ou des méthodes paramétrées par des types génériques. Ainsi, une classe A peut définir un attribut sur un type générique T si ce dernier est un paramètre de A (en Java, on note `A<T>`). Dans notre cas, la classe d'un acteur modélisant une tâche pour un problème d'ordonnancement est `Task<A>` où A est un type générique qui symbolise n'importe quelle classe concrète des objets ordonnancés.

Instancier un CSP modélisé pour une implémentation concrète nécessite trois étapes de la part du concepteur.

1. Typer les classes concrètes par des acteurs. Par exemple, si l'on cherche à ordonnancer un graphe, les nœuds du graphe sont typés en acteur tâches. Si la classe d'un nœud est `Node`, la classe de l'acteur des tâches est alors `Task<Node>`.
2. Initialiser, si besoin, les domaines des variables en fonction de l'état des objets concrets. Pour un ordonnancement de graphe, cela consiste à initialiser les domaines des variables *duration* (cf. figure 8.1) à une durée constante pour chaque nœud.
3. Imposer les contraintes du problème en appelant les règles définies dans les acteurs. L'appel explicite des règles associe les interactions possibles des acteurs à la sémantique concrète des objets. Par exemple, toujours dans le cas d'un ordonnancement de graphe, la règle de dépendance entre deux tâches (cf. figure 8.1, ligne 10) est appelée pour chaque lien du graphe. L'appel de cette règle est fait sur l'acteur correspondant au nœud destination, son unique paramètre est le nœud source du lien (l'acteur correspondant est récupéré automatiquement).

Le code Java de la figure 8.3 illustre l'instanciation d'un problème d'ordonnancement (cf. figure 8.1) à une implémentation concrète d'un graphe. Les nœuds constituent les éléments à ordonnancer, ils sont typés en acteur `Task` par la création (ligne 1) d'un objet `SolveScheduling` paramétré par la classe `Node`. Cet objet contient toutes les instances des acteurs du problème, il se charge également d'associer chaque objet concret (`Node`) à son instance d'acteur respective (`Task<Node>`). Par

```
1  SolveScheduling<Node> scheduling = SchedulingFactory.eINSTANCE.createSolveScheduling();
2
3  for (Node n : graph.getNodes()) {
4    int d = n.getLatency();
5
6    ITask<Node> task = scheduling.getTask(n);
7    task.getDuration().setDomain(d,d);
8
9    for(Node predecessor: n.getPredecessors()){
10     scheduling.impose(task.buildDependencyConstraints(predecessor));
11   }
12 }
```

FIGURE 8.3 – Instantiation d'un problème d'ordonnancement pour un graphe. La sémantique du graphe est exprimée en Java par un appel explicite de la règle de dépendance entre deux tâches (cf. figure 8.1).

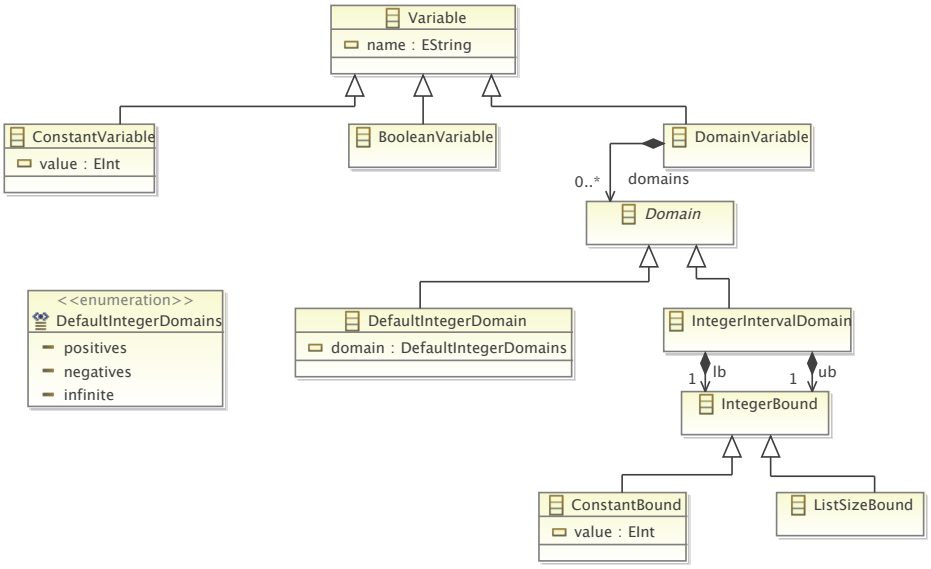
exemple, la ligne 6 récupère l'instance de l'acteur `Task<Node>` d'un nœud du graphe à ordonnancer. S'il s'agit du premier accès à l'acteur de ce nœud, un nouvel objet `Task<Node>` est automatiquement créé et initialisé en se basant sur la description des domaines de variables de cet acteur.

Dans l'exemple (cf. figure 8.1), aucun domaine n'est associé explicitement aux variables de l'acteur `Task`, les domaines sont alors $\mathbb{N}$ (domaine par défaut). Si le domaine d'une variable varie selon l'instance de son acteur, il est nécessaire d'initialiser son domaine en fonction des états et des contextes des objets concrets. C'est le cas des durées des tâches qui sont initialisées à la ligne 7 à la valeur fournie par chaque nœud.

Le code initialisant le domaine d'une variable dépend évidemment du solveur ciblé par le générateur de code d'ARCAdE (ici il s'agit du solveur JaCoP). Une fois que tous les domaines des variables sont initialisés, l'utilisateur appelle explicitement les différentes règles à appliquer. L'objet `SolveScheduling` contient l'ensemble des contraintes imposées pour chaque règle appliquée. Dans l'exemple, la seule règle appliquée (ligne 10) est celle qui exprime une dépendance entre un nœud et chacun de ses prédécesseurs.

8.3.2 Déclaration des variables

Les variables d'un problème sont déclarées comme des attributs des acteurs ou comme des variables locales à une zone de description de contraintes (e.g., règle). ARCAdE supporte trois types de variables : constantes, booléennes et variables à domaines finis. La figure 8.4 présente la déclaration et l'initialisation des trois types de variables. Les variables `DomainVariable` contiennent une liste de domaines de valeurs. Outre les domaines définis par des intervalles, il existe des domaines par défaut qui simplifient la déclaration des variables : $\mathbb{N}$, $\mathbb{Z}^-$ et $\mathbb{Z}$. Comme évoqué précédemment, si aucun domaine n'est associé à une variable alors le domaine par défaut est $\mathbb{N}$. D'autre part, il est possible d'utiliser la taille d'une liste comme une borne d'un intervalle de domaine. Cette construction s'avère notamment utile lors de la déclaration d'une variable locale servant d'indice dans une contrainte *Element* (cf. section 2).



⟨*ConstantVariable*⟩ → '`int`' ⟨*ID*⟩ '`;`'
| '`int`' ⟨*name*⟩ '`=`' ⟨*value*⟩ '`;`'

⟨*BooleanVariable*⟩ → '`boolean`' ⟨*ID*⟩ '`;`'

⟨*DomainVariable*⟩ → '`var`' ⟨*ID*⟩ '`;`'
| '`var`' ⟨*ID*⟩ '`::`' ⟨*Domain*⟩ ('`,`'⟨*Domain*⟩)* '`;`'

⟨*Domain*⟩ → DefaultIntegerDomain | IntegerIntervalDomain

⟨*DefaultIntegerDomains*⟩ N | Z- | Z

⟨*IntegerIntervalDomain*⟩ → '`[`'⟨*lb*⟩ '`..`' ⟨*ub*⟩ '`]`'

⟨*IntegerBound*⟩ → ConstantBound | ListSizeBound

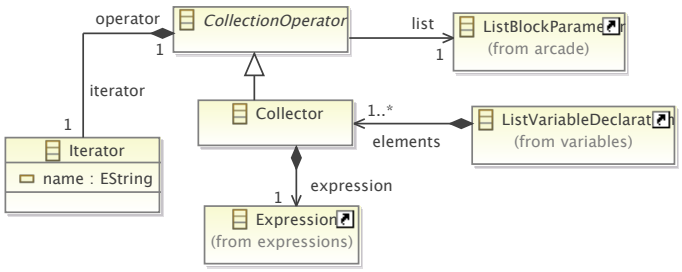
⟨*ListSizeBound*⟩ → ⟨*List*⟩'`.`' '`size`'

⟨*ConstantBound*⟩ → ⟨*value*⟩

FIGURE 8.4 – Déclaration et initialisation des variables.

La déclaration d'une variable peut être annotée d'informations spécifiques utilisées par le solveur cible. Par exemple, le choix du sélecteur des valeurs (e.g., par ordre croissant) dans un domaine peut être indiqué explicitement par une annotation sur la déclaration de la variable associée (e.g., `@indomain->"InDomainMin"`). Le recours à une annotation permet d'exprimer les spécificités de certains solveurs tout en restant indépendant : si un solveur ne la supporte pas, il n'effectuera aucun traitement particulier.

Les contraintes portent parfois sur de multiples variables. Les paramètres de ces contraintes sont donc des listes de variables. La figure 8.5 présente la structure et la syntaxe du collecteur permettant de construire ces listes de variables à partir de listes d'acteurs (`ListBlockParameter`) fournies en paramètres des règles. Un itérateur permet de référencer les variables de chaque acteur de cette liste



⟨*ListVariableDeclaration*⟩ → ‘`list`’ ⟨*ID*⟩ ‘`=`’ ⟨*Collector*⟩ (‘`U`’ ⟨*Collector*⟩)* ‘`;`’

⟨*Collector*⟩ → ⟨*ListBlockParameter*⟩ ‘`.`’ ‘`collect`’ ‘`(`’⟨*Iterator*⟩ ‘`|`’ ⟨*Expression*⟩‘`)`’

⟨*Iterator*⟩ → ⟨*ID*⟩

FIGURE 8.5 – Déclaration et initialisation d'une liste de variables.

dans une expression arithmétique dont le résultat constitue la variable collectée :

```
list ends = tasks1.collect(t1 | t1.start + t1.duration);
```

De plus, plusieurs collections peuvent être concaténées (opérateur `U`) en une unique liste de variables :

```
list ends = tasks1.collect(t1 | t1.end)U tasks2.collect(t2 | t2.end);
```

8.3.3 Déclaration des contraintes

Les contraintes d'un problème sont définies dans des zones spécifiques (règles et stratégies). Les contraintes portent sur les variables des acteurs qui sont fournis en paramètre de ces zones ou encore sur des variables locales. On distingue deux familles de contraintes :

- Contraintes arithmétiques : expressions arithmétiques associées à un opérateur de comparaison. Les contraintes résultantes constituent des arbres de contraintes primitives dont les techniques de filtrage s'appuient sur des consistances d'arcs. Par exemple :

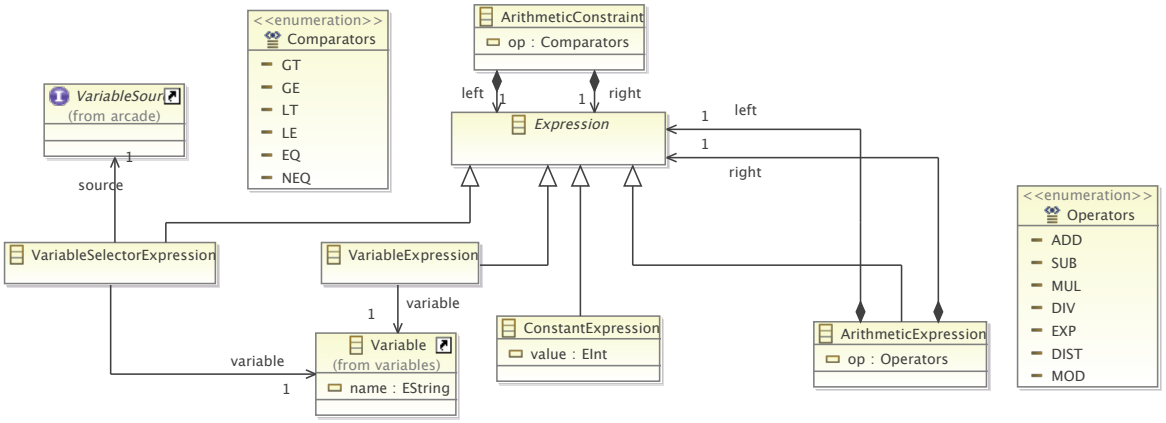
$$x * y > z * (a + b)$$

- Contraintes spécifiques : contraintes dont les techniques de filtrage sont spécifiques à la sémantique de la contrainte. Par exemple :

$$v = sum(vars)$$

Les contraintes arithmétiques, dont la structure et la syntaxe sont présentées dans la figure 8.6, permettent d'imposer des relations de comparaison ($>$, $<$, $\geq$, $\leq$, $\neq$ et $=$) entre des arbres d'expressions arithmétiques. De même que dans les DSL existants (e.g., MiniZinc et s-COMMA), la syntaxe est proche d'une notation mathématique et facilite l'expression de contraintes arithmétiques complexes. À titre de comparaison, la contrainte arithmétique décrite dans ARCAdE par $x * y > z * (a + b)$ correspond aux contraintes JaCoP de la figure 8.7. Chaque expression y est décomposée en une ou plusieurs contraintes primitives liées par des variables temporaires.

Les variables référencées dans les expressions peuvent être des variables locales ou des variables de l'acteur contenant la règle éditée. Elles peuvent également provenir d'une source externe (e.g., acteur



⟨*ArithmeticConsraint*⟩ → ⟨*Expression*⟩ ⟨*Comparators*⟩ ⟨*Expression*⟩';'

⟨*Expression*⟩ → ArithmeticExpression | PrimitiveExpression

⟨*ArithmeticExpression*⟩ → ⟨*PrimitiveExpression*⟩ ⟨*Operators*⟩ ⟨*PrimitiveExpression*⟩

⟨*PrimitiveExpression*⟩ → '(' Expression ')' | VariableExpression | ConstantExpression | VariableSelectorExpression

FIGURE 8.6 – Structure et syntaxe des contraintes arithmétiques.

```
1 Variable a,b,tmp1,right,left;
2 Constraint c1 = new XplusYeqZ(a,b,tmp1);
3 Constraint c2 = new XmulYeqZ(z,tmp1,right);
4 Constraint c3 = new XmulYeqZ(x,y,left);
5 Constraint c4 = new XgtY(left,right);
```

FIGURE 8.7 – Contraintes JaCoP pour la contrainte arithmétique $x * y > z * (a + b)$.

fourni en paramètre, itérateur d'une collection). Une variable est alors référencée par une expression de type `VariableSelectorExpression`.

Les autres contraintes supportées par ARCAdE sont celles définies dans la spécification JSR331 [6] (c.-à-d., *Element*, *Sum*, *Min*, *Max*, *AllDifferent*, *Card*, *IfThenElse* et *Reified*) proposant une API Java pour la programmation par contraintes. Les contraintes globales *Diff2* et *Cumulative* sont supportées malgré leur absence dans la spécification JSR331, car elles sont utilisées intensivement dans la modélisation des différents problèmes d'ordonnancement et de partage de ressources.

8.3.4 Composition des aspects d'un problème

Certains problèmes d'optimisation (e.g., ordonnancement de tâches) sont récurrents et peuvent être appliqués à de multiples domaines (e.g., ordonnancement d'instructions, allocation de ressources, gestion de projet, etc.). Cependant, un domaine spécifique apporte généralement un ensemble de contraintes qui lui sont propres. Le problème résultant est alors une spécialisation du problème général,

6. http://jcp.org/en/jsr/detail?id=331

```
1  import "platform:/resource/org.arcade.examples/src/arcade/scheduling/scheduling.arcade";
2
3  aspect Task : Resource {
4    var resource : "Identifier of the allocated resource";
5  }
6
7  actor ResourceSet {
8    rule allocate(Task tasks*) {
9      list x = tasks.collect(t | t.start);
10     list y = tasks.collect(t | t.resource);
11     list w = tasks.collect(t | t.duration);
12     list h = tasks.collect(t | 1);
13     diff2(x, y, w, h);
14   }
15 }
```

FIGURE 8.8 – Modélisation d'un problème d'allocation de ressource en ajoutant un aspect sur une tâche pour identifier la ressource allouée.

une approche objet offre donc une capitalisation des différents niveaux de spécialisation des problèmes. De plus, l'héritage multiple (cf. sous-section 8.3.1) supporté par ARCAdE favorise la modularité en permettant par exemple à un problème de combiner deux problèmes différents. Toutefois, cette modularité se paie par une modélisation difficile si la hiérarchie d'héritage d'un problème composite est complexe.

La programmation orientée aspect [107] (AOP [7]) est utilisée pour ajouter des comportements (aspects) qui ne sont pas directement liés au domaine métier d'un logiciel (e.g., journalisation). Chaque aspect correspond à une intention spécifique et comporte les attributs et méthodes nécessaires à son expression. Ces caractéristiques sont particulièrement élégantes pour modéliser des CSP modulaires. En effet, les différentes variations d'un problème sont décrites sous forme d'aspects indépendants dont la composition amène à un problème plus complexe. Cependant, un inconvénient de l'AOP réside dans la difficulté de composer les différents aspects. Cette composition s'appuie sur la description explicite de points de jointure dont la complexité (e.g., dans le langage AspectJ [8]) a pour conséquence, en pratique, de limiter l'utilisation d'aspects intrusifs (c.-à-d. qui modifient la structure et le comportement des objets) pour des raisons de maintenabilité [136]. Ce constat empirique amène à s'intéresser à une AOP simplifiée qui n'autorise pas l'introduction dynamique de comportements. L'objectif est alors de préserver la modularité offerte par l'AOP (introduction statique de comportements) tout en facilitant l'analyse et le test du logiciel. Dans cette optique, ARCAdE permet de définir statiquement des aspects qui introduisent de nouvelles variables et de nouvelles règles dans un acteur. Le simple fait d'importer un aspect dans un problème modifie alors les comportements de l'acteur associé sans avoir à recourir à un mécanisme d'héritage.

La figure 8.8 illustre un exemple d'aspect ajouté à l'acteur d'une tâche (cf. figure 8.1) et qui modélise un problème d'ordonnancement sous contrainte de ressources. Le modèle ARCAdE du problème d'ordonnancement est importé à la ligne 1, les acteurs et stratégies qui y sont définis sont donc visibles dans le problème d'allocation. La déclaration de l'aspect *Resource* (ligne 3) référence l'acteur *Task*

7. Aspect Oriented Programming
8. http://www.eclipse.org/aspectj/

```
1  strategy searchOneSchedule() {
2    search ONE;
3  }
4  strategy searchAllSchedules() {
5    search ALL;
6  }
7  strategy minimizeTotalTime(Task tasks*) {
8    list ends = tasks.collect(t | t.end);
9    let var cost;
10   cost = max(ends);
11   minimize(cost);
12 }
```

FIGURE 8.9 – Stratégies de résolution d'un ordonnancement de tâches.

dont le comportement est étendu. Cet aspect se contente d'ajouter une nouvelle variable identifiant la ressource utilisée pour une tâche. Dans le problème d'allocation, les variables disponibles pour une tâche sont donc celles décrites dans la définition de l'acteur *Task* et dans celle de l'aspect *Resource*. Les contraintes de partage de ressource sont modélisées dans un nouvel acteur qui comporte une règle d'allocation. Celle-ci assure que les rectangles formés par le début et la durée de chaque tâche ne se chevaucheront pas s'ils sont alloués sur la même ressource.

8.3.5 Stratégies de résolution

Une fois le problème modélisé par des acteurs et leurs règles respectives, il est possible d'exprimer une ou plusieurs stratégies de résolution. Chacune de ces stratégies est réutilisable pour toutes les modélisations qui spécialiseront le problème où elles sont décrites. La résolution d'un CSP est la partie qui diffère le plus d'un solveur à l'autre. Elle constitue en effet une étape critique dont les performances sont conditionnées par l'adéquation entre le problème à résoudre (ou même l'instance du problème pour un jeu de données) et le parcours de l'espace de recherche. Néanmoins, il est possible d'extraire un comportement commun à la majorité des solveurs et de proposer un cadre restreint pour exprimer les objectifs d'une résolution éventuellement associée à une fonction de coût.

Une stratégie de résolution est, comme les règles des acteurs, une zone où il est possible d'imposer des contraintes et de déclarer des variables locales. En effet, une stratégie d'optimisation requiert souvent d'introduire des artefacts nécessaires à l'expression d'une fonction de coût. De plus, une stratégie de résolution est associée à une politique de recherche : recherche de toutes les solutions, recherche d'une solution ou encore recherche d'une solution optimale.

La figure 8.9 énonce les trois différentes politiques de recherche dans notre exemple d'ordonnancement de tâches. La première stratégie a pour objectif de déterminer un unique ordonnancement légal du problème alors que la seconde les identifiera tous. La dernière stratégie est une optimisation minimisant la durée totale de la solution. Elle est paramétrée par une liste de tâches dont les dates de fin sont collectées pour contraindre une variable de coût à la valeur maximale (ligne 10) de ces dates. Enfin, cette variable est utilisée dans la fonction d'optimisation qui est ici une minimisation. Il est également possible d'exprimer une maximisation en utilisant le mot clé *maximize* au lieu de *minimize*.

Pour certains problèmes complexes, l'utilisation de techniques de résolution systématique s'avère trop inefficace pour espérer les résoudre en un temps raisonnable. Dès lors, deux solutions sont envisageables :

- Guider la recherche dans la stratégie. Il existe de nombreuses techniques permettant d'adapter l'algorithme de recherche au problème. Il suffit parfois de choisir un ordre d'évaluation des variables et des valeurs pour améliorer significativement les performances de la résolution (cf. section 5.5). Si les performances ne sont toujours pas satisfaisantes, des techniques de recherche spécifiques ou encore des heuristiques peuvent être utilisées pour guider plus efficacement la recherche. Toutes ces techniques sont issues de résultats avancés de recherche opérationnelle et ne sont pas disponibles dans la majorité des solveurs. De plus, intégrer leurs sémantiques dans un DSL de modélisation induit un effort de conception important du fait de leurs spécificités.
- Mise en œuvre d'une recherche spécifique. La recherche d'une solution n'est plus générée à partir du fichier ARCAdE , elle est décrite dans le langage adapté au solveur cible. Il est alors possible d'optimiser la recherche en utilisant toutes les spécificités du solveur. Le prix à payer est évidemment une perte de généricité, la stratégie de recherche n'est plus indépendante du solveur.

De même que pour les déclarations de variables, il est possible dans ARCAdE de guider la stratégie de recherche en annotant la méthode de recherche ou d'optimisation. Si cette technique ne permet pas d'utiliser des techniques de résolution très spécifiques, il s'agit néanmoins d'une solution pragmatique, permettant de cibler plus efficacement certains solveurs tout en restant compatible avec les autres.

8.4 Etude de cas : ordonnancement de tâches

L'objectif de cette section est d'illustrer et d'étudier plus précisément la modélisation ARCAdE de différents problèmes d'ordonnancement. L'aspect modulaire des modélisations (ordonnancement, allocation et répartition de la charge de travail) permet de les composer avantageusement en un problème complexe de gestion de projet sous contrainte de ressources. Pour chaque problème, la modélisation ARCAdE et la structure du code généré sont détaillées. Dans cette étude de cas, le code est généré pour le solveur JaCoP et la cible concrète à ordonnancer est un graphe composé de nœuds (`Node`) et de liens (`Edge`).

8.4.1 Ordonnancement simple

Le problème modélisé est l'ordonnancement de tâche évoqué dans les paragraphes précédents. Il s'agit de déterminer l'ordonnancement de durée minimale d'un ensemble de tâches pouvant s'exécuter en parallèle, sous contrainte du respect de leurs dépendances. Le modèle ARCAdE complet de ce problème est décrit dans la figure 8.10, il contient l'acteur modélisant une tâche et la stratégie présentée dans la sous-section 8.3.5.

Le flot de génération de code d'ARCAdE (cf. section 8.5.1) utilise le métamétamodèle d'EMF (cf. chapitre 7) comme représentation intermédiaire. Pour chaque fichier ARCAdE, le problème produit un métamodèle définissant la structure du code exploitable par les utilisateurs. Le code java correspondant au métamodèle est ensuite généré en utilisant le flot de génération de code standard d'EMF.

```
problem scheduling;

actor Task {
  var start : "Start time of a task";
  var duration : "Total duration of a task";
  var end : "End of a task";

  where {
    end = start + duration;
  }
  rule dependency(Task previous) {
    start >= previous.start + previous.duration;
  }
}
strategy minimizeTotalTime(Task tasks*) {
  list ends = tasks.collect(t | t.end);
  let var cost;
  cost = max(ends);
  minimize(cost);
}
strategy searchOneSchedule() {
  search ONE;
}
```

FIGURE 8.10 – Modélisation ARCAdE d'un problème d'ordonnancement de tâche.

La structure du code généré pour le problème d'ordonnancement est présentée dans la figure 8.11. Afin de supporter l'héritage multiple des acteurs, les variables et règles d'un acteur définissent une interface. De cette manière, si un acteur hérite de multiples acteurs, il implémentera toutes leurs interfaces. Dans l'exemple, l'interface `ITask` correspond à l'unique acteur du problème, elle contient notamment les variables *start*, *duration* et *end* associées à chaque tâche. La classe `Task` implémente l'interface `ITask` pour ajouter une référence (*concreteObject*) sur un type générique nommé $A0$. Elle permet d'accéder à la concrétisation de l'acteur (i.e., un nœud) pour faciliter l'analyse de l'instance du problème. L'attribut *store* est spécifique à JaCoP, il référence l'objet chargé de contenir toutes les variables et contraintes JaCoP du problème.

Toutes les tâches sont contenues dans un objet `SolveScheduling` également responsable de l'enregistrement des contraintes du problème. La correspondance entre les acteurs et les objets concrets à ordonnancer est assurée par l'interface `ISolveScheduling` qui permet notamment de récupérer la tâche associée à chacun de ces objets. Dans le cas où aucune correspondance n'existe encore, une nouvelle tâche est créée et les domaines de ses variables *start*, *duration* et *end* sont initialisés à $\mathbb{N}$. De plus, cette interface contient toutes les stratégies décrites dans le problème.

Une fois le code du problème généré, instancier un problème concret consiste à suivre les trois étapes (cf. figure 8.3) détaillées dans la sous-section 8.3.1. Enfin, l'appel d'une des stratégies disponibles permet de résoudre l'instance du problème.

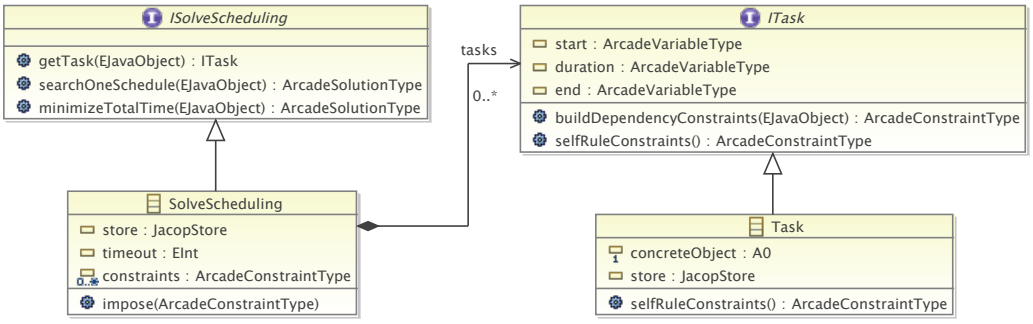


FIGURE 8.11 – Structure du code généré pour le problème d'ordonnancement.

```
problem allocation;
import "platform:/resource/org.arcade.examples/src/arcade/scheduling.arcade";

aspect Task : Resource {
  var resource;
}

actor ResourceSet {
  rule allocate(Task tasks*) {
    list x = tasks.collect(t | t.start);
    list y = tasks.collect(t | t.resource);
    list w = tasks.collect(t | t.duration);
    list h = tasks.collect(t | 1);
    diff2(x, y, w, h);
  }
}
```

FIGURE 8.12 – Modélisation ARCAdE d'un problème d'ordonnancement sous contraintes de ressources.

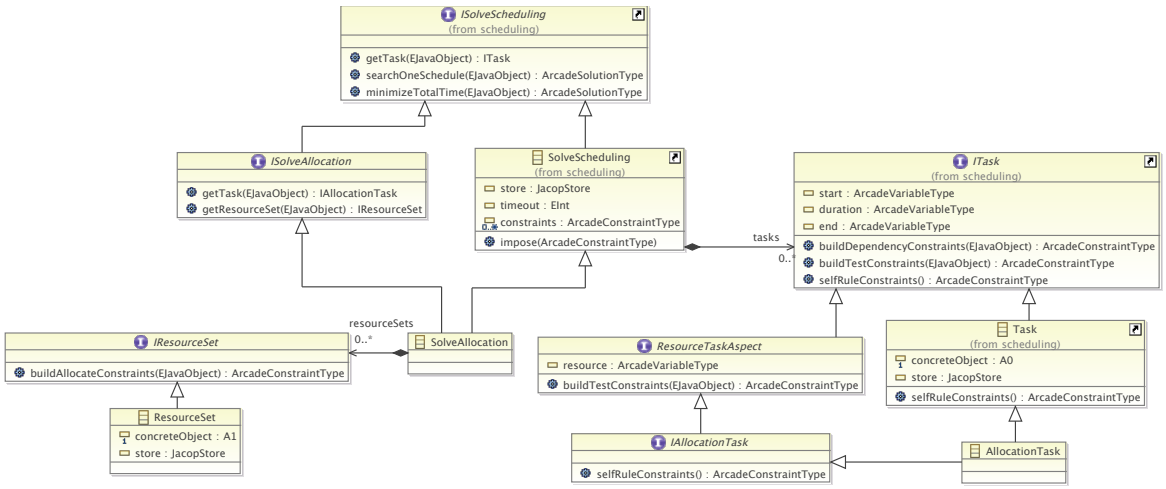


FIGURE 8.13 – Structure du code généré pour le problème d'ordonnancement sous contrainte de ressources.

8.4.2 Allocation de ressources

L'objectif est ici d'allouer une ressource pour chaque tâche à ordonnancer. De plus, une même ressource ne peut être allouée à deux tâches simultanément. Chaque tâche nécessite donc une nouvelle variable (*resource*) qui identifie la ressource utilisée.

La figure 8.12 décrit la modélisation du problème dans ARCAdE. Le problème d'ordonnancement précédent est importé, l'acteur tâche et toutes les stratégies qui y sont définies sont donc disponibles dans le problème courant. Un aspect (ligne 4) ajoute la variable *resource* à l'acteur d'une tâche, elle devient alors accessible pour toute référence à un acteur `Task`. Le partage de l'ensemble des ressources disponibles (acteur `ResourceSet`) est assuré par la règle d'allocation comportant une unique contrainte $Diff2$. Intuitivement, celle-ci garantit que des rectangles représentant l'exécution de chaque tâche sur une ressource ne s'entrelacent pas. L'abscisse du rectangle d'une tâche t_n est $start_n$, son ordonnée est $resource_n$, sa largeur est $duration_n$ et sa hauteur est unitaire.

La figure 8.13 illustre la structure du code généré. Une tâche dans ce problème hérite de la classe `Task` et implémente l'unique aspect (`ResourceTaskAspect`) déclaré dans le modèle. La notion d'aspect dans un modèle ARCAdE est donc mise en œuvre dans le code généré par un mécanisme d'héritage. Cet héritage est multiple si plusieurs aspects existent pour un même acteur. Or, Java ne supporte pas l'héritage multiple de différentes classes. Les interfaces permettent cependant de simuler ce comportement en annotant les méthodes du métamodèle du problème par leur comportement java. Lors de la génération finale du code par EMF, la classe d'un acteur comporte alors les prototypes et comportements (issus des annotations) de toutes les méthodes décrites dans les aspects visibles pour cet acteur. Dans le cas où deux aspects surchargent ou déclarent la même règle (i.e., noms et paramètres identiques), le comportement de la règle correspond à la concaténation des contraintes décrites dans chacun des aspects.

Un exemple de code utilisé pour déterminer l'ordonnancement et l'allocation d'un graphe est présenté dans la figure 8.14. Tout d'abord, le problème est instancié (ligne 1) en typant les classes concrètes : les nœuds (`Node`) sont typés en acteur $Task$ et l'ensemble des ressources disponibles

```
1  SolveAllocation<Node,MyResources> problem = AllocationFactory.eINSTANCE.
       createSolveAllocation();
2
3  for (Node n : graph.getNodes()) {
4    int d = n.getLatency();
5    IAllocationTask<Node> task = problem.getTask(n);
6    task.getDuration().setDomain(d,d); // Initialize task duration from the node information
7    task.getResource().setDomain(0,myResources.getSize()); // Number of resources is limited
8
9    for(Node predecessor: n.getPredecessors()){
10     problem.impose(task.buildDependencyConstraints(predecessor));
11   }
12 }
13 ResourceSet<MyResources> resources = problem.getResourceSet(myResources);
14 problem.impose(resources.buildAllocateConstraints(graph.getNodes()));
15
16 problem.minimizeTotalTime(graph.getNodes());
```

FIGURE 8.14 – Résolution du problème d'ordonnancement sous contraintes de ressource pour un graphe.

(`MyResources`) en acteur *ResourceSet*. Ensuite, l'étape d'initialisation des domaines des variables de chaque tâche tient compte de la durée d'un nœud et du nombre maximum de ressources disponibles (ligne 7). En plus de la règle issue de chaque dépendance de nœuds, l'appel de la règle d'allocation (ligne 14) garantit l'exclusivité temporelle et spatiale des tâches sur l'ensemble de ressources disponibles (`myResources`). Enfin, l'instance du problème est résolue en utilisant la même stratégie d'optimisation (ligne 16) que pour le problème d'ordonnancement simple.

8.4.3 Répartition d'une charge de travail

Dans cet exemple, on dispose d'un ensemble de travailleurs à répartir afin d'effectuer les différentes tâches à ordonnancer. L'exécution d'une tâche nécessite un certain nombre de travailleurs et chaque travailleur ne peut être affecté qu'à une seule tâche à la fois. Pour modéliser le problème, on définit un travail par tâche et pour chaque travailleur. Si la tâche est assignée au travailleur alors ce travail est considéré comme actif. Dans le cas contraire, il est inactif, le travailleur n'est pas affecté à la tâche.

Le modèle ARCAdE de la figure 8.15 est basé sur celui de l'ordonnancement simple, il ajoute notamment un aspect sur les tâches pour y associer un poids. Pour qu'une tâche soit exécutée, le nombre de travaux actifs qui lui sont liés doit être égal à son poids. La règle *dispatch* (ligne 7) exprime cette contrainte de répartition des charges pour une liste d'acteurs correspondant aux travaux possibles de la tâche.

Un travail (acteur $Work$) lie une tâche à un travailleur, elle est définie par trois variables *start*, *duration* et *active* qui déterminent respectivement le moment où un travailleur est affecté à une tâche, la durée du travail et si le travail est actif. Un travail étant lié à une tâche, sa date de début et sa durée sont égales à celles de la tâche (règle *scheduling*).

D'autre part, un travailleur (acteur $Worker$) ne peut travailler sur plus d'une tâche à la fois. La règle *working* garantit le respect de cette propriété en utilisant une contrainte cumulative. L'abscisse et la largeur de chaque rectangle sont définies respectivement par le début et la durée du travail

```
1  problem workers;
2  import "platform:/resource/org.arcade.examples/src/arcade/scheduling.arcade";
3
4  aspect Task : Weight {
5    int weight : "Number of workers required to complete this task";
6
7    rule dispatch(Work worksForTask*) {
8      list activeWorks = worksForTask.collect(w | w.active);
9      weight=sum(activeWorks);
10   }
11 }
12
13 actor Work {
14   var start : "Start time of this work";
15   var duration : "Duration of this work";
16   boolean active : "True if the work is activated";
17
18   rule scheduling(Task task){
19     duration = task.duration;
20     start = task.start;
21   }
22 }
23
24 actor Worker {
25   rule working(Work works*) {
26     list x = works.collect(e | e.start);
27     list w = works.collect(e | e.duration);
28     list h = works.collect(e | e.active);
29     cumulative(x,w,h,1);
30   }
31 }
```

FIGURE 8.15 – Ordonnancement avec répartition de charge de travail.

associé. De plus, la hauteur d'un rectangle correspond à la variable *active* et si celle-ci s'avère être nulle, le rectangle respectera toujours la limitation de la contrainte cumulative. Autrement dit, sur tous les rectangles correspondant aux travaux potentiels d'un travailleur, seul un pourra être actif simultanément puisque la limite de la contrainte cumulative (ligne 29) vaut un.

De même que pour le problème d'ordonnancement sous contrainte de ressources, la structure du code généré par ARCAdE pour un ordonnancement avec répartition des charges (cf. figure 8.16) n'introduit qu'un seul nouvel aspect sur les tâches. Cet aspect est mis en œuvre par l'interface $WeightTaskAspect$ contenant la variable associée au poids d'une tâche ainsi que la méthode correspondant à la règle qui garantit que le nombre de travailleurs affecté à la tâche est suffisant. Les nouveaux acteurs du problème ($Work$ et $Worker$) sont contenus dans la classe $SolveWorkers$ permettant de construire et résoudre une instance concrète du problème.

8.4.4 Gestion de projet

L'objectif est ici de réunir tous les problèmes évoqués précédemment pour énoncer un problème complexe de gestion de projet. Chaque tâche d'un projet dispose d'une durée et d'un poids indiquant le nombre de personnes nécessaires pour accomplir la tâche. Chaque tâche est effectuée dans une salle

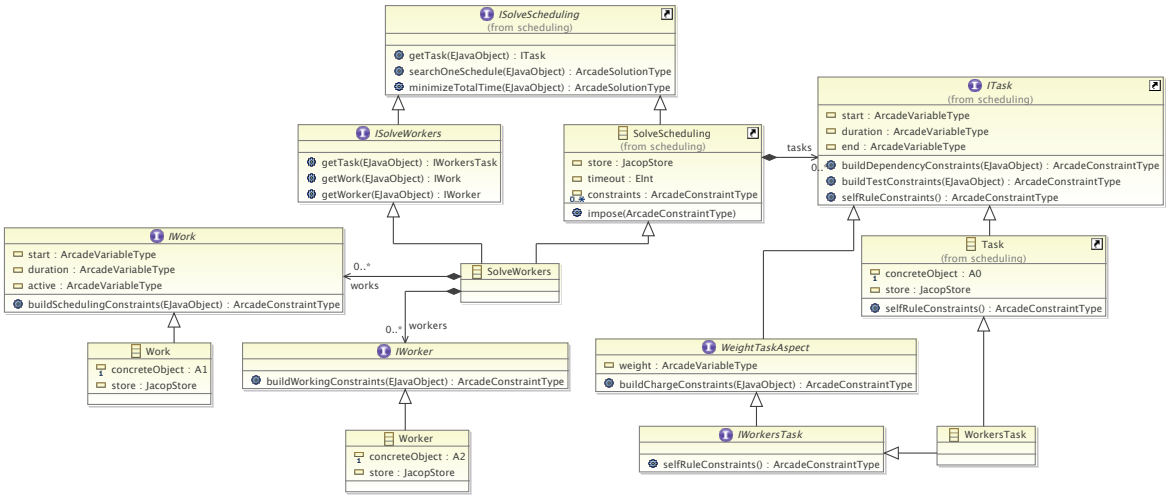


FIGURE 8.16 – Structure du code généré pour le problème d'ordonnancement avec répartition de charge de travail.

qui ne peut être partagée avec aucune autre tâche au même moment. L'effectif du projet étant limité, les personnes doivent être réparties sur les différentes tâches. Chaque personne ne peut s'occuper de plusieurs tâches simultanément.

Si ce problème est le plus complexe des exemples étudiés, il est aussi le plus simple à modéliser avec ARCAdE puisque tous les sous-problèmes qui le composent ont été modélisés précédemment de manière modulaire. Il suffit donc d'importer les sous-problèmes d'allocation et de répartition de charge pour composer automatiquement le problème de gestion de projet. Une tâche est le rôle commun de chacun des sous-problèmes et les aspects qui y sont ajoutés apportent toutes les variables et règles nécessaires à la modélisation du problème composite.

Afin d'illustrer la composition des aspects d'une tâche, la figure 8.17 présente la structure du code généré. Par souci de lisibilité, seules les tâches sont représentées. Les aspects $ResourceTaskAspect$ et $WeightTaskAspect$ ont été combinés en une unique interface $IPlanningTask$. La classe modélisant une tâche dans le problème est $PlanningTask$, elle implémente l'interface combinant les différents aspects et hérite de $Task$ pour y ajouter toutes les variables et règles issues des sous-problèmes importés.

8.5 Support d'un solveur existant

ARCAdE est un environnement basé sur une approche générative : 1) l'utilisateur modélise le problème 2) un générateur de code produit une mise en œuvre du problème qui est exploitable par un ou plusieurs solveurs existants. Pour simplifier le support d'un nouveau solveur, le flot de génération de code a pour objectif de minimiser l'effort de conception et d'intégration du générateur associé. Ainsi, un assistant génère à la fois l'infrastructure du nouveau générateur et le code permettant de l'intégrer à Eclipse (i.e., lancement du flot de génération à partir de l'éditeur). Le comportement de l'infrastructure de génération est ensuite à adapter aux spécificités de la cible.

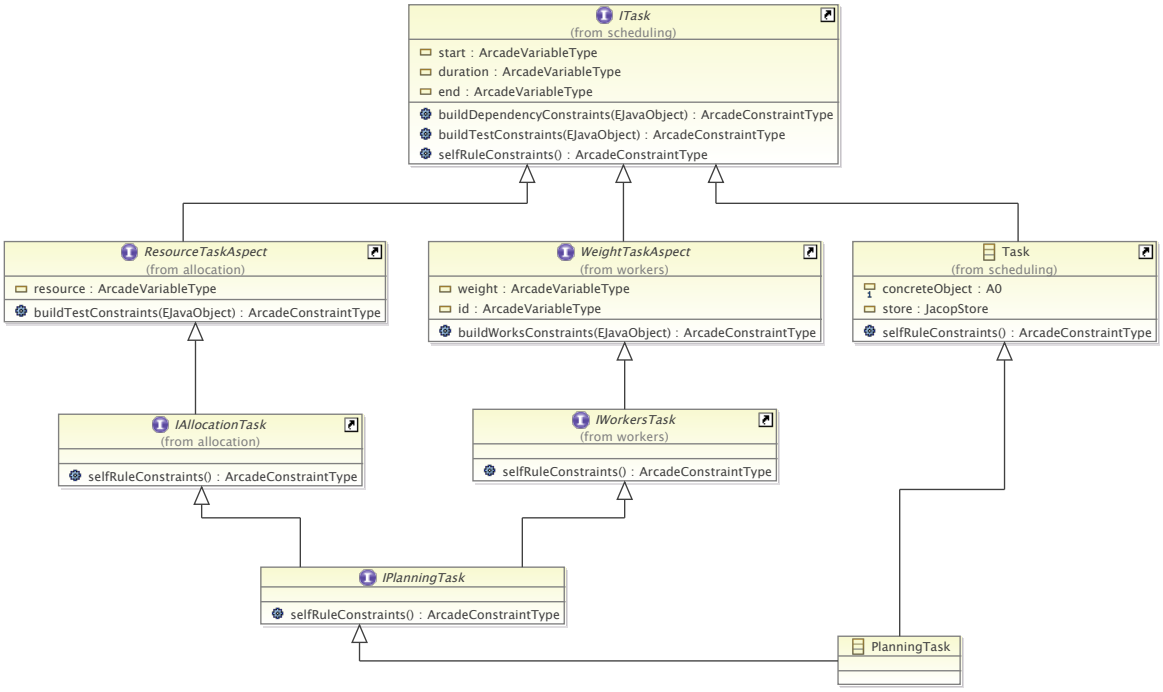


FIGURE 8.17 – Structure du code généré pour les tâches d'un problème de gestion de projet.

8.5.1 Flot reciblable de génération de code

L'indépendance vis-à-vis des solveurs est un élément critique d'un outil de modélisation de CSP. Pour la conserver, deux solutions sont envisageables.

- Disposer de générateurs de code spécifiques à chaque solveur. Le comportement du code généré pour un problème est spécifique : la déclaration des variables et des contraintes respecte la syntaxe du solveur cible. Le code est donc dépendant du solveur et peut être directement utilisé pour résoudre une instance du problème.
- Produire des problèmes exprimés dans une représentation intermédiaire commune dédiée à la description de CSP. Dans ce cas, le code généré pour un problème est unique. Les variables et les contraintes des acteurs et des stratégies sont conformes à une représentation intermédiaire pouvant ensuite être adaptée à de multiples solveurs. La résolution d'une instance de problème nécessite alors une étape supplémentaire (générative ou adaptative) permettant de construire une forme compréhensible par le solveur ciblé. Cependant, cette étape supplémentaire peut s'avérer coûteuse si l'instance du problème est de taille importante.

Dans les deux cas et quelle que soit la syntaxe ciblée, la structure du code générée par ARCAdE est identique. En effet, cette structure est toujours composée de classes modélisant les acteurs ainsi que d'une classe permettant d'énoncer et de résoudre une instance du problème (cf. section 8.4). Les spécificités des syntaxes ciblées ne portent que sur les types des éléments constituant un CSP (variables, domaines et contraintes) et sur les primitives permettant de les créer et de les initialiser.

Le flot de génération de code d'ARCAdE utilise Xtend2, un langage dédié à la génération de texte (cf. paragraphe 7.3.2), pour générer un métamodèle conforme à la cible (solveur ou représentation

```
1  class JacopEcoreGenerator extends ArcadeEcoreGenerator{
2
3    override customize(org.arcade.System s){
4      '''« addCustomType(JacopConstants::STORE_TYPE_NAME,JacopConstants::STORE_TYPE) »'''
5    }
6
7    def dispatch customize(Actor e){
8      '''« customAttribute("store","//"+JacopConstants::STORE_TYPE_NAME,false) »'''
9
10   }
11 }
```

FIGURE 8.18 – Personnalisation d'un générateur structurel pour JaCoP. Un attribut de type `JaCoP.core.Store` est ajouté à la classe de chaque acteur.

intermédiaire PPC). Afin de simplifier l'ajout de nouvelles cibles, les générateurs sont répartis en deux familles : générateurs de la *structure* du code et ceux du *comportement*.

Générateurs structurels Les générateurs structurels produisent la description du métamodèle d'un problème (au format Ecore d'EMF). Dans ce métamodèle, le type de toutes les variables et celui des contraintes sont dépendants de la cible, ils sont représentés par des types abstraits associés explicitement dans le métamodèle à leur concrétisation. Par exemple, les type concrets d'une variable et d'une contrainte dans JaCoP sont respectivement `JaCoP.core.Variable` et `JaCoP.constraints.Constraint`.

Un premier générateur structurel est chargé d'indiquer explicitement ces correspondances. Il doit donc être impérativement spécialisé lors de l'ajout d'une nouvelle cible. Les autres générateurs structurels produisent notamment les classes des acteurs en analysant les différents aspects à composer. Ils n'ont, a priori, pas besoin d'être modifiés pour cibler de nouvelles syntaxes ou représentations intermédiaires. Cependant, il est parfois nécessaire d'ajouter des informations aux différentes classes générées. Ainsi, dans le cas de JaCoP, les classes relatives aux acteurs et au problème ont besoin d'une référence sur le conteneur JaCoP des variables et des contraintes.

Pour répondre aux spécificités de chaque cible, il est possible de personnaliser simplement la structure des classes générées. Il suffit de spécialiser une fonction qui est appelée lors de la génération de la structure de chaque classe. La figure 8.18 illustre la personnalisation mise en œuvre pour JaCoP. Le type correspondant au conteneur JaCoP est ajouté dans le métamodèle du problème (ligne 4). Il est utilisé par les attributs *store* (ligne 8) générés pour chaque acteur.

Générateurs du comportement Les comportements des classes générées sont ajoutés sous forme d'annotations dans le métamodèle produit par les générateurs structurels. Le générateur de code Java d'EMF utilise ensuite ces annotations pour produire le corps de chaque méthode. La génération de ces comportements est répartie en quatre générateurs à spécialiser en fonction de la cible :

- générateur du code des contraintes spécifiques.
- générateur du code des variables et des domaines.
- générateur des méthodes de recherche et d'optimisation.
- générateur du code des contraintes arithmétiques.

Les générateurs de comportements constituent le cœur de l'interface entre ARCAdE et le solveur ou la représentation intermédiaire ciblée. Ils s'avèrent, en pratique, relativement simples à décrire grâce aux facilités offertes par Xtend2 (e.g., *multiple dispatch*). À titre d'exemple, les générateurs des variables et des domaines pour JaCoP [177] et Choco [101] sont décrits par une cinquantaine de lignes de code Xtend2. Le générateur des contraintes globales spécifiques est évidemment le plus long à décrire du fait du nombre de contraintes à supporter. La génération du code Java des contraintes arithmétiques peut également s'avérer fastidieuse selon le solveur ciblé. Ainsi, la syntaxe de JaCoP ne permet pas d'exprimer des contraintes arithmétiques arborescentes. Il est alors nécessaire de décomposer chaque contrainte arithmétique complexe en plusieurs sous-contraintes et variables temporaires. Cependant, ce processus est considérablement simplifié par un algorithme de décomposition généré à partir d'une description des contraintes arithmétiques supportées par le solveur (cf. 8.5.2).

D'autre part, les générateurs du comportement peuvent tirer parti des annotations pour exprimer des comportements spécifiques au solveur. Par exemple, l'annotation suivante

```
1 @selector->"MostConstrainedStatic"
2 minimize(cost);
```

guide JaCoP dans la recherche d'une solution en indiquant explicitement que l'ordre d'évaluation des variables est déterminé par le nombre de contraintes associées. Le code généré pour une minimisation d'une fonction de coût est donc modifié en fonction des annotations présentes dans le modèle du problème.

8.5.2 Décomposition des contraintes arithmétiques

Si un solveur ne dispose pas de contraintes arithmétiques arborescentes, l'algorithme BURG [62] permet néanmoins d'automatiser le processus de décomposition. L'objectif est ici de réduire le nombre total de contraintes primitives (c.-à-d., supportées par le solveur ciblé) nécessaires pour exprimer une contrainte arithmétique complexe. BURG est souvent utilisé en compilation pour produire le code conforme à un jeu d'instructions à partir d'une représentation intermédiaire arborescente. Il offre, en un temps polynomial, une solution optimale à la minimisation de la somme des coûts statiques des instructions sélectionnées.

L'algorithme s'appuie sur la notion de motifs arborescents et la sélection de leurs occurrences dans l'arbre constitue la réponse au problème. Ces motifs sont définis par des règles pondérées d'un coût statique. Le principe de l'algorithme est de parcourir l'arbre de manière ascendante et d'associer à chaque nœud, une règle qui minimise le coût du nœud et dont le motif est compatible avec l'arborescence fille. Le coût d'un nœud dépend de celui de ces fils et de la règle sélectionnée.

La figure 8.19 expose les coûts possibles de chacun des nœuds de la contrainte $a+b+c > d$ dans le cas où deux règles peuvent être appliquées à $n2$. La première de ces règles ne contient qu'une addition, elle est associée à un coût déterminé par une fonction $COST(k)$ qui privilégie les règles ayant le plus d'opérandes. Ainsi, le coût d'une règle à deux opérandes sera plus élevé qu'une règle en ayant trois. Le coût total du nœud en sélectionnant cette règle est donc $c^1_{n2} = c_{n4} + c_{n5} + COST(2)$. Une autre règle (double addition) est compatible avec le nœud $n2$, son occurrence inclut également $n4$ et dispose de trois opérandes qui sont toutes des terminaux de l'arbre. Le coût associé est $c^2_{n2} = c_{n7} + c_{n8} + c_{n5} + COST(3)$, on en déduit que $c^2_{n2} < c^1_{n2}$ et que la deuxième règle de $n2$ est donc sélectionnée. L'algorithme s'achève lorsque le nœud racine ($n1$) est atteint. Dès lors, un second parcours de l'arbre

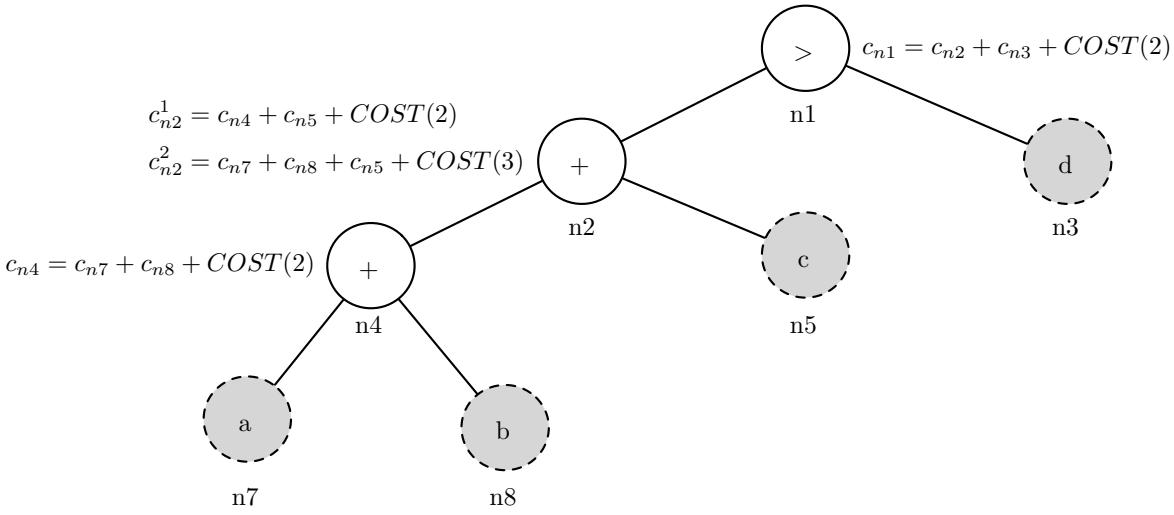


FIGURE 8.19 – Couverture de la contrainte $a+b+c > d$ avec l'algorithme BURG pour les contraintes natives de JaCoP.

```
1  [x,y] ::x=y  {"new JaCoP.constraints.XeqY(var:x,var:y)"}
2  [x,y] ::x>y  {"new JaCoP.constraints.XgtY(var:x,var:y)"}
3  [x,y,z] ::x+y=z {"new JaCoP.constraints.XplusYeqZ(var:x,var:y,var:z)"}
4  [x,y,q,z] ::x+y+q=z {"new JaCoP.constraints.XplusYplusQeqZ(var:x,var:y,var:q,var:z)"}
```

FIGURE 8.20 – Extrait de la description des règles arithmétiques supportées par JaCoP .

est effectué de manière descendante. En partant du nœud racine, on effectue une action associée à la règle sélectionnée. Cette action consiste tout d'abord à exécuter les actions des nœuds correspondant à ses opérandes, puis de produire le code de la règle.

L'algorithme BURG est généré à partir d'une description des contraintes arithmétiques supportées nativement par le solveur cible. Celles-ci constituent les règles applicables aux nœuds des arbres issus de chaque contrainte à décomposer. Un DSL simplifie la description des règles. Chaque règle y est définie par une contrainte arithmétique ARCAdE et est associée explicitement au code généré correspondant. La description des règles de JaCoP permettant de couvrir l'arbre de la figure 8.19 est présenté dans la figure 8.20. Chaque règle contient une liste de variables disponibles, une contrainte arithmétique et le code Java correspondant. Les variables du code de la règle sont nommées selon une convention particulière (i.e., la variable x est identifiée par `var:x` dans le code) afin de les remplacer lors de la génération par celles de la contrainte à décomposer.

Des contraintes arithmétiques de formes différentes ont parfois la même sémantique. Par exemple, les contraintes $a + b * c = d$ et $b * c + a = d$ sont équivalentes alors que leurs structures arborescentes sont différentes. Pour un solveur disposant d'une règle $a+b*c = d$, l'algorithme ne détectera donc pas la règle si la contrainte est dans une autre forme. La description de toutes les formes possibles d'une règle n'est évidemment pas une solution raisonnable. Cependant, elles peuvent être obtenues automatiquement en analysant la commutativité et l'associativité des opérateurs arithmétiques impliqués. Ainsi, pour chaque règle définie, les règles équivalentes sémantiquement sont générées et ajoutées automatiquement à celles supportées par l'algorithme de sélection.

8.6 Conclusion

ARCAdE constitue un nouvel environnement qui simplifie considérablement la modélisation et la capitalisation de problèmes de programmation par contraintes. Le langage dédié permet de décrire et de composer très simplement des sous-problèmes en problèmes complexes en se basant sur la notion d'aspects et d'héritage multiple.

Le générateur de code d'ARCAdE est extensible de manière à cibler n'importe quel solveur Java. À ce jour, deux générateurs de code sont disponibles : pour le solveur JaCoP et pour l'interface spécifiée par la JSR331 (notamment compatible avec les solveur JaCoP et Choco).

ARCAdE a notamment été utilisé pour modéliser les différentes techniques de couverture de graphe et d'extension de jeu d'instructions présentées dans le chapitre 3. La modélisation ARCAdE de ces problèmes est détaillée dans l'annexe A.

Conclusion

Les processeurs extensibles constituent une solution attractive pour la conception de processeurs spécialisés. Ils sont à la fois capables d'exécuter n'importe quelle application sur le GPP hôte et d'améliorer l'efficacité du traitement de zones critiques du code en les déportant sur une extension matérielle contrôlée par des instructions spécialisées. Concevoir un processeur extensible reste cependant un problème complexe du fait des multiples caractéristiques architecturales à envisager par le concepteur en amont du flot de compilation. Ce constat implique de placer le compilateur au centre d'un processus d'exploration semi-automatique qui guidera le concepteur vers un compromis qu'il juge adapté aux exigences du produit final.

Cette étape de compilation diffère des techniques habituelles de sélection de code et d'ordonnancement puisque le jeu d'instructions et l'architecture du processeur sont dépendants de la cible applicative. Ainsi, le jeu d'instructions est une sortie du processus de compilation qui sélectionnera et ordonnancera celles qui amélioreront le plus l'efficacité du processeur pour exécuter l'application. Dans cette thèse, nous nous sommes intéressé à l'automatisation des tâches de sélection et d'ordonnancement d'instructions spécialisées ainsi qu'à leurs mises en œuvre dans une infrastructure de compilation.

Contributions

Techniques de sélection et d'ordonnancement d'instructions spécialisées basées sur la couverture de graphes

L'exploitation du parallélisme dans un processeur extensible nécessite de sélectionner les ISE en tenant compte de leur ordonnancement. En effet, dans le cas contraire, la sélection favorisera des ISE qui semblent pertinentes (e.g., celles dont la taille est plus importante) mais dont l'ordonnancement n'exploitera pas le parallélisme potentiel de l'architecture.

Nous proposons une nouvelle technique de couverture de graphe basée sur la programmation par contraintes pour modéliser dans un unique problème, la sélection et l'ordonnancement d'occurrences de motifs. Le modèle de contraintes est conçu pour traiter des ordonnancements parallèles et permet de couvrir et d'ordonnancer des graphes de plusieurs centaines de nœuds en quelques secondes.

Cette technique de couverture de graphe a le mérite de pouvoir être aisément spécialisée en ajoutant de nouvelles contraintes au problème. Ainsi, ce problème a été étendu à la problématique d'extension de jeu d'instructions pour deux architectures. Les résultats expérimentaux obtenus montrent qu'il est possible d'accélérer généralement d'un facteur 2 l'exécution d'applications sur une architecture où l'extension matérielle ne peut exécuter qu'un seul motif à la fois (taille inférieure à 10 nœuds). Le second modèle d'architecture s'appuie sur une exécution VLIW pour exécuter, en parallèle, le comportement de plusieurs motifs. Les accélérations obtenues peuvent être supérieures à un facteur 10 pour des applications exposant un niveau suffisant de parallélisme et si les données externes au graphe analysé sont lues et écrites dans des mémoires de l'extension. Le fait de ne plus utiliser cette mémoire interne diminue significativement l'accélération (e.g., une accélération 24x est alors réduite à 5x) et montre l'intérêt d'analyser la provenance des données d'un bloc de base afin d'éviter l'utilisation du

processeur pour charger des données produites, à l'origine (e.g., lors d'une itération précédente du corps de boucle), sur l'extension.

Transformations de boucles guidées par la sélection d'instructions spécialisées

Les interactions entre les optimisations d'un compilateur et la passe de sélection d'ISE est un sujet qui n'a que très peu abordé. Pourtant, il est évident que la qualité des regroupements d'opérations est fortement dépendante des transformations qui auront été effectuées en amont.

Dans cette thèse, nous nous sommes intéressé à la transformation de nids de boucles de manière à faire apparaître des instructions spécialisées qui n'auraient pas pu être identifiées par une approche n'analysant que les blocs de base d'une application. L'approche mêle les ordonnancements affines structurés de Feautrier au problème de couverture de graphe précédent et permet à la fois de sélectionner des ISE vectorisables ainsi que de transformer les nids de boucles pour rendre leur sélection légale. Dans ce cas, les occurrences de motifs représentent des zones de calcul (appelées macroblocs) intégralement déportées sur l'extension matérielle et peuvent contenir plusieurs ISE qui communiquent entre elles, à des itérations différentes, via des mémoires embarquées sur l'extension.

L'ingénierie dirigée par les modèles au cœur d'une infrastructure de compilation

La mise en œuvre d'une étape de compilation pour un processeur extensible implique de lourdes tâches de développement qui peuvent être considérablement simplifiées par l'IDM. Dans cette optique, cette thèse a largement contribué à l'intégration de l'IDM au sein de l'infrastructure de compilation GeCoS.

Ainsi, nous avons développé de multiples outils transversaux (e.g, GraphMapper, TomMapper) aux problématiques rencontrées dans un compilateur. Ces outils nous ont permis de réduire considérablement le temps nécessaire à mettre en œuvre les multiples tâches de transformation, d'optimisation et de génération de code.

D'autre part, la compilation s'appuie par essence sur des abstractions d'un langage (i.e., représentations intermédiaires) et se trouve donc être très proche des thématiques abordées par l'IDM : une bonne représentation intermédiaire est celle qui est la plus adaptée à résoudre un problème spécifique. Intégrer les méthodologies de conception de l'IDM au sein d'un compilateur se fait donc naturellement et offre une valeur ajoutée significative en termes de qualité du code par une mise en œuvre systématique de bonnes pratiques de programmation. De plus, cette intégration permet de s'interfacer, à moindre coût, avec de nombreux outils existants qui facilitent notamment la mise en œuvre de transformations et de vérifications des instances de représentations intermédiaires.

Environnement de modélisation modulaire de problèmes d'optimisation

La programmation par contraintes est utilisée par toutes nos techniques d'extension de jeu d'instructions. De manière à faciliter la conception et l'exploration de différents modèles de contraintes, nous avons conçu un environnement de modélisation modulaire qui repose sur des concepts avancés de programmation-objet (e.g., héritage multiple, aspects).

L'environnement permet de modéliser et de résoudre de manière « naturelle » des problèmes complexes dans un langage dédié : 1) identification des différents acteurs d'un problème 2) description de leurs interactions sous forme de contraintes 3) définition des stratégies possibles de résolution.

La modularité est un des points particulièrement intéressants de l'environnement : pour composer un problème complexe, il suffit d'importer des sous-problèmes qui seront alors automatiquement combinés.

Le langage proposé est indépendant du solveur utilisé et le générateur de code peut être étendu simplement pour cibler n'importe quel solveur Java.

Perspectives

La compilation optimisante pour les processeurs extensibles est un axe de recherche particulièrement riche. Les défis à relever sont nombreux et au cours de cette thèse, nous avons proposé à la fois de nouvelles techniques d'optimisation et les moyens logiciels de les mettre en œuvre efficacement. Les perspectives, à plus ou moins long terme, de ces travaux sont donc nombreuses. Nous nous contentons ici d'en résumer celles qui sont liées directement à la compilation ou aux processeurs extensibles.

Analyse et amélioration de l'espace de recherche pour la couverture de graphe

La résolution des problèmes de couverture de graphe proposés dans cette thèse s'appuie sur les méthodes standard de la programmation par contraintes pour déterminer une solution. Comme évoqué plusieurs fois dans les chapitres précédents, l'ordre d'évaluation des variables ainsi que le choix de leurs valeurs ont une influence considérable sur l'efficacité de la recherche.

Nous pensons qu'une analyse précise de l'arbre de recherche permettrait de dégager les comportements qui compliquent ou facilitent la recherche. Une fois ces comportements extraits, il sera possible de guider la recherche en ajoutant des contraintes redondantes ou encore en concevant des ordres dynamiques d'évaluation (i.e., dépendant de l'état courant des variables lors de la résolution) et qui seront spécifiques à notre problème de couverture et d'ordonnancement sous contraintes de ressources.

Exploitation de l'espace conjoint de transformation de boucles et de sélection d'ISE

La technique de transformation de code et de sélection d'instructions offre une expressivité fertile à de futures recherches. En effet, avec ce formalisme il est possible d'ajouter simplement des contraintes supplémentaires qui porteront sur les coefficients d'ordonnancement de chaque occurrence de motif.

Pour l'instant, nous nous sommes contenté d'assurer la possibilité de vectoriser les instructions spécialisées sélectionnées mais on pourrait envisager, par exemple, de contraindre la distance des dépendances (i.e., différence entre la date d'ordonnancement de l'itération consommant une donnée et celle la produisant) à ne pas dépasser un certain seuil afin de favoriser la localité des données sur l'extension.

Limitation de la taille des mémoires embarquées sur l'extension

Comme évoqué dans la perspective précédente, il est possible d'ajouter des contraintes supplémentaires à l'ordonnancement de chaque occurrence de motif candidate. Parmi ces contraintes, la gestion d'une taille limite pour les communications entre deux ISE apparaît comme prioritaire. En effet, la sélection d'un motif dans notre approche, implique l'utilisation d'une mémoire pour temporiser les données produites et ce, quelle que soit sa taille requise. Ainsi, dans le cas du triple produit matriciel

étudié dans le chapitre 3, la taille de la mémoire nécessaire à la temporisation des résultats est égale à la taille des matrices. Ceci n'est évidemment pas acceptable pour des matrices dont la taille est trop importante.

Malheureusement, limiter la taille de ces mémoires est loin d'être évident. Il s'agit d'un problème ouvert qui, à notre connaissance, n'a pas encore de solution dans le cas multidimensionnel. Il existe des techniques de contraction mémoire [5] mais celles-ci ne peuvent être appliquées que si l'ordonnancement est connu.

Utilisation de contraintes non linéaires pour des problèmes d'ordonnancement affines

Enfin, une perspective à nos travaux provient de la jonction effectuée entre le modèle polyédrique et la programmation par contrainte. En effet, il serait intéressant d'évaluer l'intérêt et le passage à l'échelle de contraintes globales ou tout simplement non linéaires ajoutées à des problèmes d'ordonnancement affine habituellement résolus avec un solveur ILP. Par exemple, si on impose une contrainte $AllDifferent$ sur les coefficients d'ordonnancement d'une dimension scalaire (i.e., cas où seul le coefficient de la partie constante du prototype d'ordonnancement n'est pas nul), chacune des instructions du PRDG sera alors distribuée dans un nid de boucle différent.

Annexes

Annexe A

Modélisation ARCAdE de la couverture de graphe

Cette annexe contient les descriptions ARCAdE des différents problèmes de couverture et d'ordonnancement présentés dans le chapitre 3.

A.1 Sélection des occurrences de motifs

```
problem covering;
actor Node {
  var match : "Identifier of covering match";
  var relativeMatchID : "Position of covering match in candidates";

  rule covered(Match matches*) {
    list matchesRelativesIds = matches.collect(m | m.id);
    match = matchesRelativesIds[relativeMatchID];
  }
}
actor Match {
  int id : "Identifier of a match";
  boolean selected : "True if match is selected";
  rule cover(Node nodes*) {
    list matches = nodes.collect(n | n.match);
    let var nbCoveredNodes ::[0, nodes.size];
    nbCoveredNodes = card(matches, id);
    selected <=> nbCoveredNodes != 0;
  }
}
strategy searchOneCovering() {
  search ONE;
}
strategy minimizeNumberOfSelectedMatches(Match matches*) {
  let var cost;
  list selections = matches.collect(e | e.selected);
  cost = sum(selections);
  minimize(cost);
}
```

A.2 Allocation de ressources pour une couverture de graphe

```
problem acovering;
import
"platform:/resource/fr.irisa.cairn.graph.patterns.covering/src/arcade/covering.arcade";

aspect Match : Resource {
  var rm : "Resource identifier of a match";
}

aspect Node : Resource {
  var rn : "Resource identifier of a node";

  rule covered(Match matches*) {
    list resources = matches.collect(m | m.rm);
    rn = resources[relativeMatchID];
  }
}

strategy minimizeNumberOfResources(Node nodes*){
  list resources = nodes.collect(n | n.rn);
  let var nbResources;
  nbResources = card(resources);
  minimize(nbResources);
}
```

A.3 Sélection et ordonnancement d'occurrences de motifs (sans contraintes de ressources)

```
problem scovering;
import
"platform:/resource/fr.irisa.cairn.graph.patterns.covering/src/arcade/covering.arcade";
import
"platform:/resource/org.arcade.toolbox/src/arcade/scheduling.arcade";

aspect Node : Scheduled extends Task{
  rule dependency(Node source) {
    start >= source.start;
    if(match != source.match) {
      start >= source.end;
    }
  }
  rule staticDependency(Node source) {
    start >= source.end;
  }
  rule covered(Match matches*) {
    list starts = matches.collect(m | m.tm);
    list delays = matches.collect(m | m.dm);
    start = starts[relativeMatchID];
    duration = delays[relativeMatchID];
  }
}

aspect Match : Scheduled {
  var tm : "Start time of a match";
  var dm : "Duration of a match";
}

strategy searchBestSchedule(Node nodes*) {
  let var cost;
  list ends = nodes.collect(n | n.end);
  cost = max(ends);
  minimize(cost);
}

strategy minimizeSequentialSchedule(Match matches*){
  let var cost;
  list durations = matches.collect(m | m.dm*m.selected);
  cost = sum(durations);
  minimize(cost);
}
```

A.4 Sélection et ordonnancement d'instructions spécialisées

A.4.1 Processeur extensible

```
problem asip;
import
"platform:/resource/fr.irisa.cairn.graph.patterns.covering/src/arcade/scovering.arcade";
import
"platform:/resource/fr.irisa.cairn.graph.patterns.covering/src/arcade/mcovering.arcade";

aspect Node : Output {
  boolean isOutput : "True if node has at least one data consumed by the core";

  rule isOutputNode(Match oneNodeOutputs*) {
    list selected = oneNodeOutputs.collect(e | e.selected);
    let var nbOutputs ::[0 .. oneNodeOutputs.size];
    nbOutputs = sum(selected);
    isOutput <=> nbOutputs > 0;
  }
}


aspect Match : Execution {
  var executionTime : "Duration of match execution on the extension";
}

aspect Match : MemoryAccess {
  NB_IN_PER_CYCLE = 2;
  NB_OUT_PER_CYCLE = 1;
  var nbReadDatas : "Number of datas coming from the core";
  var nbWriteDatas : "Number of datas written to the core";
  boolean isMoreThanOneCycle : "True if match has a more than one cycle duration";
  boolean isWriting : "True if match write at least one data to the core";
  boolean isReading :"True if match read at least one data from the core";
  int staticInputs : "Minimal number of datas coming from the core";
  var ERT : "Extra cycles used to read datas";
  var EWT : "Extra cycles used to write datas";
  var SWT : "Start of write back task";
  var WD : "Write duration";
  var RD : "Read duration";

  where {
    isWriting <=> nbWriteDatas > 0;
    WD = EWT + isWriting * isMoreThanOneCycle;
    SWT >= tm + ERT + executionTime - isMoreThanOneCycle;
    SWT = tm + dm - WD;
```

```
    dm >= ERT + executionTime + EWT;
    RD = ERT + isReading;
    isReading <=> nbReadDatas > 0;
  }
  rule inputs(Match oneNodeInputs*) {
    let var reads ::[Z];
    list oneNodeInputsSelected = staticInputs U oneNodeInputs.collect(e | e.selected);
    let var cin;
    cin < $NB_IN_PER_CYCLE;
    nbReadDatas = sum(oneNodeInputsSelected);
    nbReadDatas =(ERT + isReading) * $NB_IN_PER_CYCLE - cin;
  }

  rule outputs(Node nodes*) {
    list outputs = nodes.collect(n | n.isOutput);
    let var cout;
    cout < $NB_OUT_PER_CYCLE;
    nbWriteDatas = sum(outputs);
    nbWriteDatas =(EWT + isWriting) * $NB_OUT_PER_CYCLE - cout;
  }
}
```

A.4.2 Extension séquentielle

```
problem sequential;
import "platform:/resource/fr.irisa.cairn.asip/src/arcade/asip.arcade";

enum ResourceType {
  PROCESSOR, EXT
}

actor Processor {
  rule sharing(Match matches*, Match oneNodesMatches*) {
    list x1 = matches.collect(m | m.tm) //read
           U oneNodesMatches.collect(m | m.tm) //processor nodes
           U matches.collect(m | m.SWT); //write

    list y1 = matches.collect(m | 0) //read
         U oneNodesMatches.collect(m | 0) //processor nodes
         U matches.collect(m | 0);

    list w1 = matches.collect(m | m.ERT + 1) //read
           U oneNodesMatches.collect(m | m.dm) //processor nodes
           U matches.collect(m | m.WD); //write

    list h1 = matches.collect(m | m.selected) //read
           U oneNodesMatches.collect(m | m.selected) //processor nodes
```

```
          U matches.collect(m | m.selected); //write

    diff2(x1, y1, w1, h1);
    matches.each(m | m.rm = ResourceType::EXT);
    oneNodesMatches.each(m | m.rm = ResourceType::PROCESSOR);
  }
}
```

A.4.3 Extension parallèle

```
problem sequential;
import "platform:/resource/fr.irisa.cairn.asip/src/arcade/asip.arcade";

enum Data {
  SEND,RECEIVE
}

actor Extension {
  int nbOfCells : "Number of available reconfigurable cells on the extension";

  rule shareCells(Match matches*){
    list x = matches.collect(m | m.tm);
    list y = matches.collect(m | m.rm);
    list w = matches.collect(m | m.dm);
    list h = matches.collect(m | m.selected);

    diff2(x, y, w, h);
    matches.each(m | m.rm < nbOfCells);
  }
}
actor Processor {
  int Dmax: "Maximal duration of a scheduing";
  /*
  * The processor can't launch anything on the extension when
  * computing an instruction
  */
  rule execution(Extension extension,Match ematches*,Match oneNodeMatches*){
    list x =  ematches.collect(m | m.tm + m.ERT)  //Launch ISE
        U  oneNodeMatches.collect(m | m.tm);  //Launch processor instruction
    list y =  ematches.collect(m | m.rm)
        U oneNodeMatches.collect(m | 0);
    list w =  ematches.collect(m | m.selected)
        U oneNodeMatches.collect(m | m.selected);
    list h = ematches.collect(m | 1)
        U oneNodeMatches.collect(m | extension.nbOfCells);
    diff2(x,y,w,h);
  }
```

```
38
39   /*
40   * The extensible processor can't send or
41   * receive any data when computing an instruction
42   */
43   rule datas(Match ematches*,Match oneNodeMatches*){
44     list x =  ematches.collect(m | m.tm )  //Send data
45         U  ematches.collect(m | m.SWT )  //Receive data
46         U  oneNodeMatches.collect(m | m.tm);  //Launch processor instruction
47
48     list y =  ematches.collect(m | Data::SEND)
49         U ematches.collect(m | Data::RECEIVE)
50         U oneNodeMatches.collect(m | 0);
51     list w =  ematches.collect(m | m.RD)
52         U ematches.collect(m | m.WD)
53         U oneNodeMatches.collect(m | 2);
54     list h = ematches.collect(m | m.selected)
55         U  ematches.collect(m | m.selected)
56         U oneNodeMatches.collect(m | m.selected);
57     diff2(x,y,w,h);
58   }
59
60   strategy minimizeNumberOfResourcesUnderTimingConstraints(Node nodes*){
61     let var nbResources;
62     let var end;
63     list resources = nodes.collect(n | n.rn);
64     list ends = nodes.collect(n | n.end);
65     end = max(ends);
66     end < Dmax;
67     nbResources = card(resources);
68     minimize(nbResources);
69   }
70 }
```

Liste des Abréviations

ADL Architecture Description Language

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

ASIP Application Specific Instruction Set Processor

AST Abstract Syntax Tree

BB Branch and Bound

CDFG Control Data Flow Graph

CP Constraint Programming

CSP Constraint Satisfaction Problem

DAG Directed Acyclic Graph

DFG Dataflow Graph

DFT Dataflow Tree

DMA Direct Memory Access

DSL Domain Specific Language

DSP Digital Signal Processor

FPGA Field Programmable Gate Array

GPP General Purpose Processor

HCDG Hierarchical Conditional Dependency Graph

IDM Ingénirie Dirigée par les modèles

ILP Integer Linear Programming

IP Intellectual Property

IR Intermediate Representation

ISA Instruction Set Architecture

ISE Instruction Set Extension

M2M Model to Model

M2T Model to Text

MAC Multiply Accumulate

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MISO Multiple Inputs Single Output

PPC Programmation par constraintes

RISC Reduced Instruction Set Processor

RTL Register Transfer Level

SCoP Static Control Part

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SoC System on Chip

T2M Text to Model

UAL Unitée Arithmétique et Logique

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

WCET Worst Case Execution Time

Liste des publications

Journaux internationaux

[1] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. *ACM transactions on Reconfigurable Technology and Systems (TRETS)*, 2012. à paraitre.

[2] E. Raffin, C. Wolinski, F. Charot, E. Casseau, A. Floch, K. Kuchcinski, S. Chevobbe et S. Guyetant : Scheduling, Binding and Routing System for a Run-Time Reconfigurable Operator Based Multimedia Architecture. *Journal of Embedded and Real-Time Communication Systems*, 3(1):1–30, jan. 2012.

Conférences internationales

[1] A. Floch, C. Wolinski et K. Kuchcinski : Combined Scheduling and Instruction Selection for Processors with Reconfigurable Cell Fabric. *In 21th IEEE International Conference on Application-specific Systems, Architectures and Processors, (ASAP 2010)*, Rennes, France, juil. 2010. IEEE.

[2] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye et R. France : Model-Driven Engineering and Optimizing Compilers : A bridge too far ? *In International Conference on Model Driven Engineering Languages and Systems*, Wellington, Nouvelle-Zélande, oct. 2011.

[3] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint-Driven Instructions Selection and Application Scheduling in the DURASE system. *In 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, (ASAP 2009)*, p. 145–152, Boston, États-Unis.

[4] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint-Driven Identification of Application Specific Instructions in the DURASE System. *In 9th International Workshop on Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS 2009)*, vol. 5657 de *Lecture Notes in Computer Science*, p. 194–203, Samos, Grèce, 2009. Springer Berlin / Heidelberg.

Conférences nationales

[1] A. Floch, F. Charot, S. Derrien, K. Martin, A. Morvan et C. Wolinski : Sélection d'instructions et ordonnancement parallèle simultanés pour la conception de processeurs spécialisés. *In Symposium en Architecture de Machines (Sympa'14)*, St Malo, France, mai 2011.

[2] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Sélection automatique d'instructions et ordonnancement d'applications basés sur la programmation par contraintes. *In 13ème Symposium en Architecture de machines (SympA'13)*, Toulouse, France, 2009.

Workshops

[1] C. Wolinski, K. Kuchcinski, K. Martin, A. Floch, E. Raffin et F. Charot : Graph Constraints in Embedded System Design. *In Worshop on Combinatorial Optimization for Embedded System Design (COESD 2010)*, Bologne, Italie, juin 2010.

Bibliographie

[1] Target - http ://www.retarget.com/.

[2] T. Achterberg : SCIP : solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, juil. 2009.

[3] A. V. Aho, R. Sethi et J. D. Ullman : *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] A. Aletà, J. M. Codina, A. González et D. Kaeli : Removing communications in clustered microarchitectures through instruction replication. *ACM Trans. Archit. Code Optim.*, 1:127–151, June 2004.

[5] C. Alias, F. Baray et A. Darte : Bee+cl@k : an implementation of lattice-based array contraction in the source-to-source translator rose. *SIGPLAN Not.*, 42(7):73–82, juin 2007.

[6] C. Alippi, W. Fornaciari, L. Pozzi et M. Sami : A DAG-based design approach for reconfigurable VLIW processors. *In DATE '99 : Proceedings of the conference on Design, automation and test in Europe*, p. 57, New York, NY, USA, 1999. ACM Press.

[7] Altera : NiosII custom instruction user guide.

[8] R. Andonov, S. Balev, S. Rajopadhye et N. Yanev : Optimal semi-oblique tiling. *In SPAA '01 : Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, p. 153–162, New York, NY, USA, 2001. ACM.

[9] R. Andonov, P.-Y. Calland, S. Niar, S. Rajopadhye et N. Yanev : First steps towards optimal oblique tile sizing. *In 8th International Workshop on Compilers for Parallel Computers*, p. 351–366, 2000.

[10] K. R. Apt et M. Wallace : *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[11] G. Araujo, S. Malik et M. T.-C. Lee : Using register-transfer paths in code generation for heterogeneous memory-register architectures. *In Proceedings of the 33rd annual Design Automation Conference*, DAC '96, p. 591–596, New York, NY, USA, 1996. ACM.

[12] K. Atasu, G. Dündar et C. Özturan : An integer linear programming approach for identifying instruction-set extensions. *In CODES+ISSS '05 : Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, p. 172–177, New York, NY, USA, 2005. ACM.

[13] K. Atasu, W. Luk, O. Mencer, C. Ozturan et G. Dundar : Fish : Fast instruction synthesis for custom processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(1):52 –65, jan. 2012.

[14] K. Atasu, L. Pozzi et P. Ienne : Automatic application-specific instruction-set extensions under microarchitectural constraints. *In DAC '03 : Proceedings of the 40th conference on Design automation*, p. 256–261, New York, NY, USA, 2003. ACM Press.

[15] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro et J. Oliver : Nanoscompiler : A research platform for OpenMP extensions. *In In First European Workshop on OpenMP*, p. 27–31, 1999.

[16] D. Barthou, J.-F. Collard et P. Feautrier : Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput.*, 40(2):210–226, 1997.

[17] A. Barvinok : A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *In Foundations of Computer Science, 1993. 34th Annual Symposium*, p. 566 –572, nov 1993.

[18] C. Bastoul : Code generation in the polyhedral model is easier than you think. *In PACT '04 : Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, p. 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[19] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton et D. Wonnacott : OMPVerify : polyhedral analysis for the OpenMP programmer. *In Proceedings of the 7th international conference on OpenMP in the Petascale era*, IWOMP'11, p. 37–53, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen et C. Bastoul : The polyhedral model is more widely applicable than you think. *In Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, Paphos, Cyprus, mars 2010. Springer-Verlag.

[21] R. V. Bennett, A. C. Murray, B. Franke et N. Topham : Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. *In LCTES '07 : Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, p. 83–92, New York, NY, USA, 2007. ACM.

[22] P. Biswas, S. Banerjee, N. Dutt, P. Ienne et L. Pozzi : Performance and energy benefits of instruction set extensions in an FPGA soft core. *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, p. 6 pp.–, Jan. 2006.

[23] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi et P. Ienne : ISEGEN : Generation of high-quality instruction set extensions by iterative improvement. *In DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, p. 1246–1251, Washington, DC, USA, 2005. IEEE Computer Society.

[24] B.M.Smith, S.C.Brailsford, P.M.Hubbard et H.P.Williams : The progressive party problem : Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1995.

[25] U. Bondhugula, A. Hartono, J. Ramanujam et P. Sadayappan : A practical automatic polyhedral parallelizer and locality optimizer. *In PLDI '08 : Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, p. 101–113, New York, NY, USA, 2008. ACM.

[26] U. K. Bondhugula : *Effective Automatic Parallelization and Locality Optimization using the Polyhedral Model*. Thèse de doctorat, The Ohio State University, 2008.

[27] P. Bonzini et L. Pozzi : Code transformation strategies for extensible embedded processors. *In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, p. 242–252, New York, NY, USA, 2006. ACM.

[28] P. Bonzini et L. Pozzi : Polynomial-time subgraph enumeration for automated instruction set extension. *In DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, p. 1331–1336, San Jose, CA, USA, 2007. EDA Consortium.

[29] P. Bonzini et L. Pozzi : Recurrence-aware instruction set selection for extensible embedded processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(10):1259–1267, 2008.

[30] P. Boulet, A. Darte, T. Risset et Y. Robert : (pen)-ultimate tiling ? *Integr. VLSI J.*, 17(1):33–51, 1994.

[31] J. M. Cardoso et P. C. Diniz : *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 2008.

[32] P. Caspi, D. Pilaud, N. Halbwachs et J. A. Plaice : Lustre : a declarative language for programming synchronous systems. *In 14th ACM Conf. on Principles of Programming Languages*, Munich, jan 1987.

[33] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr et G. Braun : C compiler retargeting based on instruction semantics models. *In Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, p. 1150–1155, Washington, DC, USA, 2005. IEEE Computer Society.

[34] B. Chamberlain, D. Callahan et H. Zima : Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

[35] J. Chame et S. Moon : A tile selection algorithm for data locality and cache interference. *In ICS '99 : Proceedings of the 13th international conference on Supercomputing*, p. 492–499, New York, NY, USA, 1999. ACM.

[36] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun et V. Sarkar : X10 : an object-oriented approach to non-uniform cluster computing. *In ACM SIGPLAN Notices*, vol. 40, p. 519–538. ACM, 2005.

[37] C. Chen, J. Chame et M. Hall : Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. *In CGO '05 : Proceedings of the international symposium on Code generation and optimization*, p. 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[38] X. Chen, D. L. Maskell et Y. Sun : Fast identification of custom instructions for extensible processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):359–368, 2007.

[39] R. Chenouard, L. Granvilliers et R. Soto : Model-driven constraint programming. *In Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '08, p. 236–246, New York, NY, USA, 2008. ACM.

[40] H. Choi, J. H. Yi, J.-Y. Lee, I.-C. Park et C.-M. Kyung : Exploiting intellectual properties in ASIP designs for embedded dsp software. *In Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, p. 939–944, New York, NY, USA, 1999. ACM.

[41] N. Clark, H. Zhong et S. Mahlke : Automated custom instruction generation for domain-specific processor acceleration, 2005.

[42] M. Clavreul, O. Barais et J.-M. Jézéquel : Integrating legacy systems with MDE. *In ICSE'10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering and ICSE Workshops*, vol. 2, p. 69–78, Cape Town, South Africa, May 2010.

[43] S. Coleman et K. S. McKinley : Tile size selection using cache organization and data layout. *In PLDI '95 : Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, p. 279–290, New York, NY, USA, 1995. ACM.

[44] J. Cong, Y. Fan, G. Han et Z. Zhang : Application-specific instruction generation for configurable processor architectures. *In FPGA '04 : Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, p. 183–189, New York, NY, USA, 2004. ACM Press.

[45] L. P. Cordella, P. Foggia, C. Sansone et M. Vento : A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[46] O. Coudert : On solving binate covering problems. *In Proceedings of the Design Automation Conference*, p. 197–202, 1996.

[47] CoWare : Lisatek datasheet - http ://www.coware.com.

[48] K. Darby-Dowman et J. Little : Properties of some combinatorial optimization problemsand their effect on the performance of integer programming and constraint logic programming. *INFORMS J. on Computing*, 10:276–286, March 1998.

[49] R. Dechter : Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41:273–312, 1990.

[50] S. Demassey : *Méthodes hybrides de programmation par contraintes et programmation linéaire pour le problème d'ordonnancement de projet à contraintes de ressources.* Thèse de doctorat, Université d'Avignon, 2003.

[51] Q. Dinh, D. Chen et M. D. F. Wong : Efficient ASIP design for configurable processors with fine-grained resource sharing. *In FPGA '08 : Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, p. 99–106, New York, NY, USA, 2008. ACM.

[52] K. Esseghir : Improving data locality for caches. Mémoire de D.E.A., Dept. of Computer Science, Rice University, 1993.

[53] P. Feautrier : Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[54] P. Feautrier : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[55] P. Feautrier : Some efficient solutions to the affine scheduling problem : I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[56] P. Feautrier : Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, 1992.

[57] P. Feautrier : Scalable and structured scheduling. *Int. J. Parallel Program.*, 34(5):459–487, 2006.

[58] F. Focacci, A. Lodi et M. Milano : Cutting planes in constraint programming : A hybrid approach. *In Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, p. 187–201, London, UK, 2000. Springer-Verlag.

[59] B. B. Fraguela, M. G. Carmueja, D. Andrade, G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, E. Zapata, B. B. F. A, M. G. C. A et D. A. A : Optimal tile size selection guided by analytical models. *In In PARCO*, p. 565–572, 2005.

[60] R. France et B. Rumpe : Model-driven development of complex software : A research roadmap. *In* L. Briand et A. Wolf, éds : *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[61] B. Franke, M. O'Boyle, J. Thomson et G. Fursin : Probabilistic source-level optimisation of embedded programs. *In Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '05, p. 78–86, New York, NY, USA, 2005. ACM.

[62] C. W. Fraser, R. R. Henry et T. A. Proebsting : Burg : fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[63] C. Galuzzi et K. Bertels : The instruction-set extension problem : A survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4:18 :1–18 :28, mai 2011.

[64] C. Galuzzi, K. Bertels et S. Vassiliadis : A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions. *In ARC*, p. 130–141, 2007.

[65] C. Galuzzi, K. Bertels et S. Vassiliadis : The spiral search : A linear complexity algorithm for the generation of convex MIMO instruction-set extensions. *In Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, p. 337–340, Dec. 2007.

[66] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels et S. Vassiliadis : Automatic selection of application-specific instruction-set extensions. *In CODES+ISSS '06 : Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, p. 160–165, New York, NY, USA, 2006. ACM.

[67] A. Gamatié : *Designing Embedded Systems with the SIGNAL Programming Language - Synchronous, Reactive Specification.* Springer, 2010.

[68] J. G. Gaschnig : *Performance measurement and analysis of certain search algorithms.* Thèse de doctorat, Pittsburgh, PA, USA, 1979. AAI7925014.

[69] M. L. Ginsberg : Dynamic backtracking. *J. Artif. Int. Res.*, 1:25–46, August 1993.

[70] S. GIRBAL : *Optimisation d'applications - Composition de transformations de programme : modèle et outils.* Thèse de doctorat, Université de Paris XI Orsay, 2005.

[71] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler et O. Temam : Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[72] C. P. Gomes et D. Schmoys : The promise of LP to boost CSP techniques for combinatorial problems. *In* N. Jussien et F. Laburthe, éds : *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, p. 291–305, Le Croisic, France, mars, 25–27 2002.

[73] D. Goodwin et D. Petkov : Automatic generation of application specific processors. *In CASES '03 : Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, p. 137–147, New York, NY, USA, 2003. ACM Press.

[74] G. Goossens, D. Lanneer, W. Geurts et J. Van Praet : Design of asips in multi-processor socs using the chess/checkers retargetable tool suite. *In System-on-Chip, 2006. International Symposium on*, p. 1 –4, nov. 2006.

[75] G. Goumas, M. Athanasaki et N. Koziris : An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14:1034, 2003.

[76] G. Goumas, N. Drosinos et N. Koziris : Communication-aware supernode shape. *IEEE Trans. Parallel Distrib. Syst.*, 20(4):498–511, 2009.

[77] M. Gries et K. Keutzer : *Building ASIPs : The Mescal Methodology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[78] A. Grösslinger : *The Challenges Of Non-linear Parameters And Variables In Automatic Loop Parallelisation*. Thèse de doctorat, Universität Passau, 2009.

[79] S. Guelton : *Building Source-to-Source compilers for Heterogenous targets*. Thèse de doctorat, Télécom Bretagne, 2011.

[80] C. Guettier : *Optimisation globale du placement d'applications de traitement du signal sur architectures parallèles en utilisant la programmation logique avec contraintes*. Thèse de doctorat, Ecoles des mines de Paris, 1997.

[81] Y. Guo, G. J. Smit, H. Broersma et P. M. Heysters : A graph covering algorithm for a coarse grain reconfigurable system. *In LCTES '03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, p. 199–208, New York, NY, USA, 2003. ACM.

[82] G. Gupta et S. Rajopadhye : The z-polyhedral model. *In PPoPP '07 : Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, p. 237–248, New York, NY, USA, 2007. ACM.

[83] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge et R. B. Brown : Mibench : A free, commercially representative embedded benchmark suite. *In WWC '01 : Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, p. 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[84] R. Hartenstein : A decade of reconfigurable computing : a visionary retrospective. *In DATE '01 : Proceedings of the conference on Design, automation and test in Europe*, p. 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[85] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam et P. Sadayappan : Parametric multi-level tiling of imperfectly nested loops. *In ICS '09 : Proceedings of the 23rd international conference on Supercomputing*, p. 147–157, New York, NY, USA, 2009. ACM.

[86] J. Hauser et J. Wawrzynek : Garp : a MIPS processor with a reconfigurable coprocessor. *In FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, p. 12 –21, apr 1997.

[87] E. Hodzic et W. Shang : On time optimal supernode shape. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1220–1233, 2002.

[88] A. Hoffmann, H. Meyr et R. Leupers : *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[89] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen et H. Meyr : A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. *In Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, p. 625–630, Piscataway, NJ, USA, 2001. IEEE Press.

[90] K. Högstedt, L. Carter et J. Ferrante : On the parallel execution time of tiled loops. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):307–321, 2003.

[91] C.-h. Hsu et U. Kremer : A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.

[92] P. Ienne et R. Leupers : *Customizable Embedded Processors : Design Technologies and Applications (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[93] ILOG : Ilog solver. http ://www-01.ibm.com/software/websphere/ilog/.

[94] F. Irigoin, P. Jouvelot et R. Triolet : Semantical interprocedural parallelization : an overview of the pips project. *In Proceedings of the 5th international conference on Supercomputing*, ICS '91, p. 244–251, New York, NY, USA, 1991. ACM.

[95] F. Irigoin et R. Triolet : Supernode partitioning. *In POPL '88 : Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 319–329, New York, NY, USA, 1988. ACM.

[96] ISL : Integer set library. http ://freshmeat.net/projects/isl/.

[97] ITRS : System drivers - http ://www.itrs.net/links/2011itrs/2011chapters/2011sysdrivers.pdf.

[98] J. Jaffar, S. Michaylov, P. J. Stuckey et R. H. C. Yap : The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14:339–395, May 1992.

[99] M. Jain, M. Balakrishnan et A. Kumar : ASIP design methodologies : survey and issues. *In VLSI Design, 2001. Fourteenth International Conference on*, p. 76 –81, 2001.

[100] B. Jayaraman et P. Tambay : Modeling engineering structures with constrained objects. *In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, p. 28–46, London, UK, UK, 2002. Springer-Verlag.

[101] N. Jussien, C. Prud'homme, H. Cambazard, G. Rochar et F. Laburthe : Choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[102] R. Kastner, A. Kaplan, S. O. Memik et E. Bozorgzadeh : Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):605–627, 2002.

[103] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh et M. Sarrafzadeh : Instruction generation for hybrid reconfigurable systems. *In ICCAD*, p. 127, 2001.

[104] N. Kavvadias et S. Nikolaidis : A flexible instruction generation framework for extending embedded processors. *In Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*, p. 125 –128, 2006.

[105] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman et D. Wonnacott : The omega library. Rap. tech., University of Maryland, Nov. 1996.

[106] B. W. Kernighan et S. Lin : An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[107] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin : Aspect-oriented programming. *In* M. Akşit et S. Matsuoka, éds : *ECOOP'97 — Object-Oriented Programming*, vol. 1241 de *Lecture Notes in Computer Science*, chap. 10, p. 220–242. Springer-Verlag, Berlin/Heidelberg, 1997.

[108] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye et M. M. Strout : Multi-level tiling : M for the price of one. *In SC '07 : Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, p. 1–12, New York, NY, USA, 2007. ACM.

[109] A. A. Kountouris et C. Wolinski : Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):380–412, 2002.

[110] C. Kozyrakis et D. Patterson : Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36 – 45, nov.-dec. 2003.

[111] K. Kuchcinski : Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, 2003.

[112] I. Kuon et J. Rose : Measuring the gap between fpgas and asics. *In FPGA '06 : Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, p. 21–30, New York, NY, USA, 2006. ACM Press.

[113] C. Lee, M. Potkonjak et W. H. Mangione-Smith : Mediabench : A tool for evaluating and synthesizing multimedia and communicatons systems. *In International Symposium on Microarchitecture*, p. 330–335, 1997.

[114] J.-E. Lee, K. Choi et N. D. Dutt : Instruction set synthesis with efficient instruction encoding for configurable processors. *ACM Trans. Des. Autom. Electron. Syst.*, 12:9 :1–9 :37, February 2007.

[115] S. I. Lee, T. A. Johnson et R. Eigenmann : Cetus - an extensible compiler infrastructure for source-to-source transformation. *In* L. Rauchwerger, éd. : *LCPC*, vol. 2958 de *Lecture Notes in Computer Science*, p. 539–553. Springer, 2003.

[116] X. Leroy : Formal certification of a compiler back-end or : programming a compiler with a proof assistant. *In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 42–54. ACM, 2006.

[117] R. Leupers, K. Karuri, S. Kraemer et M. Pandey : A design flow for configurable embedded processors based on optimized instruction set extension synthesis. *In DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, p. 581–586, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[118] R. Leupers et P. Marwedel : Retargetable generation of code selectors from hdl processor models. *In Proceedings of the 1997 European conference on Design and Test*, EDTC '97, p. 140–, Washington, DC, USA, 1997. IEEE Computer Society.

[119] R. L. Leupers et S. Bashford : Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5:794–814, October 2000.

[120] H. LeVerge : A note on chernikova's algorithm. Rap. tech. no 635, INRIA, 1992.

[121] L. L'Hours : Generating Efficient Custom FPGA Soft-Cores for Control-Dominated Applications. *In Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, vol. 0, p. 127–133, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[122] T. Li, W. Jigang, Y. Deng, T. Srikanthan et X. Lu : Accelerating identification of custom instructions for extensible processors. *Circuits, Devices Systems, IET*, 5(1):21 –32, january 2011.

[123] S. Liao, S. Devadas, K. Keutzer et S. Tjiang : Instruction selection using binate covering for code size optimization. *In Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, p. 393 –399, nov 1995.

[124] Y.-S. Lu, L. Shen, L.-B. Huang, Z.-Y. Wang et N. Xiao : Optimal subgraph covering for customisable VLIW processors. *IET Computers and Digital Techniques*, 3(1):14–23, 2009.

[125] K. Martin : *Génération automatique d'extensions de jeux d'instructions de processeurs*. Thèse de doctorat, Université de Rennes 1, 2010.

[126] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint-driven identification of application specific instructions in the durase system. *In SAMOS '09 : Proceedings of the 9th International Workshop on Embedded Computer Systems : Architectures, Modeling, and Simulation*, p. 194–203, Berlin, Heidelberg, 2009. Springer-Verlag.

[127] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint-driven instructions selection and application scheduling in the durase system. *In ASAP 2009- 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009.

[128] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch et F. Charot : Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. *ACM transactions on Reconfigurable Technology and Systems (TRETS)*, jan. 2012.

[129] C. Mauras : *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. Thèse de doctorat, Université de Rennes 1, IFSIC, décembre 1989.

[130] MCrypt : http ://sourceforge.net/projects/mcrypt/.

[131] B. Mei, S. Vernalde, D. Verkest, H. De Man et R. Lauwereins : Adres : An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *In* P. Y. K. Cheung et G. Constantinides, éds : *Field Programmable Logic and Application*, vol. 2778 de *Lecture Notes in Computer Science*, p. 61–70. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45234-8_7.

[132] M. Milano et P. V. Hentenryck, éds. *Hybrid Optimization : The Ten Years of CPAIOR.* Springer, 2011.

[133] U. Montanari : Networks of constraints : Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[134] A. Morvan, S. Derrien et P. Quinton : Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion. *In Field-Programmable Technology (FPT), 2011 International Conference on*, p. 1 –10, dec. 2011.

[135] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, J., Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramírez, D. Rodenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunović et A. Zaks : ACOTES Project : Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, 38, 2010.

[136] F. Munoz, B. Baudry, R. Delamare et Y. Le Traon : Inquiring the usage of aspect-oriented programming : an empirical study. *In 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, Sep-Oct 2009.

[137] A. Murray et B. Franke : Compiling for automatically generated instruction set extensions. *In International Symposium on Code Generation and Optimization (CGO '12)*, San Jose, CA, USA, April 2012.

[138] N. Museux : *Aide au placement d'applications de traitement du signal sur machines parallèles MULTI-SPMD.* Thèse de doctorat, Ecoles des mines de Paris, 2001.

[139] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck et G. Tack : Minizinc : Towards a standard CP modelling language. *In* C. Bessiere, éd. : *CP*, vol. 4741 de *Lecture Notes in Computer Science*, p. 529–543. Springer, 2007.

[140] S. P. K. Nookala et T. Risset : A library for z-polyhedral operations. Rap. tech. 1330, IRISA, 2000.

[141] NXP : http ://www.nxp.com/.

[142] M. Pasha, S. Derrien et O. Sentieys : System level synthesis for ultra low-power wireless sensor nodes. *In Digital System Design : Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, p. 493 –500, 2010.

[143] PLUTO : http ://pluto-compiler.sourceforge.net/.

[144] PolarSSL : Small cryptographic library, http ://polarssl.org/.

[145] PolyLib : A library of polyhedral functions - http ://icps.u-strasbg.fr/polylib/.

[146] N. Pothineni, A. Kumar et K. Paul : Application specific datapath extension with distributed i/o functional units. *In VLSID '07 : Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, p. 551–558, Washington, DC, USA, 2007. IEEE Computer Society.

[147] L.-N. Pouchet, C. Bastoul, A. Cohen et J. Cavazos : Iterative optimization in the polyhedral model : Part II, multidimensional time. *In ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI'08), p. 90–100, Tucson, Arizona, June 2008. ACM Press.

[148] L.-N. Pouchet, C. Bastoul, A. Cohen et N. Vasilache : Iterative optimization in the polyhedral model : Part I, one-dimensional time. *In IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*, p. 144–156, San Jose, California, March 2007. IEEE Computer Society press.

[149] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan et N. Vasilache : Loop transformations : convexity, pruning and optimization. *In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, p. 549–562, New York, NY, USA, 2011. ACM.

[150] L. Pozzi et P. Ienne : Exploiting pipelining to relax register-file port constraints of instruction-set extensions. *In CASES '05 : Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, p. 2–10, New York, NY, USA, 2005. ACM.

[151] PPL : The parma polyhedra library - http ://www.cs.unipr.it/ppl/.

[152] P. Prosser : Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[153] W. Pugh : The omega test : a fast and practical integer programming algorithm for dependence analysis. *In Supercomputing '91 : Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, p. 4–13, New York, NY, USA, 1991. ACM.

[154] F. Quilleré, S. Rajopadhye et D. Wilde : Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, 2000.

[155] D. J. Quinlan : Rose : Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[156] P. Quinton : The systematic design of systolic arrays. *In Centre National de Recherche Scientifique on Automata networks in computer science : theory and applications*, p. 229–260, Princeton, NJ, USA, 1987. Princeton University Press.

[157] P. Quinton et T. Risset : Structured scheduling of recurrence equations : Theory and practice. *In Embedded Processor Design Challenges : Systems, Architectures, Modeling, and Simulation - SAMOS*, p. 112–134, London, UK, UK, 2002. Springer-Verlag.

[158] J. Ramanujam : Non-unimodular transformations of nested loops. *In Supercomputing '92 : Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, p. 214–223, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[159] J. Ramanujam et P. Sadayappan : Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108 – 120, 1992.

[160] J. Reddington, G. Gutin, A. Johnstone, E. Scott et A. Yeo : Better than optimal : Fast identification of custom instruction candidates. *In Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 2, p. 17 –24, aug. 2009.

[161] L. Renganarayana et S. Rajopadhye : Positivity, posynomials and tile size selection. *In SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[162] L. Renganarayanan, D. Kim, S. Rajopadhye et M. M. Strout : Parameterized tiled loops for free. *In PLDI '07 : Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, p. 405–414, New York, NY, USA, 2007. ACM.

[163] G. Rivera et C.-W. Tseng : A comparison of compiler tiling algorithms. *In CC '99 : Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, p. 168–182, London, UK, 1999. Springer-Verlag.

[164] F. Rossi, P. v. Beek et T. Walsh : *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[165] K. Rounioja et K. Puusaari : Implementation of an hsdpa receiver with a customized vector processor. *In System-on-Chip, 2006. International Symposium on*, p. 1 –4, nov. 2006.

[166] D. Schmidt : Guest editor's introduction : Model-driven engineering. *Computer*, 39(2):25 – 31, feb. 2006.

[167] A. Schrijver : *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[168] C. Schulte, M. Z. Lagerkvis et G. Tack : Gecode. http ://www.gecode.org/.

[169] SoCLib : Open platform for virtual prototyping of multi-processors system on chip (mp-SOC), http ://www.soclib.fr.

[170] N. Sonmez et A. Yurdakul : Sixd : A configurable application-specific sisd/SIMD microprocessor soft-core. p. 1–4, Nov. 2006.

[171] S. Sorlin et C. Solnon : A global constraint for graph isomorphism problems. *In Proceedings First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, Nice, France, April 20-22, 2004.

[172] R. M. Stallman et G. J. Sussman : Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[173] J. Steel et J.-M. Jézéquel : On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, 2007.

[174] Stretch : Stretch instruction set architecture reference - http ://www.stretchinc.com/.

[175] L. Su : Digital media, the new frontier for supercomputing. Opening Keynote, MPSoC, 2005.

[176] Synopsys : Arc700 family overview - http ://www.synopsys.com/.

[177] R. Szymanek et K. Kuchcinski : Jacop. http ://jacop.osolpro.com/.

[178] Tenselica : Xtensa instruction set architecture - http ://www.tensilica.com/.

[179] The Object Management Group : UML 2.0 : Superstructure Specification. Version 2.0, OMG, formal/05-07-04, 2005.

[180] W. Thies, F. Vivien, J. Sheldon et S. Amarasinghe : A unified framework for schedule and storage optimization. *In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, p. 232–242, New York, NY, USA, 2001. ACM.

[181] A. Tisserand : Étude et conception d'opérateurs arithmétiques. Habilitation à Diriger les Recherches de l'Université de Rennes 1, 2010.

[182] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks et I. Rosen : Polyhedral-model guided loop-nest auto-vectorization. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:327–337, 2009.

[183] J. R. Ullmann : An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[184] K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman et M. Weiss : Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Appl. Signal Process.*, 2005:2613–2625, jan. 2005.

[185] P. Van Hentenryck : *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[186] N. Vasilache : *Scalable Program Optimization Techniques in the Polyhedral Model*. Thèse de doctorat, Université de Paris-Sud 11, 2007.

[187] S. Vassiliadis et D. Soudris : *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 1st édn, 2007.

[188] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner et M. Bruynooghe : Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, juin 2007. URL : http ://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl ?id=41970, DOI : 10.1007/s00453-006-1231-0.

[189] A. K. Verma, P. Brisk et P. Ienne : Rethinking custom ise identification : a new processor-agnostic method. *In CASES '07 : Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, p. 125–134, New York, NY, USA, 2007. ACM.

[190] F. Vivien : On the optimality of Feautrier's scheduling algorithm. *Concurrency and Computation Practice and Experience*, 15(11-12):1047–1068, 2003.

[191] O. Wahlen, M. Hohenauer, R. Leupers et H. Meyr : Instruction scheduler generation for retargetable compilation. *Design Test of Computers, IEEE*, 20(1):34 – 41, jan-feb 2003.

[192] R. Wittig et P. Chow : Onechip : an FPGA processor with reconfigurable logic. *In FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, p. 126 –135, apr 1996.

[193] M. E. Wolf : *Improving locality and parallelism in nested loops*. Thèse de doctorat, Stanford, CA, USA, 1992.

[194] M. J. Wolfe : *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[195] C. Wolinski et K. Kuchcinski : Identification of application specific instructions based on subgraph isomorphism constraints. *In Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, p. 328–333, juil. 2007.

[196] C. Wolinski et K. Kuchcinski : Automatic selection of application-specific reconfigurable processor extensions. *DATE*, 2008.

[197] I.-W. Wu, S.-C. Huang, C.-P. Chung et J.-J. Shann : Instruction set extension generation with considering physical constraints. *In Proceedings of the 2nd international conference on High performance embedded architectures and compilers*, HiPEAC'07, p. 291–305, Berlin, Heidelberg, 2007. Springer-Verlag.

[198] C. Xiao et E. Casseau : An efficient algorithm for custom instruction enumeration. *In Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, GLSVLSI '11, p. 187–192, New York, NY, USA, 2011. ACM.

[199] Xilinx : Microblaze processor reference guide - http ://www.xilinx.com.

[200] J. Xue : Transformations of nested loops with non-convex iteration spaces. *Parallel Comput.*, 22(3):339–368, 1996.

[201] J. Xue : Communication-minimal tiling of uniform dependence loops. *J. Parallel Distrib. Comput.*, 42(1):42–59, 1997.

[202] J. Xue : *Loop tiling for parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[203] Z. Ye, A. Moshovos, S. Hauck et P. Banerjee : Chimaera : a high-performance architecture with a tightly-coupled reconfigurable functional unit. *In Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, p. 225 –235, june 2000.

[204] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella *et al.* : Titanium : A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.

[205] J. M. Youn, J. Lee, Y. Paek, J. Lee, H. Scharwaechter et R. Leupers : Fast graph-based instruction selection for multi-output instructions. *Softw. Pract. Exper.*, 41(6):717–736, mai 2011.

[206] P. Yu et T. Mitra : Satisfying real-time constraints with custom instructions. *In Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, p. 166–171, New York, NY, USA, 2005. ACM.

[207] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger et K. O'Brien : Automatic creation of tile size selection models. *In CGO '10 : Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, p. 190–199, New York, NY, USA, 2010. ACM.

[208] S. Zampelli : *A Constraint Programming Approach to Subgraph Isomorphism.* Thèse de doctorat, Ecole polytechnique de Louvain, 2008.

[209] H. Zhu et R. Alkins : Towards role-based programming. *In Workshop on Role-Based Collaboration, CSCW*, 2006.