Déploiement d'applications multimédia sur architecture reconfigurable à gros grain : modélisation avec la programmation par contraintes

Erwan Raffin

N° d'ordre : 4341 ANNÉE 2011

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

École doctorale Matisse

présentée par

Erwan RAFFIN

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

Déploiement d'applications multimédia sur architecture reconfigurable à gros grain : modélisation avec la programmation par contraintes

Thèse soutenue à Rennes
le 13 juillet 2011

devant le jury composé de :

Jean-François NEZAN
Professeur à l'INSA Rennes / examinateur

Smail NIAR
Professeur à l'Univ. de Valenciennes / rapporteur

Tanguy RISSET
Professeur à l'INSA Lyon / rapporteur

Henk HEIJNEN
Manager à Technicolor Rennes / examinateur

Christophe WOLINSKI
Professeur à l'Univ. de Rennes 1
directeur de thèse

François CHAROT
Chargé de recherche à l'INRIA Rennes
co-directeur de thèse

A celles et ceux qui m'ont soutenu et cru en moi...

Pour Élise.

Remerciements

Tout d'abord, je tiens à remercier Jean-François Nezan, Professeur à l'INSA de Rennes, qui me fait l'honneur de présider ce jury.

Je remercie, Samil Niar, Professeur à l'Université de Valenciennes, d'avoir bien voulu accepter la charge de rapporteur ainsi que Tanguy Risset, Professeur à l'INSA de Lyon, d'avoir bien voulu juger ce travail.

Je remercie tout particulièrement Christophe Wolinski, Professeur à l'Université de Rennes 1, et François Charot, Chargé de recherche à l'INRIA de Rennes qui ont dirigé ma thèse à l'IRISA, ainsi que Henk Heijnen et Emmanuel Jolly pour leur encadrement à Technicolor Rennes. Tous ont contribué à l'aboutissement de cette thèse.

Je remercie chaleureusement toutes les personnes qui m'ont entouré durant toute la période de la thèse et auprès de qui j'ai beaucoup appris : l'équipe de recherche CAIRN à l'IRISA en particulier Kevin Martin, Antoine Floc'h, George Adouko, Adeel Pasha, Steven Derrien, Antoine Morvan, Laurent Perraudeau, Naeem Abbas, Emmanuel Casseau, Daniel Chillet, Daniel Ménard, Sébastien Pillement, Arnaud Tisserand, Cécile Beaumin, Antoine Eiche, Nadia Saintpierre, et notre chef d'équipe Olivier Sentieys, mais aussi toutes les personnes de Technicolor et de Thomson Silicon Components, ainsi que les personnes impliquées dans les projets Ter@ops, SoClib et ROMA.

Evidemment je tiens à remercier toutes les personnes qui m'ont encouragé, soutenu et surtout celles et ceux qui ont cru en moi.

A toutes ces personnes, MERCI.

Table des matières

Introduction

Contexte des travaux

Les applications multimédia sont de plus en plus présentes dans les systèmes électroniques embarqués grand public : télévisions numériques, téléphones portables de type « *Smart Phone* », baladeurs audio-vidéo, « *box* » des opérateurs de télécommunication, consoles de jeux vidéo et bien d'autres encore. Autant ce fait est bien connu et appréhendé par la majorité des consommateurs de ces produits de haute technologie, autant la conception et la programmation des systèmes permettant d'exécuter ce type d'application représentent un vrai challenge pour la communauté scientifique comme pour les industriels.

Prenons comme exemple l'application d'encodage vidéo numérique H.264 (ou MPEG-4 AVC) [106], une des plus récentes normes de compression vidéo supportée dans les lecteurs et récepteurs vidéo numériques récents en tous genres : lecteur/enregistreur Blu-ray, récepteur « HD TV » de la Télévision Numérique Terrestre (TNT) en France, etc. Cette norme amène un gain de plus de 50% d'efficacité de compression sur une large gamme de résolutions vidéo et de débits comparée au standard précédent comme MPEG-2 [85] qui est utilisé pour les DVD notamment. Une vue d'ensemble de cette application montrant les principales fonctions qui la composent est donnée figure 1.

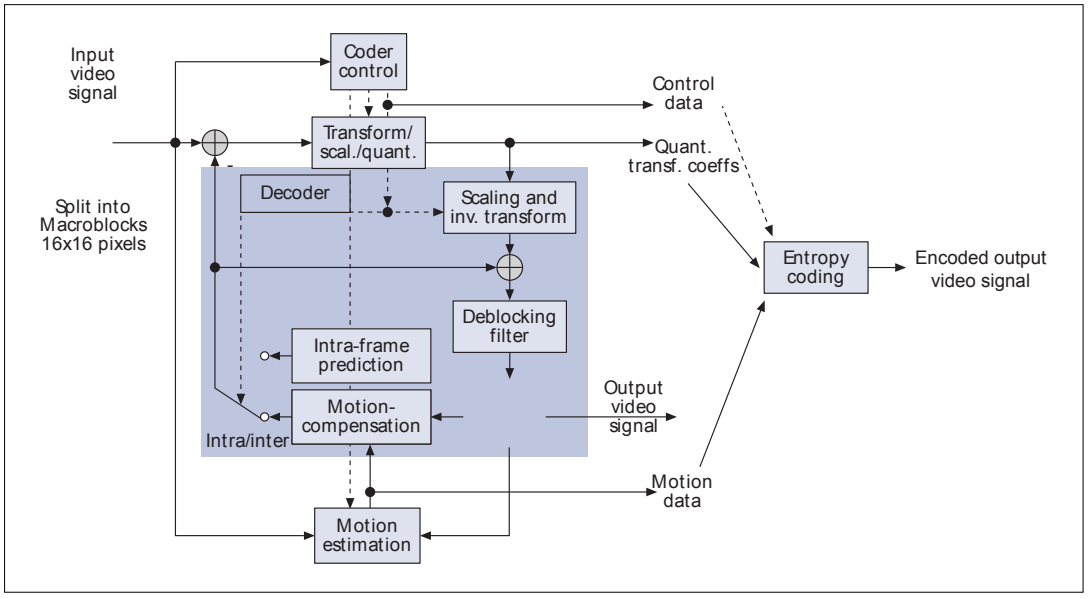


FIG. 1 – Encodeur vidéo H.264 - Diagramme fonctionnel (adapté de [71]).

La plupart des études évaluant la complexité de cette application, dont celle publiée dans l'article [26], ont démontré deux choses. D'une part, il est très difficile d'implémenter cette application sur des processeurs conventionnels (du même type que ceux équipant les ordinateurs des particuliers) car pour encoder une vidéo la puissance de calcul et la quantité de données à traiter sont très élevées : plus de 300 Giga opérations par seconde pour les calculs et plus de 460 Giga octets par seconde de bande passante pour les transferts mémoire. D'autre part, une grande partie du temps d'exécution sur un processeur classique est passée dans quelques fonctions clés comme le montre la répartition du temps d'exécution sur la figure 2 issue de [99].

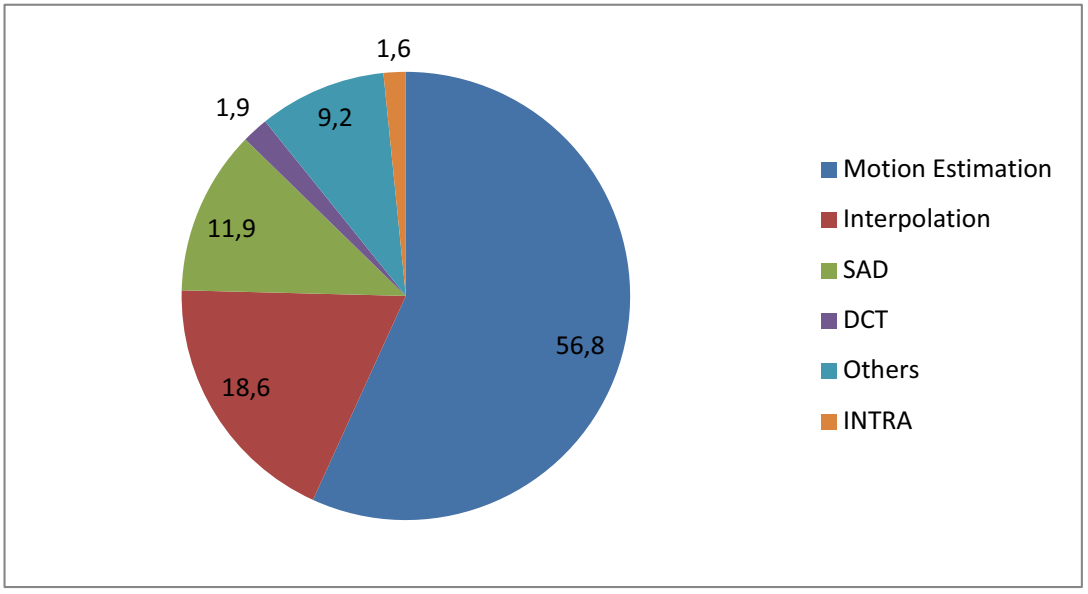


FIG. 2 – Encodeur vidéo H.264 - Répartition du temps dans les principales fonctions [99].

En fait, ce constat est valable pour de nombreuses applications multimédia récentes : elles sont de plus en plus complexes, requièrent une grande puissance de calcul et traitent de plus en plus de données. De plus, certaines fonctions clés sont potentiellement au cœur de la problématique d'implémentation de ces applications sur des processeurs à usage général.

D'un point de vue technologique, les progrès de la miniaturisation permettent aujourd'hui d'intégrer un système électronique complet dans une seule puce. On parle de système sur puce ou de SoC (*System on Chip*). Les lois empiriques décrivant l'évolution de cette miniaturisation, appelées lois de Moore, prévoyaient une croissance exponentielle de la densité des transistors (élément de base d'un circuit intégré électronique) ce qui, jusqu'à aujourd'hui, s'avère assez proche de la réalité. De nos jours, même si ce phénomène tend à ralentir dû à des limitations physiques entraînant de faibles rendements, de nouvelles approches font leurs apparitions comme par exemple la conception probabiliste et approximative dans les systèmes embarqués [86], ouvrant ainsi de nouvelles perspectives.

L'avènement des SoC a permis d'ouvrir de nouvelles opportunités quant à la réalisation de systèmes électroniques embarqués grand public supportant une multitude d'applications diverses où la taille (surface des composants), la performance et l'autonomie sont au centre des besoins. Mais cela introduit aussi une augmentation importante de la complexité à

concevoir et à utiliser efficacement ce nouveau type de système.

Les nombreuses fonctionnalités pouvant être intégrées dans un SoC poussent à une certaine hétérogénéité interne. On trouve généralement dans l'architecture d'un SoC une multitude de blocs qui ont chacun un rôle précis au sein du système.

Il est courant de trouver dans un SoC un processeur généraliste qui prend en charge le contrôle du système global, c'est un peu le chef d'orchestre. On trouve aussi des processeurs spécialisés pour des domaines d'application, on trouve bien sûr des blocs mémoires, dans certains cas des composants analogiques ou encore mixtes mais aussi et surtout des accélérateurs matériels dédiés. Ces derniers sont très importants car c'est à eux que revient le rôle de traiter les fonctions clés qui consomment une grande partie du temps d'exécution sur un processeur conventionnel. Pour l'application d'encodage vidéo H.264, ces accélérateurs sont principalement dédiés aux fonctions identifiées par la figure 2, dont l'estimation de mouvement (*Motion Estimation*). Ainsi il est très courant d'observer qu'un accélérateur soit dédié à cette unique tâche dans un SoC d'encodage vidéo H.264. C'est le cas dans l'architecture proposée dans [26] dont une photographie du circuit d'encodage est donnée figure 3. On peut y voir les blocs IME et IFE dont les rôles sont respectivement l'estimation de mouvement au niveau pixel, *Integer-pel Motion Estimation*, et à un niveau inférieur, *Fractional-pel ME*.

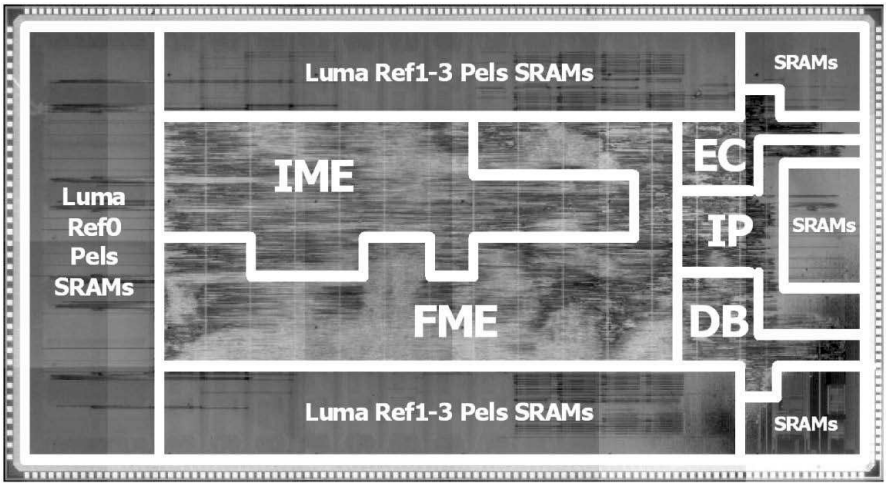


FIG. 3 – Photographie d'une puce d'encodage H.264 [26].

Conception d'accélérateurs matériels

Contraintes et enjeux

Les accélérateurs matériels sont au cœur des performances d'un SoC moderne et c'est une des raisons qui nous pousse à nous intéresser plus particulièrement dans cette thèse à la conception et à la compilation d'application pour ces systèmes.

Aujourd'hui, la conception d'un accélérateur de calcul pour les applications multimédia récentes est devenue un vrai challenge. Non seulement, nous l'avons vu, les besoins en calcul sont de plus en plus importants mais il est devenu indispensable de concevoir des accélérateurs flexibles, de par la grande variété d'applications et de normes développées et

de par leur courte durée de vie sur le marché. En effet, les accélérateurs matériels dédiés sont généralement peu flexibles du fait de leur spécialisation nécessaire à l'obtention des performances souhaitées. Ainsi, dans des systèmes multi-applications, la multiplication de ces blocs dédiés pourrait conduire à une faible utilisation globale du SoC et à une durée de vie trop courte au vue de celle des applications, en particulier dans le domaine multimédia.

De plus, les enjeux liés à la consommation d'énergie deviennent de plus en plus importants dans la conception de SoC car avec les avantages de la miniaturisation viennent aussi ses inconvénients. Parmi ces inconvénients, l'augmentation globale de la consommation électrique et donc de la dissipation d'énergie, mais aussi l'augmentation de la part de la consommation statique [1] dans les circuits électroniques dont la finesse de gravure est égale ou inférieur à 90 nm [2] posent de réels problèmes. En fait, dans les SoC embarqués et plus particulièrement mobiles, on s'accorde facilement à dire que l'autonomie est primordiale. Or les progrès technologiques dans le domaine des batteries ne vont pas à la même vitesse que ceux de la miniaturisation. Étant donné que les systèmes consomment plus et que l'autonomie des batteries n'augmente pas, il est devenu incontournable de trouver des solutions pour concevoir des systèmes à « faible consommation ».

A toutes ces problématiques viennent s'ajouter les problématiques propres aux industriels. L'augmentation des acteurs dans le domaine de la recherche et développement (R&D) en électronique grand public, en particulier asiatiques, amène une forte concurrence sur ce marché de masse où les investissements sont gigantesques. Cela vient du fait que certains coûts et temps de développement sont incompressibles. Par exemple, pour produire un circuit intégré de manière classique [3], il est nécessaire de concevoir un jeu de masques qui à lui seul coûte environ un million de dollars en technologie 90nm et à chaque fois que la finesse de gravure est divisée par deux le coût des masques est au moins multiplié par deux. Cela a tendance à orienter les développements vers une réutilisation maximale de blocs matériels déjà développés (et surtout validés) permettant de réduire les risques de problèmes lors de leur intégration dans un SoC. Cela amène aussi et surtout à l'utilisation massive de systèmes programmables dont la durée de vie est largement augmentée par la mise à disposition régulière de mises à jour logicielles.

Architecture reconfigurable à gros grain

Pour répondre à ces problématiques, nous pensons que les architectures reconfigurables à gros grain, CGRA pour *Coarse-Grained Reconfigurable Architecture*, représentent un axe de recherche privilégié et cette conviction est partagée au regard des nombreux travaux dans ce domaine durant les deux dernières décennies [53, 111]. Comme le montre la figure 4, ce type d'architecture est un compromis entre trois extrêmes. Les processeurs généralistes GPP (*General Purpose Processor*) sont des circuits denses et ils sont efficaces pour des algorithmes où le contrôle prédomine, mais ils sont chers à concevoir et consomment beaucoup d'énergie. Les circuits reconfigurables à grain fin, de type FPGA pour *Field Programmable Gate Array*, sont très flexibles et performants mais ils sont aussi peu denses car basés sur une cellule de base générique et consomment encore beaucoup d'énergie en particulier lors de leur reconfiguration qui se fait au niveau bit. Les circuits intégrés spécifiques à une application appelés ASIC pour *Application Specific Integrated Circuit* sont

[1] La consommation statique représente la consommation électrique du circuit au repos.
[2] 90 nm est la dimension du canal d'un transistor atteinte au début des années 2000.
[3] Il existe en réalité de nombreux procédés de fabrication.

eux très performants, consomment très peu d'énergie mais ne sont pas du tout flexibles et leur conception a un coût élevé. Ces deux derniers types d'architecture sont souvent privilégiés pour les accélérateurs de calcul. Cela est principalement dû à la possibilité de paralléliser massivement les traitements sur ces architectures, ce qui n'est possible que par la multiplication des cœurs dans une architecture à base de GPP.

Les accélérateurs de type CGRA sont performants car spécialisés, flexibles car reconfigurables, moins consommateurs d'énergie et moins complexes à configurer car reconfigurables à gros grain. Des avancées dans ce domaine sont encore possibles comme le montrent les récents travaux présentés dans [69] qui proposent une CGRA efficace en particulier au niveau énergétique pour les applications dites de flux de données parmi lesquelles on trouve principalement les applications multimédia, de télécommunication, de traitement du signal et de cryptographie.

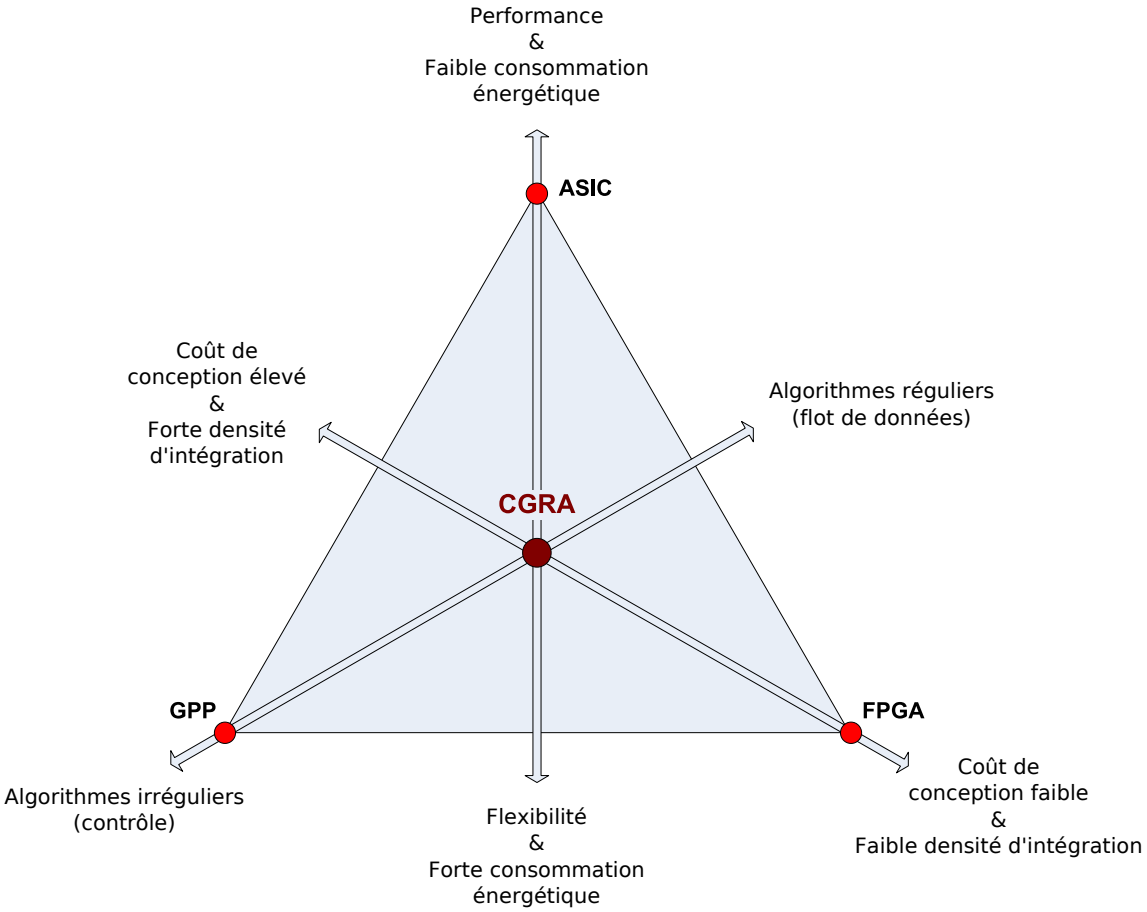


FIG. 4 – Positionnement des CGRA par rapport aux autres solutions matérielles.

Conception et compilation pour architecture reconfigurable à gros grain

La conception d'un accélérateur matériel dans le domaine de l'embarqué est fortement contrainte par les caractéristiques des parties critiques de l'application visée, on parle d'adéquation architecture ⇄ algorithme connue sous le sigle AAA[4]. Les outils d'aide à la conception de systèmes électroniques embarqués, dont le rôle est de trouver la meilleure adéquation tout en respectant les objectifs fixés par les concepteurs et les contraintes technologiques en termes de performance, de flexibilité, de surface, de consommation énergétique et de coût, se retrouvent au centre de nombreux travaux de recherche.

Dans un premier temps, il est nécessaire de définir ou de réutiliser un modèle abstrait d'architecture générique, dont une instanciation servira de cible architecturale dans le flot de compilation. Ce modèle sert de base pour la phase d'exploration de l'espace de conception durant laquelle plusieurs instanciations peuvent être étudiées de manière itérative. Les résultats en termes de performances temporelles, de consommation énergétique ou encore de surface, etc. issus de la compilation d'applications représentatives permettent de guider les concepteurs et d'anticiper tout problème de mauvais dimensionnement.

Les principales étapes d'un flot de « compilation » au niveau système pour CGRA sont représentées sur la figure 5.

- Le *Front end* de compilation permet de transformer une spécification en langage de haut niveau en une représentation intermédiaire de haut niveau, généralement sous la forme d'un graphe ou d'un arbre. Cette étape est suivie généralement d'un ensemble d'optimisations génériques indépendantes de l'architecture cible (élimination de code mort, etc.).

- Le partitionnement de l'application. C'est à cette étape que les parties critiques ou noyaux de calcul intensif sont identifiés et extraits pour être compilés pour la CGRA.

- Une compilation conventionnelle aboutissant à la génération de code exécutable de la partie de l'application s'exécutant sur un processeur généraliste.

- Une compilation pour CGRA produisant typiquement les configurations ou contextes d'exécution qui détermineront le comportement de l'architecture tout au long de l'exécution :
  - les optimisations dépendantes de l'architecture sont effectuées lorsque l'architecture est prédéfinie, si elle ne l'est pas, on parle de génération d'architecture, étape durant laquelle sont définies les différentes ressources (spécialisation et optimisation) ;
  - l'ordonnancement, le placement des traitements sur l'architecture et le routage des données entre les composants (mémoires, unités de traitement, etc.) aboutissent à la génération d'un enchainement de configurations de l'architecture.

On remarque que le flot présenté se sépare pour traiter différemment les parties identifiées comme critiques, représentant un potentiel d'accélération de l'application, et les autres parties. Ce partitionnement de l'application peut être fait en fonction de différents

[4] Adéquation Architecture Algorithme, nous utiliserons dans ce manuscrit une vision plus large en considérant l'adéquation Architecture Application qui nous paraît plus appropriée aux CGRA.

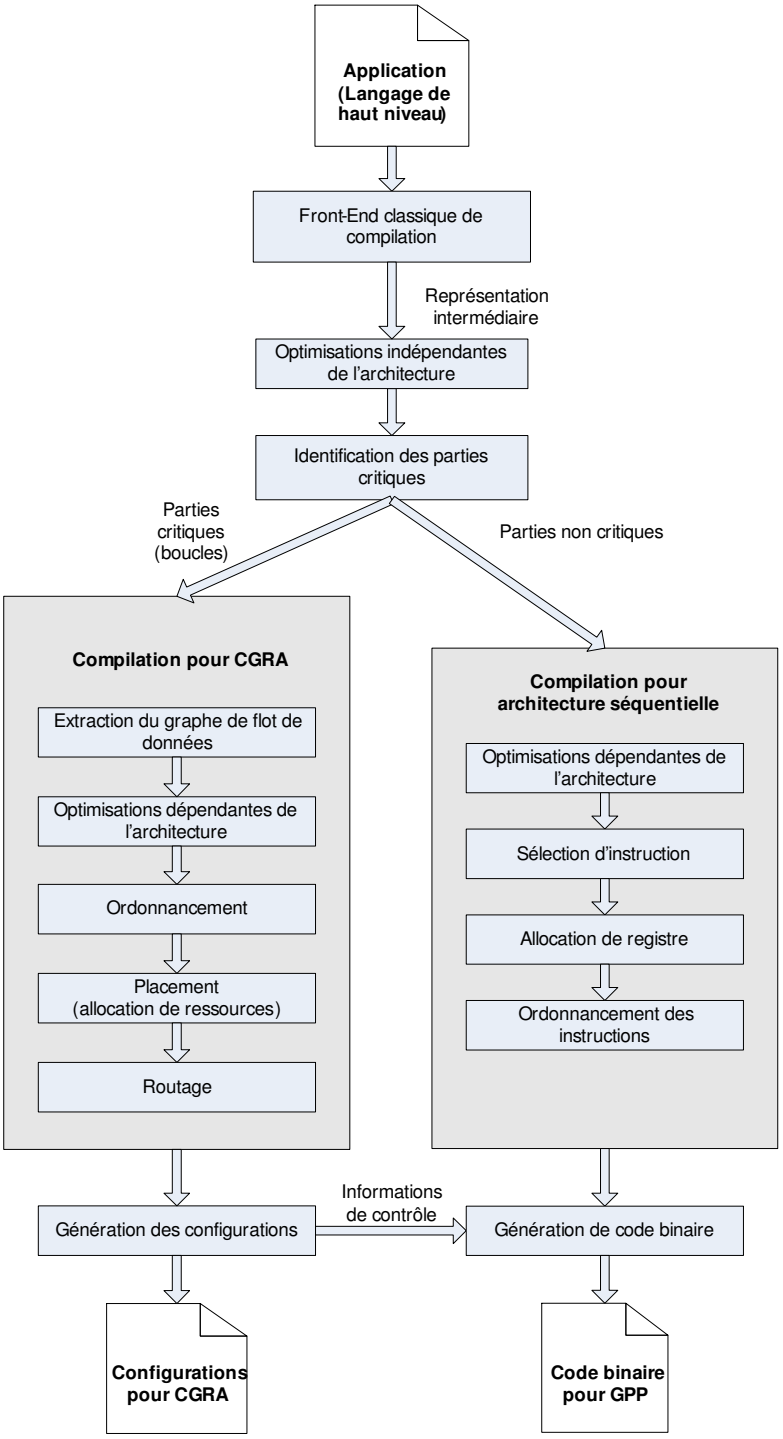


FIG. 5 – Flot de compilation générique pour CGRA.

critères. Dans la majorité des cas, le partitionnement est fait en fonction des blocs présents dans l'architecture, si elle est formée de GPP et d'accélérateurs matériels, on parle de co-développement logiciel/matériel.

Points durs dans la conception et la compilation pour CGRA

Une multitude d'architectures et de flots de conception et de compilation a été proposée ces dernières années mais il subsiste encore des points durs. Celle-ci est révélatrice de la difficulté à déterminer ce que l'on peut appeler l'adéquation Architecture $\rightleftarrows$ Flot de conception/compilation.

Reconfiguration dynamique au niveau système

La reconfiguration dynamique au niveau système présente dans les CGRA permet de modifier leurs chemins de données en cours d'exécution. La conséquence est importante sur la chaîne d'outils associés à une CGRA car cette flexibilité augmente sa complexité, comparée à une chaîne d'outils pour une architecture reconfigurable statiquement. Cela vient en partie de la forte corrélation entre les différentes étapes de conception et de compilation, particulièrement entre l'étape de synthèse matérielle qui produit des unités de traitement adaptées aux besoins applicatifs et l'étape d'ordonnancement, placement et routage de l'algorithme sur l'architecture.

L'étape de conception des unités de traitement spécialisées consiste à réaliser des opérateurs optimisés capables d'exécuter des motifs de calcul, c.-à-d. des expressions arithmétiques et logiques complexes comme la multiplication avec accumulation par exemple ou encore le papillon qui sont des motifs de calcul utilisés respectivement dans la convolution de Dirichlet ou la transformée de Fourier rapide. Les motifs utilisés sont déterminés en fonction de leur capacité à accélérer l'exécution de l'application qui les comporte.

Les résultats de l'étape d'ordonnancement, de placement et de routage de l'algorithme sur l'architecture dépendent fortement des motifs de calcul déterminés précédemment.

Par conséquent, la définition du grain des unités de traitement, l'identification des motifs de calcul qui seront synthétisés, et ensuite leur utilisation efficace sont des étapes interdépendantes rendant extrêmement complexes le flot de conception et de compilation. Mais elles sont indispensables pour permettre de répondre aux fortes contraintes imposées.

Identification et synthèse de motifs de calcul optimisés

Dans le flot de compilation DURASE de l'équipe CAIRN à l'IRISA [73, 72], une des étapes clés consiste à identifier et extraire les motifs de calcul récurrents et/ou intéressants pour accélérer une application à partir de sa représentation intermédiaire. Dans le cas de DURASE, l'architecture cible est un processeur spécialisé pour une application à travers une extension comportant une unité fonctionnelle reconfigurable. Dans le but de synthétiser une unité fonctionnelle reconfigurable au niveau système nous avons cherché à fusionner les motifs définis précédemment. Toutes les méthodes proposées dans la littérature permettent d'optimiser un seul paramètre (généralement la surface de l'unité reconfigurable) mais ne permettent pas de prendre en compte les autres contraintes architecturales et/ou technologiques. Il peut en résulter un déséquilibre qui se manifeste par exemple par des unités petites mais peu performantes ou au contraire des unités performantes mais dont la surface est importante.

Déploiement d'application

Parmi les différentes étapes de compilation pour CGRA, l'ordonnancement, le placement et le routage du ou des noyaux de calcul sur l'architecture dont les unités de calcul ont été spécialisées et peuvent se reconfigurer, sont au cœur de l'outil de conception et de compilation. C'est durant ces étapes que les opérations seront ordonnancées dans le temps, placées sur des unités de calcul et où les communications entre unités de calcul et mémoires seront routées. Le but de ces étapes est de tirer au maximum partie des spécificités de l'architecture tout en conservant la sémantique du programme (dépendance de données entre opérations, etc.). Les principales spécificités d'une CGRA sont principalement : le parallélisme de l'architecture (différent selon le modèle choisi), la reconfiguration dynamique au niveau système (une unité peut exécuter différents motifs de calcul) et l'interconnexion des unités de traitements. Il est donc facile de comprendre que l'utilisation efficace d'une CGRA est grandement tributaire de ces étapes. De plus, dans un flot itératif d'exploration de l'espace de conception, les informations extraites à chaque itération sont utilisées pour instancier une architecture mieux adaptée au compromis souhaité, et de converger ainsi vers une solution répondant aux objectifs visés.

Dans le but de maximiser les possibilités offertes par l'architecture, ces étapes peuvent être traitées simultanément mais le problème devient alors plus complexe à résoudre. C'est pourquoi de nombreuses approches ont été proposées par la communauté scientifique.

Parmi toutes les approches proposées dans la littérature, on peut observer deux tendances.

La première correspond aux méthodes permettant d'obtenir un résultat optimal. Selon les approches et les objectifs des travaux issus de la littérature on trouve l'utilisation de la programmation linéaire en nombres entiers (ILP pour *Integer Linear Programming*), de l'algorithme de recuit simulé ou encore de méthodes d'améliorations itératives (*iterative improvement algorithms*). Mais ces méthodes ont un temps de compilation qui croît exponentiellement en fonction de la taille du problème. C'est pourquoi elles sont principalement utilisées comme référence ou encore pour l'exploration de l'espace de conception, pour laquelle on accepte facilement des temps de calcul importants. La principale difficulté est de résoudre des problèmes de grande taille.

La seconde tendance correspond aux méthodes permettant d'obtenir rapidement un bon résultat sur des cas pratiques par l'utilisation d'heuristiques. La technique de *list scheduling* est une des techniques les plus utilisées, étendues et/ou adaptées et fait aussi souvent office de référence. On trouve aussi une multitude de techniques simples comme les algorithmes ASAP (*As Soon As Possible*) et ALAP (*As Last As Possible*) dont la différence représente le degré de liberté d'ordonnancement d'une opération. Mais ces algorithmes ont des lacunes. Dans certains cas, ils ne prennent pas en compte précisément les contraintes architecturales dans leur approche de résolution. De plus, il est souvent très difficile (voire impossible) de les adapter à un autre contexte que celui pour lequel ils ont été proposés. Par exemple, quelques différences dans l'architecture peuvent rendre inutilisables la solution proposée.

La principale difficulté dans la résolution du problème d'ordonnancement, placement et routage réside dans la résolution du problème d'optimisation globale. Par exemple, minimiser le temps d'exécution ou encore minimiser le nombre de (re)configurations et cela tout en prenant en compte les contraintes architecturales issues des spécificités de

l'architecture. Avec la méthodologie basée sur la programmation par contraintes (notée CP pour *Constraint Programming*), il est possible de répondre à cette problématique. En effet, comme le mentionne [60], la CP est une approche prometteuse pour produire efficacement des solutions de haute qualité pour la synthèse d'architectures complexes impliquant des optimisations multi-objectifs. Cette méthode a déjà été utilisée dans un contexte proche de celui de cette thèse [66] mais l'utilisation de cette approche dans le cadre de CGRA implique d'exprimer des contraintes architecturales spécifiques.

Problématique et contributions de la thèse

Les architectures reconfigurables représentent un énorme potentiel d'accélération d'applications embarquées mais les outils actuels n'arrivent pas encore à tirer pleinement partie de ces architectures de par la complexité des tâches à effectuer. Une manière personnelle de synthétiser la problématique générale de conception de systèmes embarqués est de mettre en avant les adéquations mises en jeu : d'une part l'adéquation architecture $\rightleftarrows$ application, d'autre part l'adéquation architecture $\rightleftarrows$ alot de conception/compilation. On peut donc en déduire qu'il existe une relation d'adéquation entre ces trois notions : application $\rightleftarrows$ architecture $\rightleftarrows$ alot de conception/compilation. Dans cette thèse, nous proposons des solutions à certains aspects liés à l'adéquation applications multimédia $\rightleftarrows$ CGRA $\rightleftarrows$ flot de conception et de compilation en utilisant la programmation par contraintes.

Contributions de la thèse

Les travaux présentés dans ce manuscrit s'inscrivent dans la problématique de modélisation des aspects architecturaux sous forme de contraintes dans le cadre de la compilation et de la conception d'un accélérateur reconfigurable à gros grain. Il est proposé dans cette thèse la résolution de certaines étapes d'un flot de conception et de compilation, basées sur la méthodologie de programmation par contraintes, qui vise à transformer un code de haut niveau en une série de configurations pour CGRA. L'objectif est de proposer des solutions globales dans un temps de compilation maîtrisé en prouvant, si possible, l'optimalité de la solution ce qui permet d'aboutir à l'exécution efficace de traitements sur une architecture reconfigurable à gros grain.

La première contribution présentée a trait à la fusion de motifs de calcul sous contraintes architecturales. Le but de cette étape dans le flot de compilation est de générer des unités de calcul performantes reconfigurables au niveau système dont la surface est la plus petite possible. Cette technique a été développée dans le cadre de l'extension du jeu d'instructions d'un processeur généraliste embarqué par le biais d'une unité reconfigurable à gros grain.

La seconde et principale contribution présentée dans ce manuscrit répond à la problématique de modélisation à l'aide de contraintes des aspects architecturaux dans le cadre de l'ordonnancement, du placement et du routage de noyaux de calculs intensifs sur un accélérateur reconfigurable à gros grain.

L'approche utilisée dans les deux contributions proposées est basée sur la méthodologie de programmation par contraintes. Cette approche nous a permis, d'une part, de modéliser les aspects applicatifs et architecturaux d'une CGRA sous forme de contraintes et, d'autre part, de résoudre de manière globale les problèmes d'optimisation adressés tout en prouvant

l'optimalité de la solution obtenue dans la majorité des cas. Dans la première contribution l'objectif est de minimiser la surface de l'unité reconfigurable synthétisée. Dans la deuxième contribution, l'objectif est la minimisation du temps d'exécution ou de la consommation d'énergie lors de l'exécution d'un noyau de calcul intensif sur un accélérateur reconfigurable à gros grain. Dans les deux cas, l'objectif est de contraindre la solution par un modèle de contraintes applicatives et architecturales.

Organisation du document

La suite du manuscrit se décompose en cinq parties.

Le premier chapitre présente un état de l'art des architectures reconfigurables à gros grain ainsi que leurs environnements de conception et de compilation. Ce chapitre a pour objectif de présenter l'axe de recherche sur lequel se situent les contributions décrites dans cette thèse. Après avoir prouvé la pertinence de cet axe de recherche, cette première partie développe les problématiques propres à ce domaine.

Le deuxième chapitre présente la programmation par contraintes comme une nouvelle approche pour la conception et la compilation dans le cadre des architectures reconfigurables à gros grain. Ce chapitre présente aussi certains outils développés dans l'équipe CAIRN de l'IRISA et utilisant cette méthodologie.

Le troisième chapitre est consacré à la première contribution de cette thèse : le modèle de contraintes pour la fusion d'unités fonctionnelles reconfigurables sous contraintes architecturales et technologiques dans le cadre de l'extension du jeu d'instructions d'un processeur embarqué.

Le quatrième chapitre expose le modèle de contraintes pour l'ordonnancement, le placement et le routage de noyaux de calculs intensifs sur une architecture reconfigurable à gros grain. Cette architecture cible a été définie dans le cadre du projet collaboratif ROMA [23]. Ce projet a pour ambition de définir et de développer une CGRA optimisée pour accélérer les applications multimédia ainsi qu'un flot de conception et de compilation basé sur la CP. Les principales propriétés offertes par la CGRA ROMA sont : une faible consommation d'énergie, une forte densité d'intégration et une grande flexibilité. Ces caractéristiques sont primordiales dans les systèmes multimédia embarqués. Ce chapitre développe la seconde et principale contribution de cette thèse.

La conclusion de ce manuscrit permet au lecteur de revenir sur les principales problématiques à l'origine de mes travaux de recherche, de synthétiser les contributions présentées au cours des deux chapitres précédents pour en extraire les principaux avantages et inconvénients. De ces derniers découlent un certain nombre de perspectives de recherche qui seront ici mises en évidence.

Chapitre 1

Architectures reconfigurables à gros grain - État de l'art

De nos jours, les architectures reconfigurables s'imposent comme une cible potentielle pour accélérer des applications embarquées de par le compromis flexibilité/performance qu'elles proposent. Cette tendance se traduit, dans la recherche comme dans l'industrie, par de nombreuses contributions destinées à être intégrées dans des produits de grande consommation. Ces architectures modernes sont composées de nombreuses unités de calcul, homogènes ou hétérogènes, pouvant être spécialisées pour un domaine, une classe d'applications, voire pour une seule application.

Le début de ce chapitre propose une définition d'une architecture reconfigurable, de la notion de grain et de couplage. Ensuite, sont présentées, d'une part les principales caractéristiques d'une architecture reconfigurable à gros grain, CGRA pour *Coarse-Grained Reconfigurable Architecture*, d'autre part les différentes étapes de développement de ces architectures et les problématiques associées. Enfin, la présentation d'une sélection d'architectures reconfigurables à gros grain dédiées à l'accélération d'applications multimédia et de leurs outils de conception et de compilation proposés par des laboratoires de recherche et par un industriel permettra d'illustrer ce chapitre.

1.1 Architectures reconfigurables

1.1.1 Origines et définitions

Aujourd'hui encore, le terme reconfigurable est associé aux « circuits logiques programmables » et surtout au type d'architecture qui l'a popularisé, le FPGA (*Field-Programmable Gate Array*). Introduit sur le marché au milieu des années 80 par l'un des leaders mondiaux actuels (Xilinx), le FPGA est formé de trois différents éléments de base : la table de correspondance ou LUT (*Look Up Table*) ou encore CLB (*Configurable Logic Bloc*) selon le fabricant, le multiplexeur et la bascule. A noter que la notion de reconfigurable, telle qu'on la connaît aujourd'hui, est bien plus ancienne et pourrait être attribuée à Estrin qui présente dans son papier de 1960 (récemment évoqué dans [39]) une architecture hybride composée d'un processeur standard et d'une matrice matérielle « reconfigurable ».

Définition 1.1 : *Une architecture reconfigurable est une architecture capable de modifier son comportement par la modification du schéma de connexion entre ses éléments de base.*

1.1.2 Évolutions

Les architectures reconfigurables ont énormément évolué ces dernières années pour satisfaire des besoins applicatifs toujours croissants. Une des évolutions est apparue comme incontournable pour exploiter pleinement la flexibilité apportée par les FPGA ; il s'agit de la reconfiguration dynamique. En effet, avant cette évolution, il fallait arrêter toute activité sur le composant avant de pouvoir en modifier son comportement et obtenir ainsi une nouvelle fonctionnalité. C'est à la fin des années 90 que la reconfiguration dynamique partielle est apparue sur les composants Xilinx notamment, permettant d'améliorer le compromis performance/flexibilité.

Parmi ces évolutions, on trouve aussi l'apparition de différents niveaux de reconfiguration : du plus petit (niveau bit) au plus gros (unité fonctionnelle), on parle de granularité ou de grain de reconfiguration. Les caractéristiques d'une architecture reconfigurable varient selon la granularité à laquelle elle peut être reconfigurée. On distingue communément trois niveaux de reconfiguration : logique, fonctionnel et système.

1.1.2.1 Reconfiguration au niveau logique (grain fin)

La reconfiguration au niveau logique appelée aussi reconfiguration à grain fin est typiquement représentée par les systèmes FPGA[1]. Leur reconfiguration se fait au niveau bit ce qui permet une grande flexibilité, comme notamment d'adapter la largeur des données en fonction des besoins. Mais à ce niveau, certains désavantages apparaissent.

Tout d'abord, on observe qu'une grande partie de la surface d'un circuit FPGA est occupée par les multiplexeurs et les bascules utilisées pour réaliser le câblage entre éléments logiques réalisant la fonctionnalité souhaitée et que ces derniers n'occupent qu'une faible surface. De plus, les outils de placement et de routage (P&R) utilisés pour configurer un FPGA sont très complexes et peu efficaces car le problème de P&R est connu pour être difficile à résoudre. Cela l'amène même à être considéré comme « nuisible » par Hartenstein qui écrit : « Avec de sévères difficultés, les FPGA commercialisés routent dans la plupart des *designs* moins de 80-90% des LUT disponibles et même dans certains cas seulement environ 50% » [52].

Le coût global de la reconfiguration est un autre désavantage majeur. En effet, la mémoire utilisée pour stocker les différentes configurations, le temps et l'énergie nécessaires pour effectuer une reconfiguration (importants transferts de données de configuration) sont loin d'être négligeables dans la conception d'un système sur FPGA.

Parmi les solutions proposées pour pallier ces désavantages, les architectures reconfigurables à un plus gros grain ont fait leur apparition.

1.1.2.2 Reconfiguration au niveau fonctionnel (gros grain)

Ce niveau de reconfiguration, nommé aussi reconfiguration à gros grain, permet d'améliorer la surface, la consommation et la performance des systèmes comparés aux FPGA tout en en maîtrisant la perte de flexibilité pour ne pas la rendre pénalisante. Selon [41], on peut observer deux principales familles d'architecture : les architectures orientées chemin de données et celles orientées instructions. Dans la première famille, la fonctionnalité des unités de calcul est fixée par une étape de configuration et le flot de données entre ces

[1] On trouve néanmoins de plus en plus de macro-cellules pré-câblées dans les FPGA les plus récents.

unités est organisé par un contrôle spécifique sur les interconnexions. Dans la seconde, chaque élément exécute une suite d'opérations définies par un élément de contrôle à partir d'une instruction. Il existe aussi des architectures pouvant supporter les deux types de fonctionnement.

1.1.2.3 Reconfiguration au niveau système

Les architectures reconfigurables au niveau système, communément appelées processeurs programmables, sont des architectures purement orientées instructions. Plusieurs modèles, plus ou moins anciens, ont été proposés en fonction de la manière dont sont définies et traitées les instructions. On différencie ainsi principalement les processeurs dont le jeu d'instructions est réduit (RISC - *Reduce Instruction Set Processor*) ou complexe (CISC - *Complex Instruction Set Processor*) ; ceux dont on sépare le flot d'instructions et de données comme le modèle d'architecture Harvard des processeurs de traitement du signal (DSP - *Digital Signal Processor*) dans lequel sont physiquement séparées les mémoires d'instructions et de données ainsi que les bus associés ; ou encore les processeurs qui adressent plusieurs unités fonctionnelles en parallèle dans la même instruction, exploitant ainsi le parallélisme d'instructions (processeurs VLIW - *Very Long Instruction Word*) ; enfin ceux qui comportent des mécanismes matériels complexes permettant une exécution concurrente des instructions (processeurs superscalaires) et ceux qui sont capables d'exécuter des instructions vectorielles (processeurs vectoriels).

1.1.2.4 Reconfiguration à plusieurs niveaux

Certaines architectures supportent plusieurs niveaux de reconfiguration comme c'est le cas de l'architecture DART présentée plus loin (1.4.1) qui combine l'utilisation d'un bloc FPGA pour les traitements logiques et de plusieurs blocs reconfigurables à gros grain pour les traitements arithmétiques.

1.1.2.5 Remarque

Plusieurs classifications ont été proposées concernant les architectures reconfigurables, permettant ainsi de mieux appréhender leur diversité. Le lecteur souhaitant élargir son champ de connaissance sur la diversité des architectures reconfigurables peut se référer à la taxonomie publiée par Radunovic [90] qui sert souvent de référence.

1.2 Architectures reconfigurables à gros grain

Les CGRA représentent une alternative pour la conception de systèmes embarqués multimédia où les contraintes de taille, de consommation énergétique, de performance mais aussi de flexibilité sont très fortes. Elles sont un compromis entre les FPGA dont la grande flexibilité a des conséquences négatives pour une intégration dans un système embarqué en particulier sur les aspects de consommation énergétique et de densité d'intégration, et les processeurs programmables dont la généricité implique une sous-utilisation des ressources et donc une taille et une consommation énergétique plus importante qu'une CGRA.

La définition précédente d'une architecture reconfigurable (cf. 1.1.1) est un peu formelle et bien adaptée au FPGA. Elle peut être élargie comme telle.

Un système reconfigurable est un système capable d'adapter ses ressources en fonction d'un traitement. « Une architecture matérielle est généralement constituée d'une collection d'éléments de calcul, de stockage et d'un dispositif d'échange d'informations entre ces éléments et, éventuellement, d'un contrôleur pour gérer l'ensemble. Une configuration précise à la fois la fonctionnalité des éléments et les chemins de données entre ces éléments, voire le contrôle. Le terme *reconfiguration* apporte, quant à lui, une notion supplémentaire de dynamicité. Il exprime le fait qu'une configuration n'est pas fixée définitivement (comme pour les circuits ASIC[2]), mais qu'il existe un mécanisme qui, au cours du temps, puisse modifier à la fois la fonctionnalité des éléments de calcul et leur interconnexion. » [34].

Il est alors plus aisé de définir la notion de CGRA en opposition au grain fin.

Définition 1.2 : *Contrairement aux architectures reconfigurables à grain fin, la reconfiguration d'une CGRA s'effectue sur des chemins de données dont la taille est supérieure au bit.*

Un grand nombre de CGRA ont été proposés ces dernières années. Hartenstein propose de revisiter dix ans de recherche et de développement sur le reconfigurable (1990-2000) démontrant que cet axe de recherche est prometteur et mettant en avant le besoin de mettre en place des outils de compilation performants pour tirer pleinement partie de ces architectures [53]. De nombreuses autres études sur les systèmes reconfigurables et les méthodes de conception et de compilation associées ont été publiées ces dernières années [60, 111, 108, 109, 28]. Chacune de ces études présente les problématiques liées à la conception et à la programmation des CGRA sous différents angles.

Celle de Theodoridis [108] nous est apparue comme particulièrement pertinente au vu des travaux présentés dans cette thèse car elle est dédiée au CGRA et elle propose de les différencier en fonction de leur flexibilité et ainsi de les classer en deux catégories : les systèmes spécifiques à un domaine d'application notés ADSS (*Application Domain-Specific System*) et les systèmes spécifiques à une classe d'applications notés ACSS (*Application Class-Specific System*), la première catégorie étant plus flexible que la seconde. Cette distinction permet d'adapter la méthode de conception d'une CGRA en fonction du type d'architecture choisi selon les caractéristiques du spectre applicatif ciblé mettant en relation trois aspects majeurs : le modèle d'architecture (ici ADSS ou ACSS), les applications (domaine ou classe) et le flot de conception et de compilation. Cette relation est discutée à la fin de ce chapitre.

Une chose est claire, la caractéristique omniprésente dans les CGRA de tout type réside de façon incontestable dans le fait qu'elles sont conçues pour accélérer des applications et que donc l'adéquation architecture application est au cœur des problématiques rencontrées lors de leur conception.

1.2.1 Couplage Processeur - CGRA

La place d'une CGRA au sein d'un système sur puce (ou SoC pour *System on Chip*) peut varier. La majorité du temps, une CGRA est associée à un processeur programmable qui exécute le reste de l'application, c.-à-d. hors parties critiques ou potentiellement accélérables. Ce dernier possède une mémoire locale de type cache. La notion de couplage est

[2] *Application Specific Integrated Circuit*, circuits intégrés spécifiques à une application.

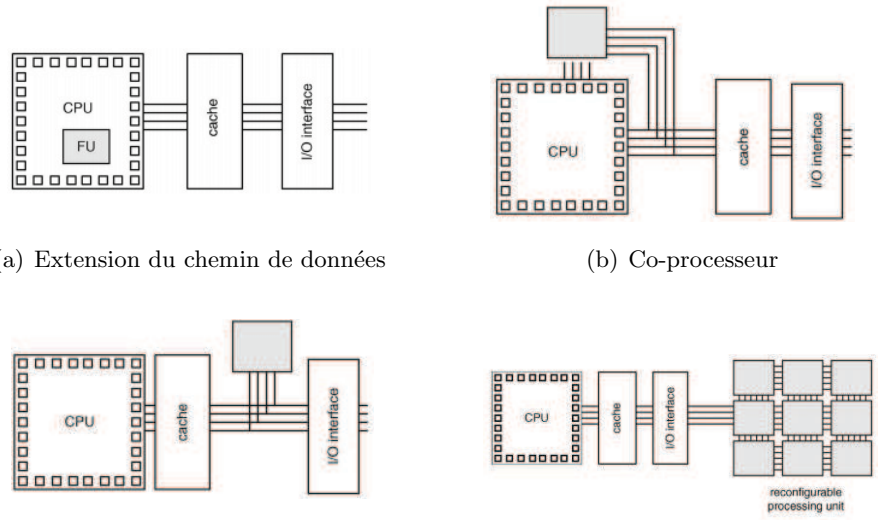
utilisée pour exprimer la relation entre le processseur et la CGRA. La figure 1.1, adaptée de [109], illustre les différents types de couplage ainsi que les dénominations associées. Du plus proche au plus éloigné du processeur (noté CPU sur la figure pour *Computational Processing Unit* ce qui signifie littéralement « unité centrale de traitement »), les types de couplage sont :

– L'extension du chemin de données d'un processeur par une ou plusieurs unités reconfigurables à gros grain (figure 1.1(a)), on parle communément d'ASIP (*Application-Specific Instruction-set Processor*). Dans ce cas, la spécialisation prend la forme de nouvelles instructions disponibles pour le compilateur ou pour le programmeur. Les deux modèles d'exécution, séquentielle ou parallèle, peuvent être utilisés.

– Le co-processeur (figure 1.1(b)), c'est un schéma dans lequel le processeur contrôle la CGRA en lui attribuant des tâches de calcul intensif courtes qui ne requièrent pas ou peu de contrôle, ainsi le processeur peut exécuter des tâches en parallèle. La mémoire cache est partagée entre le processeur et le coprocesseur ce qui implique la mise en place de mécanismes de synchronisation plus complexes que dans le cas précédent.

– Un couplage intermédiaire (figure 1.1(c)) dans lequel, à l'inverse du co-processeur, le cache n'est pas visible par la CGRA. Dans ce cas, cette dernière est plus autonome et accède uniquement à la mémoire externe (mémoire globale partagée) à travers les interfaces d'entrée/sortie partagées avec le processeur.

– L'accélérateur (figure 1.1(d)), c'est le couplage le plus lâche dans lequel la CGRA est autonome vis-à-vis du processeur. Les communications avec celui-ci étant lentes, ce type de couplage est intéressant lorsque la CGRA prend en charge de longues périodes de calculs ne nécessitant que très peu de communications avec le CPU.



(a) Extension du chemin de données

(b) Co-processeur

(c) Couplage intermédaire

(d) Accélérateur

FIG. 1.1 – Différents types de couplage Processeur - CGRA (adapté de [109]).

1.2.2 Principales caractéristiques d'une CGRA

Une CGRA totalement générique est un non-sens car son architecture est conçue en fonction des caractéristiques inhérentes aux applications à accélérer. Ainsi de nombreuses architectures ad-hoc ont été proposées ces dernières années. Néanmoins, certaines caractéristiques restent invariantes. Ces invariants peuvent être utilisés pour modéliser une CGRA à un haut niveau d'abstraction permettant une exploration de l'espace de conception architecturale dans le but de déterminer, en fonction des applications, quelle est l'instance du modèle qui répond le mieux aux contraintes imposées aux et par ses concepteurs. Les principaux éléments formant une CGRA sont :

- un ensemble d'unités de calcul à gros grain qui peuvent être reconfigurables appelées dans ce cas RPU (*Reconfigurable Processing Unit*),
- des mémoires locales (optionnelles) permettant de stocker les résultats intermédiaires, ces mémoires peuvent être directement incluses dans les RPU,
- un ou plusieurs réseaux d'interconnexion pouvant connecter les deux types d'éléments précédents, eux aussi peuvent être reconfigurables,
- un contrôleur pilotant l'exécution et/ou la reconfiguration du système,
- une mémoire de configuration contenant les différents contextes d'exécution.

La figure 1.2 représente un modèle de CGRA associé à un processeur hôte et une mémoire globale. L'architecture présentée est très proche de celle proposée par Theodoridis à la différence près que des mémoires locales ont été ajoutées dans la partie reconfigurable à gros grain. La reconfiguration peut s'effectuer principalement au niveau des RPU et/ou du réseau d'interconnexion.

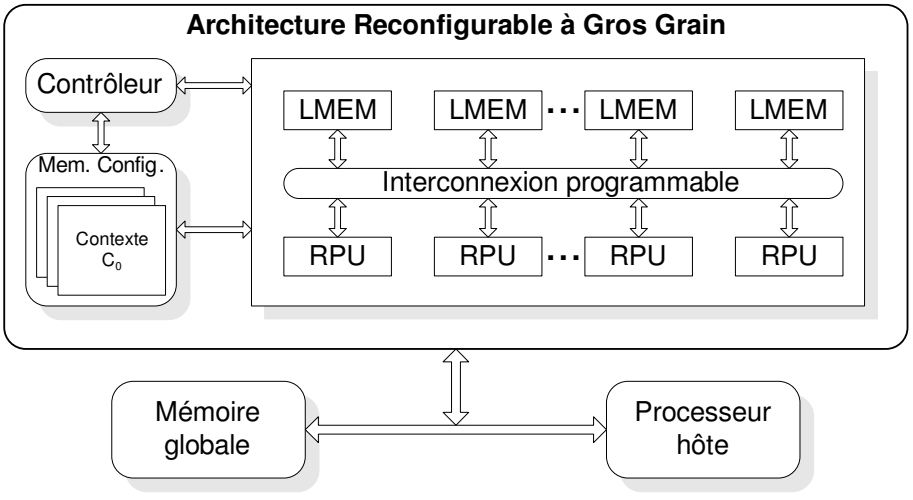


FIG. 1.2 – Modèle générique de CGRA.

1.2.2.1 Les unités de calcul reconfigurables à gros grain - RPU

Les RPU sont des unités de calcul performantes, capables d'exécuter les opérations propres à une classe ou à un domaine d'application. Elles sont flexibles car elles sont configurées suivant les différents contextes stockés dans la mémoire de configuration. L'évolution de la granularité des RPU a conduit à deux approches.

Les RPU peuvent jouer le rôle d'unités arithmétiques et logiques classiques, c.-à-d. que chacune exécute une opération simple à chaque fois (une addition, un « ou » logique, etc.). Cela n'empêche pas de les spécialiser et de les optimiser en fonction des besoins, par exemple une RPU peut se contenter de ne supporter que certaines opérations.

En opposition à cette approche, les RPU peuvent exécuter des expressions arithmétiques et logiques complexes comportant plusieurs opérations simples (la multiplication avec accumulation par exemple). Dans ce cas, la granularité des RPU est plus grande ce qui peut permettre une optimisation plus poussée de l'unité de calcul en termes de performance, de surface et de consommation d'énergie.

La figure 1.3, issue de [7], montre l'évolution des architectures au travers de ces deux approches. De plus, y figure un parallèle avec les architectures à grain fin sur lesquelles ces deux approches peuvent aussi être appliquées (le terme PLA signifie *Programmable Logique Array* et EGRA *Expression-Grained Reconfigurable Architecture*). Ce qui est important de noter ici, c'est la relation entre la représentation de l'application sous forme de graphe et le grain des unités de calcul de l'architecture vis-à-vis du type de placement.

Remarque : Dans la suite du manuscrit, nous utiliserons le terme de motif de calculs qui est défini comme une expression arithmétique et logique complexe pouvant être exécutée par une RPU.

La granularité des RPU peut donc varier du simple opérateur sur une largeur de quelques bits à un opérateur complexe pouvant exécuter différentes expressions ou motifs de calcul sur une largeur de données de 32/40 bits.

Concernant le niveau de spécialisation des RPU, on observe aussi deux tendances. La première consiste à spécialiser fortement les unités de traitement en fonction des applications visées ; cette approche ne permet pas de supporter un large spectre d'application (ou un domaine entier). Dans ce cas, des bibliothèques d'opérateurs spécialisés et optimisés peuvent être fournis (cas des ACSS). La seconde tendance correspond à l'utilisation d'opérateurs plus génériques, mais eux aussi optimisés, permettant de couvrir un spectre applicatif plus large sans avoir à fournir un ensemble d'opérateurs spécialisés (cas des ADSS).

1.2.2.2 Les mémoires

La hiérarchie mémoire des CGRA comporte plusieurs niveaux, comme celle présente dans les GPP. Par contre les mécanismes mis en oeuvre pour gérer les différents niveaux de cette hiérarchie sont plus simples dans le cas des CGRA que pour les GPP de par la nécessité d'une exécution déterministe en termes de performance permettant de garantir une exécution temps-réel par exemple. Ainsi sont privilégiées les mémoires adressables rapidement et dont le temps d'accès est constant comme les *scratch pad* au dépend des

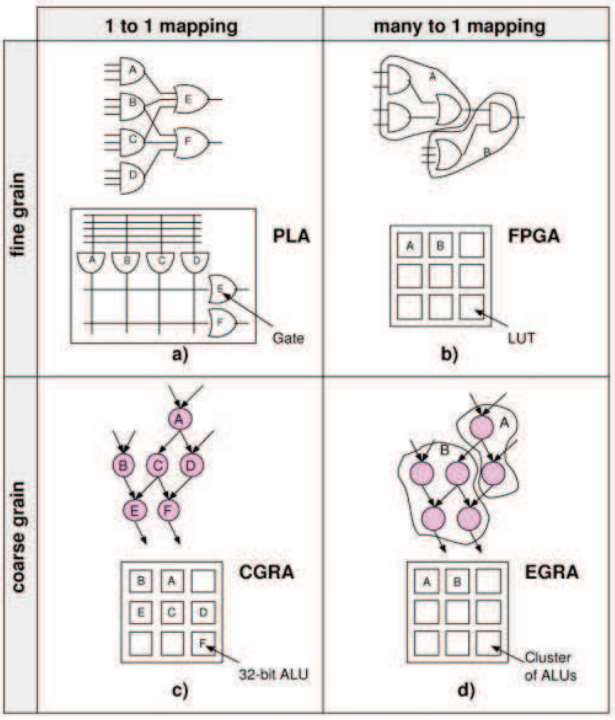


FIG. 1.3 – Illustration de l'évolution de la granularité des opérateurs de base pour les FPGA et les CGRA [7].

mémoires de type cache par exemple. Voici les principaux niveaux que l'on peut rencontrer dans la littérature, tous ne sont pas utilisés systématiquement.

Au plus près des unités de calcul, les RPU elles mêmes peuvent contenir des registres internes, en plus de ceux que l'on trouve généralement en entrée/sortie permettant une exécution pipelinée.

Ensuite, des mémoires locales peuvent-être attachées à une ou plusieurs RPU ou accessibles par tous. Leur dimensionnement est primordial car un manque de mémoire peut conduire à une sous utilisation des RPU. Les transferts incessants ralentissent alors l'exécution globale, ou au contraire, un surdimensionnement de ces mémoires conduit à une augmentation inutile de la surface mais aussi de l'énergie statique consommée. Leur profondeur, leur largeur et leur nombre sont les principaux axes de dimensionnement adressés lors de l'exploration de l'espace de conception.

Quant à la mémoire de configuration, elle est relativement petite, étant donné que la reconfiguration des opérateurs ne nécessite que quelques bits pour coder les différentes opérations ou expressions exécutables par ceux-ci. De plus, l'interconnexion se programme avec la même granularité ne requérant ainsi que peu de mémoire pour y stocker les différents contextes d'exécution. Néanmoins, la taille de la mémoire de configuration est amenée à augmenter proportionnellement à l'irrégularité des algorithmes exécutés sur l'architecture reconfigurable.

Enfin, une mémoire globale peut être accessible par la CGRA. Elle est généralement

partagée avec d'autres blocs du SoC.

1.2.2.3 Le réseau d'interconnexion

Le réseau d'interconnexion permet le transfert de données entre les différents éléments de calcul et de mémorisation. Comme pour les RPU, les caractéristiques du réseau d'interconnexion sont représentatives du type et de la largeur du spectre des applications visées. De plus, la flexibilité, la performance et la consommation énergétique d'une CGRA sont grandement dépendantes de son réseau d'interconnexion.

On peut différencier trois classes de réseaux d'interconnexion : les réseaux globaux, les réseaux point-à-point et les réseaux hiérarchiques.

Dans les **réseaux globaux**, tous les éléments (RPU, mémoire, etc.) de l'architecture peuvent communiquer entre eux directement ou indirectement. Dans cette classe d'interconnexion on distingue différents type de réseaux.

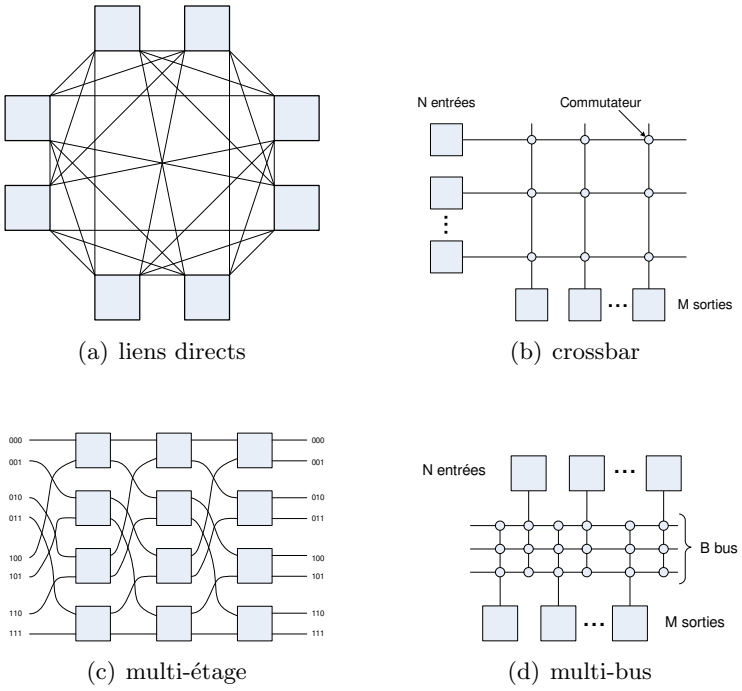
- Les réseaux totalement connectés permettent de relier tous les éléments qui y sont connectés. Plusieurs réalisations sont possibles, soit par un lien direct (figure 1.4(a)) soit par l'utilisation d'un bus global par entrée qui peut être connecté à n'importe quelle sortie par l'intermédiaire d'un commutateur comme le montre la figure 1.4(b). Cette implémentation est appelée *crossbar*.

- Les réseaux « multi-étages » permettent de réduire le nombre de commutateurs par rapport aux réseaux *crossbar* tout en conservant la flexibilité de connexion. Un exemple de réseau multi-étage nommé « Omega » est représenté sur la figure 1.4(c) (réseau Omega 8x8).

- Les réseaux « multi-bus » sont une autre alternative, plus optimisée car ils n'utilisent que le nombre de bus nécessaires correspondant au nombre maximum de transactions concurrentes. Un exemple de réseau de ce type est donné sur la figure 1.4(d) (N entrées, M sorties et B bus).

Les réseaux globaux sont très flexibles, relativement faciles à mettre en oeuvre et presque incontournables quand il s'agit de cibler un large domaine d'application (les ADSS selon Theodoridis) comme les applications de traitement du signal. Par contre leur réalisation est coûteuse voire prohibitive en surface quand le nombre d'éléments à interconnecter est grand. De plus, leur temps de traversée et leur consommation peuvent être élevés car proportionnels au nombre d'éléments connectés. Le tableau 1.1 donne une comparaison de trois types de réseaux globaux en termes de nombre de commutateurs, nombre de bus et nombre total de commutateurs par bus [119].

Les **réseaux point-à-point** sont conçus dans le but de favoriser les communications locales rapides. Parmi la multitude de réseaux de ce type, les plus utilisés sont les réseaux maillés qui permettent des communications de voisinage. Cette topologie de base est la plus adaptée au traitement d'application nécessitant beaucoup de traitement de données. La figure 1.5 représente deux topologies de réseaux maillés : un réseau maillé 2D simple « *mesh* » (1.5(a)) et un réseau maillé 2D en tore (1.5(b)).



(a) liens directs (b) crossbar

(c) multi-étage (d) multi-bus

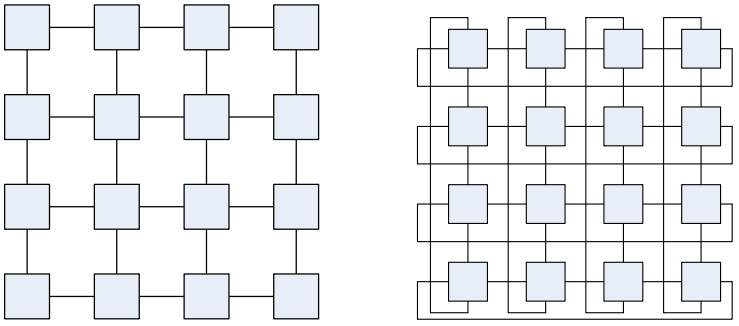
FIG. 1.4 – Différentes topologies de réseau d'interconnexion global.

	Crossbar	Multi-étage (Omega : N=M=2k)	Multi-bus
Nombre total de commutateurs	N * M	2 * N * k	B (N + M)
Nombre de bus	N	N	B
Nombre total de commutateurs par bus	M	2 * k	N + M

TAB. 1.1 – Comparaison de trois réseaux d'interconnexion globaux [119].

Les avantages des réseaux maillés résident principalement dans leur capacité à traiter des algorithmes travaillant sur des blocs de données comme c'est le cas en compression vidéo par exemple et dans leur capacité à être étendus sans un surcoût en surface.

Les principaux désavantages sont, d'une part, les communications qui ne sont pas faites dans le voisinage et qui sont coûteuses en temps et aussi en énergie car un grand nombre de commutateurs programmables doivent être traversés. D'autre part, ce type de réseau est difficilement exploitable du fait de leur non orthogonalité. Ainsi, l'étape de déploiement dans le flot de compilation (introduite plus loin en 1.3.3) se complexifie, devenant ainsi un point dur.



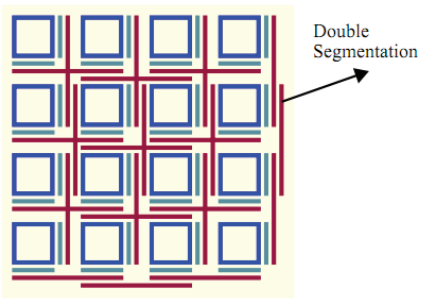
(a) mesh (b) tore

FIG. 1.5 – Différentes topologies de réseau d'interconnexion point-à-point.

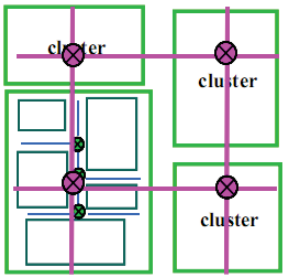
Les **réseaux hiérarchiques**, pouvant être globaux ou point-à-point, ont été proposés pour pallier aux inconvénients des deux types de réseaux précédents. Les éléments de l'architecture sont organisés en cluster de manière hiérarchique permettant plusieurs niveaux de communication, un interne au cluster et un autre entre clusters. La figure 1.6(a) montre un réseau hiérarchique homogène segmenté organisé en mesh et la figure 1.6(b) un réseau hiérarchique hétérogène en mesh généralisé. Le premier est utilisé dans les FPGA Xilinx par exemple, le second est plus couramment utilisé dans les SoC.

Ces réseaux sont particulièrement adaptés aux architectures homogènes et régulières car dans le cas hétérogène, le placement des modules, le positionnement des interfaces dans un cluster et le routage du réseau global deviennent des points durs. Une des solutions consiste à définir une architecture localement hétérogène mais globalement homogène ce qui permet de faciliter la conception d'un réseau hiérarchique efficace et facile à exploiter.

Remarque Une approche fait de plus en plus l'unanimité pour la conception de SoC, il s'agit du réseau sur puce ou NoC pour *Network on Chip*. Cette approche consiste à interconnecter les différents éléments d'une architecture (homogènes ou hétérogènes) par un réseau entre des routeurs, chaque élément ayant son propre routeur. Cette approche a de nombreux avantages comparé aux réseaux utilisant des bus : elle permet de concevoir des systèmes à large échelle, d'interconnecter plusieurs systèmes de manière hiérarchique, mais aussi d'interconnecter des modules asynchrones dans les systèmes globalement asynchrones et localement synchrones appelés GALS (Globalement Asynchrone Localement Synchrone). Cette solution est aujourd'hui la plus adaptée pour la réalisation d'un réseau d'interconnexion global au sein d'un SoC.



(a) Réseau hiérarchique homogène segmenté organisé en mesh



(b) réseau hiérarchique hétérogène en mesh généralisé

FIG. 1.6 – Différentes topologies de réseau hiérarchique.

1.3 Problématiques de conception d'une CGRA

La conception d'une CGRA se décompose en plusieurs étapes. Celles-ci sont représentées sur la figure 1.7 au sein d'un flot de conception proposé dans [108]. Ce flot est décomposé en deux grandes étapes, l'extraction des parties critiques qu'il est nécessaire d'accélérer sur la CGRA par une étape dite de prétraitement et une étape de génération d'architecture et de développement d'une méthodologie de déploiement d'une application sur la CGRA. A noter que la méthodologie de déploiement d'une application peut ne pas être développée dans le cadre d'un ACSS où toutes les applications à déployer sont connues lors de la phase de conception. Le déploiement est dans ce cas effectué lors de la génération de l'architecture, conjointement à celle des configurations nécessaires à l'exécution des applications.

1.3.1 Prétraitement

Durant la phase de prétraitement, des outils de profilage, statique et/ou dynamique, sont utilisés pour extraire les parties critiques des applications à accélérer (encore appelées noyaux de calcul ou *kernel* sur la figure 1.7). Ce sont les caractéristiques de ces parties qui détermineront principalement les choix de conception pour l'étape suivante de génération d'architecture.

1.3.2 Génération d'architecture

Lors de l'étape de génération d'une CGRA, les choix de conception concernent principalement son dimensionnement et la définition de ses principales caractéristiques :

- type et nombre de RPU,
- organisation des RPU,
- type de réseau d'interconnexion en fonction de l'organisation des RPU,
- nombre de mémoires locales et leur taille (optionnel).

1.3.2.1 Extraction de motifs de calcul

Le choix des motifs de calcul (expressions arithmétiques et logiques complexes) permettant d'accélérer les noyaux de calcul est une étape cruciale pour concevoir les RPU. De nombreux travaux ont été effectués sur ce sujet dont certains dans notre laboratoire

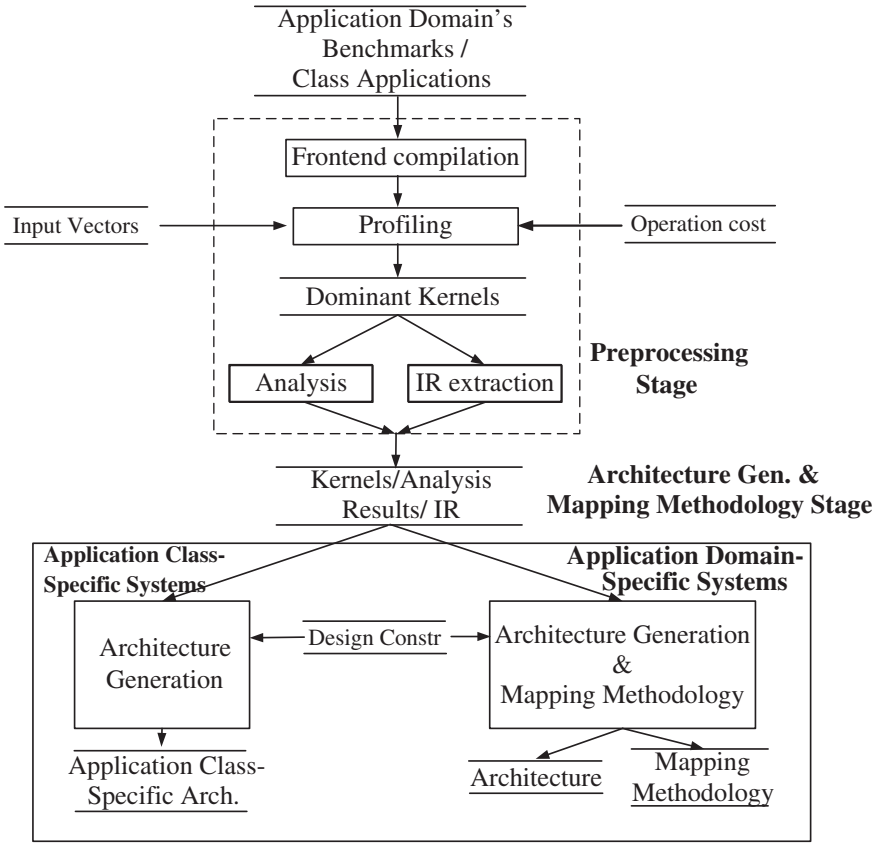


FIG. 1.7 – Flot de conception d'une CGRA, ADSS et ACSS [108].

au sein du système DURASE (*Generic Environment for Design and Utilization of Reconfigurable Application-Specific Processors Extensions*) un environnement générique pour la conception et l'utilisation d'extensions reconfigurables de processeurs spécifiques à une application [73]. Ce système est présenté plus loin (2.3.4.2). L'approche utilisée dans DURASE permet, grâce à la programmation par contraintes, d'extraire les motifs de calcul des parties critiques de l'application sous contraintes architecturales et technologiques. Les résultats montrent une forte couverture du graphe représentant les parties critiques de l'application avec peu de motifs, ce qui montre la qualité des motifs de calcul extraits en utilisant cette approche.

1.3.2.2 Synthèse et fusion de motifs de calcul

Une fois l'identification des motifs de calcul effectuée, ceux-ci sont synthétisés pour être implémentés en tant qu'unités spécialisées. Par exemple, dans le cas d'un ASIP dont le modèle contient une seule extension reconfigurable (RPU), ces motifs doivent-être synthétisés en une seule unité. Dans l'idéal, cette synthèse doit assurer le meilleur compromis surface-performance.

Il existe plusieurs méthodes pour « fusionner » des motifs de calcul et elles sont principalement issues de recherches sur les architectures reconfigurables. La majorité des méthodes utilise une modélisation des motifs de calcul sous forme de graphe acyclique orienté où chaque nœud représente une opération et chaque arc un transfert de données. Ainsi les algorithmes proposés sont logiquement issus de la théorie des graphes comme celui d'isomorphisme de sous-graphes utilisé dans [27] par exemple. Le problème d'isomorphisme de sous-graphes est connu pour être NP-complet [36] et donc les méthodes qui permettent de le résoudre sont de deux sortes : les méthodes exactes, utilisant l'ILP par exemple, et les méthodes basées sur des heuristiques comme dans [81] ; les premières pouvant permettre de prouver la pertinence des secondes.

Katherine Compton propose dans sa thèse [29] trois méthodes pour partager les liaisons entre unités de calcul et ainsi limiter l'ajout de multiplexeurs et de démultiplexeurs lors du routage des données. Les trois algorithmes proposés : un algorithme glouton, un bipartite et un basé sur la recherche d'une clique, utilisent des heuristiques basées sur la notion de similarité qui, dans ces travaux, est exprimée de deux manières différentes. Les résultats montrent que d'une manière générale, la méthode basée sur la clique est la meilleure.

D'autres approches ont été proposées. Par exemple Brisk dans [17] se base sur un algorithme permettant de trouver la plus grande sous-séquence commune entre deux (ou plus) séquences (ou chaines de caractères) pour transformer un ensemble d'instructions spécialisées en un unique chemin de données matériel capable d'exécuter toutes ces instructions. La solution proposée donne de bon résultat comparée à une synthèse par ILP qui n'exploite pas le partage de ressources avec 85,33% de surface en moins.

La contribution que nous avons jugée la plus pertinente pour débuter nos recherches sur ce sujet a été proposée par les chercheurs de Princeton dans le cadre du projet Mescal [3]. Ce projet propose un environnement de conception pour le développement d'architecture spécifique à une application [57]. Pour chaque partie critique de l'application, un chemin de données optimal est extrait. Ces chemins de données sont ensuite fusionnés (ou combinés) dans un seul chemin de données reconfigurable, une RPU. Le but de la fusion est de

partager au mieux les unités de calcul matérielles ainsi que les interconnexions pour minimiser la surface de la RPU. Suite à ces travaux, Moreano propose dans [81] une méthode quasi optimale pour la fusion de chemins de données mais sans maîtriser la performance de la RPU résultante. Cette lacune a été comblée par l'une des contributions présentées dans cette thèse.

1.3.3 Méthodologie de déploiement

La mise en place de la méthodologie de déploiement (ou *mapping*) représente l'une des étapes les plus difficiles à traiter dans le flot de conception et de compilation pour CGRA.

Comme nous l'avons dit dans l'introduction, l'étape d'ordonnancement, de placement et de routage est au cœur de l'outil de conception et de compilation pour CGRA. La figure 1.8 issue de [108] positionne cette étape au centre du flot de développement d'un ADSS. Durant cette étape, les trois principaux problèmes interdépendants que sont l'ordonnancement des opérations, l'assignement aux RPU et le routage des données sur le ou les réseaux d'interconnexion doivent être résolus.

A l'inverse des systèmes classiques, l'approche dans la conception de CGRA et de la chaine d'outils associés consiste à disposer d'une architecture simple et de repousser la complexité au niveau des outils. En effet, les architectures des processeurs généralistes récents de type superscalaire (comme les dernières versions de Pentium d'Intel ou de PowerPC d'IBM) sont très complexes, à tel point que certaines instructions machine ne sont pas exploitables ou atteignables par le compilateur ; ce qui est couramment le cas de l'instruction de décalage circulaire.

De plus, en comparaison d'une synthèse de haut niveau, les problèmes lors du déploiement d'un algorithme sur CGRA sont plus difficiles à résoudre, car dans notre cas les RPU sont connues ainsi que les propriétés du réseau d'interconnexion. La prise en compte de ces contraintes, associée à la forte corrélation des problèmes en font un axe de recherche privilégié.

De nombreuses approches ont été proposées pour résoudre le problème de déploiement d'une application, ou une partie, sur une CGRA. Toutes diffèrent du fait qu'elles ont été conçues pour une architecture particulière, c'est pourquoi nous les présenterons dans la section suivante en même temps que l'architecture, ou le type d'architecture, pour laquelle elles ont été développées.

1.4 Sélection de CGRA pour les applications multimédia

Parmi les différentes classifications proposées dans la littérature nous avons choisi celle de T. Cervero [24] concernant les architectures reconfigurables dynamiquement dédiées aux applications multimédia. Cette étude distingue trois grandes familles d'architecture en fonction du degré de couplage (cf 1.2.1) et du type d'unités de calcul : les matrices d'unités fonctionnelles (couplage intermédiaire), les architectures hybrides (coprocesseur) et les matrices de processeur (accélérateur). Le tableau 1.2 donne un récapitulatif des architectures reconfigurables pour les applications multimédia développées depuis 1996 jusqu'à 2009 et la figure 1.9, la classification issue de T. Cervero.

Nous présentons dans la suite de cette partie une architecture par classe. A ceci s'ajoute la présentation d'une architecture industrielle particulièrement intéressante dans notre cas

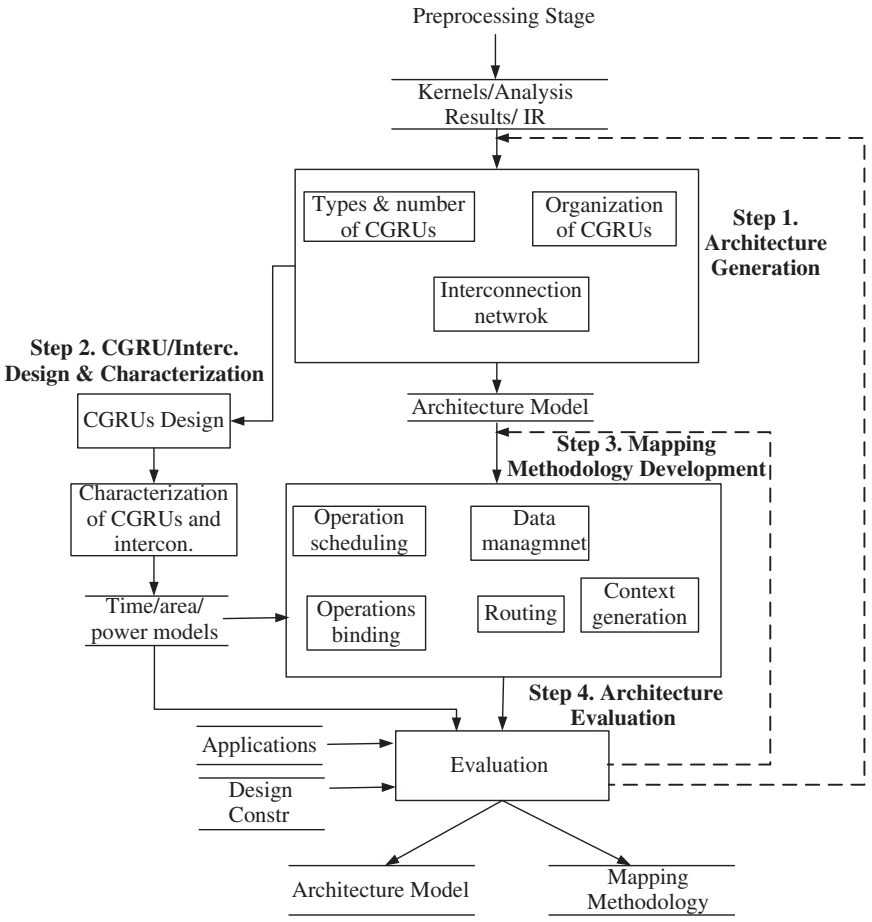


FIG. 1.8 – Génération d'architecture et développement d'une méthodologie de déploiement pour les ADSS [108].

de par sa généricité ainsi que sa méthode de spécialisation et de compilation.

Architecture	Année	Organisation	ref.
RaPiD	≈ 1996	University of Washington	[37]
MATRIX	≈ 1996	Tech. Cambridge	[78]
RAW	≈ 1997	MIT	[114]
PipeRench	≈ 1998	Carnegie Melon University	[20]
MorphoSys	≈ 1998	University of California Irvine	[102]
XPP	≈ 2000	PACT Corp.	[10]
DReAM	≈ 2000	University of Darmstadt	[11]
RAMP	≈ 2000	Texas Instrument Inc., Univ. of Texas	[92]
Ultrasonic	≈ 2002	Sonic, Imperial College	[55]
DAPDNA-2	≈ 2002	Fujitsu, IPFlex	[97]
DART	≈ 2002	University of Rennes	[87]
Montium	≈ 2002	University of Twente	[104]
DRP	≈ 2003	NEC	[110]
MOLEN	≈ 2004	Delf University of Technology	[113]
ADRES	≈ 2005	IMEC, KUL	[76]
XiRISC	≈ 2005	University of Bologna	[21]
3D-SoftChip	≈ 2005	Edith Cowan University	[64]
FLEXWAFE	≈ 2006	Technical University of Braunshweig	[70]
ECA	≈ 2006	Element CXI	[61]
QUKU	≈ 2006	University of Queensland	[101]
MORA	≈ 2007	University of Calabria	[25]
Butter	≈ 2007	Tampere University of Technology	[18]
SmartCell	≈ 2009	Worcester Polytechnic Institute	[69]

TAB. 1.2 – Recensement d'architectures reconfigurables pour les applications multimédia (adapté de [24] et complété).

1.4.1 DART

1.4.1.1 Architecture

DART [32] est une architecture hiérarchique reconfigurable dynamiquement dédiée aux applications mobiles, en particulier pour les applications de télécommunication 3G. La hiérarchie (figure 1.10) comporte trois niveaux : cluster (ou grappe), chemin de données reconfigurable ou DPR (*DataPath Reconfigurable*) et unités fonctionnelles. Le nombre d'éléments par niveau a été défini selon les besoins des applications visées. Cette architecture se classe parmi les matrices de processeurs.

Les clusters, au nombre de 4 sont indépendants, ce qui permet un contrôle et un modèle de programmation de l'architecture simplifiés. Chaque cluster contient 6 DPR interconnectés par un réseau segmenté, un module FPGA (« cœur de traitement dédié » sur la figure), un contrôleur gérant les ressources de calcul, un module dédié aux transferts de données de configuration entre une mémoire de configuration et le FPGA (module « DMA Ctrl. ») et une mémoire de données partagée entre le FPGA et les DPR. Ces derniers sont dédiés aux traitements arithmétiques et le FPGA aux traitements logiques.

Chaque DPR contient quatre unités fonctionnelles (2 multiplieurs/additionneurs et 2 unités arithmétiques et logiques (UAL)) optimisées en performance et en consommation énergétique. Le grain des DPR est de 16 bits mais grâce à des opérateurs de type SWP

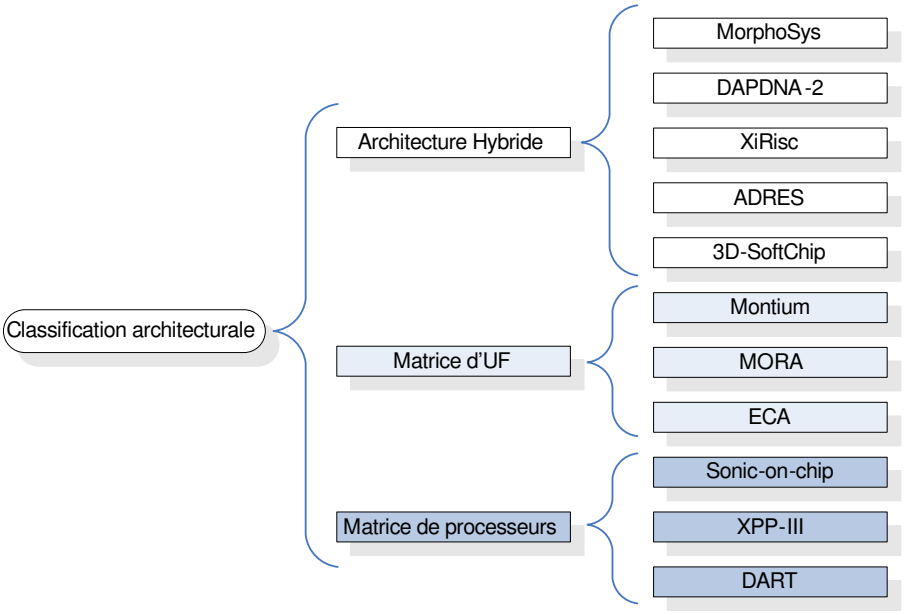


FIG. 1.9 – Classification des architectures reconfigurables pour les applications multimédia telle que proposée dans [24].

(*Sub-Word Processing*) ils sont capables de manipuler des données de 8 bits en parallèle, ce qui permet de doubler leur puissance de calcul.

1.4.1.2 Flot de compilation

Le flot de compilation pour l'architecture DART, donné figure 1.11, se base sur l'utilisation conjointe d'un *front-end* permettant la transformation et l'optimisation du code C en entrée, d'un compilateur reciblable et d'un outil de synthèse comportementale. La clé de l'approche dans ce flot de compilation est de traiter les différents types de traitement séparément. En effet, une étape de partitionnement permet de différencier les traitements réguliers, irréguliers et les manipulations de données, dont la compilation sera prise en charge respectivement par les outils cDART, gDART et ACG. Les codes générés sont simulables (module SCDA) ce qui permet une vérification et la collecte d'informations concernant la performance (nombre de cycles), l'utilisation des ressources ainsi que la consommation énergétique.

Cette approche avait déjà été utilisée dans les projets Pleiades [6] et GARP [54] qui distinguent aussi les traitements réguliers et irréguliers et les déploient sur la partie de l'architecture la mieux adaptée en utilisant l'outil de déploiement ou de compilation associé.

Le déploiement d'un graphe de flot de données sur l'architecture DART (gDART sur la figure 1.11) se fait en plusieurs étapes. D'une part, un ordonnancement au plus tôt est effectué (algorithme connu sous le nom d'ASAP pour *As Soon As Possible*). D'autre part, l'allocation mémoire permet de définir si une donnée peut résider en mémoire locale au DPR ou non. En fonction du résultat, un algorithme d'assignation différent est utilisé : soit une assignation directe simple, soit une assignation sous contrainte de placement mémoire dans le cas où les données sont présentes en mémoire locale.

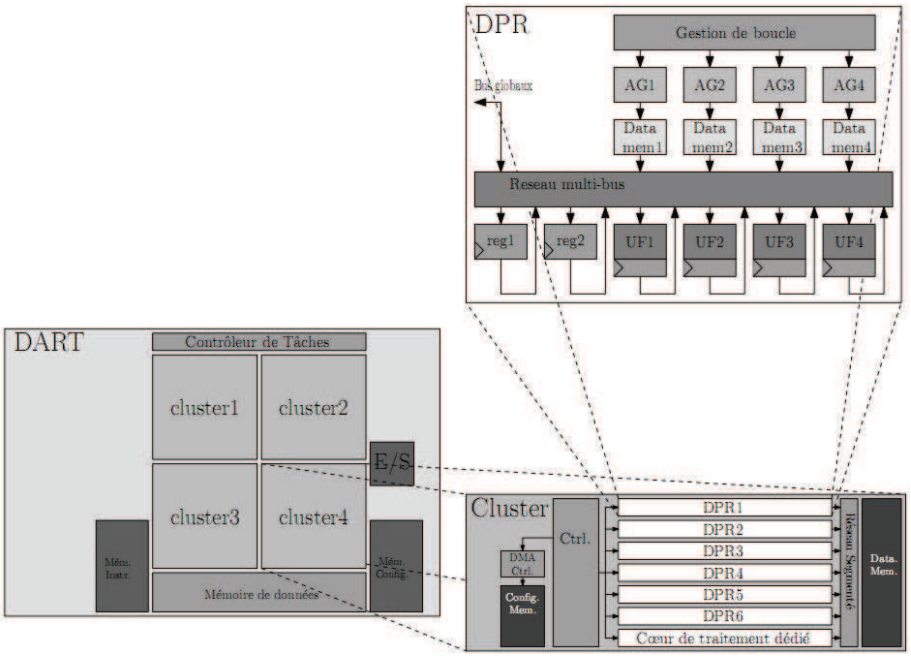


FIG. 1.10 – Différents niveaux hiérarchiques de l'architecture DART (adapté de [32]).

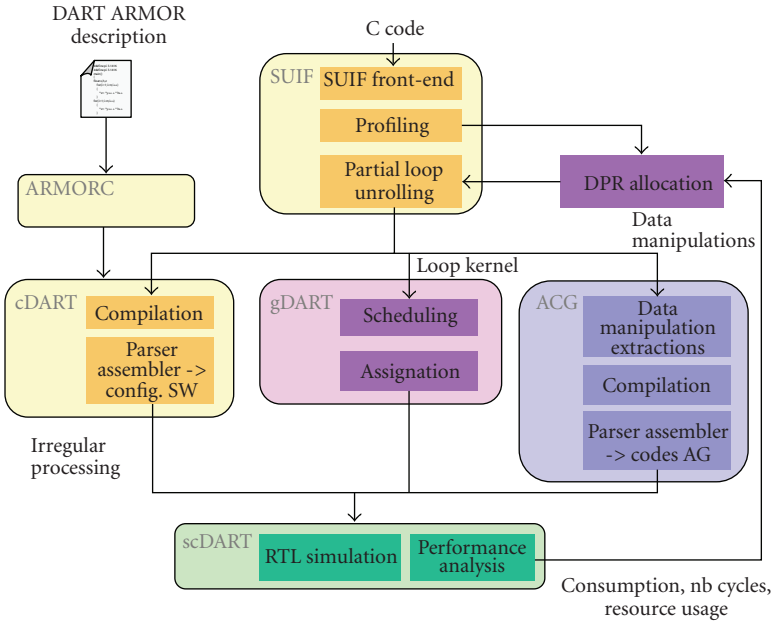


FIG. 1.11 – Flot de compilation pour l'architecture DART [87].

1.4.1.3 Performances et caractéristiques physiques

Les applications principalement visées par DART sont les applications intégrées dans les systèmes de télécommunication 3G (appelés UMTS pour *Universal Mobile Telecommunications System*) comme les traitements audio et vidéo ou encore le W-CDMA utilisé comme système de codage des transmissions.

Une première étude publiée en 2002 [33] donne une fréquence de fonctionnement de 130 MHz (technologie 0,18 μm, 1,95 V) qui permet de réaliser 260 millions d'opérations de multiplication avec accumulation (ou MAC pour *Multiply-ACcumulate*) par DPR ou 1,56 milliards d'opérations MAC par cluster sur des données de 16 bits ; ces chiffres sont doublés lorsque les données sont sur 8 bits comme pour les traitements vidéo. D'un point de vue instruction, DART délivre jusqu'à 520 millions d'instructions par seconde et par DPR ou 3,12 milliards d'instructions par seconde et par cluster. Comme une instruction inclut la génération d'adresse, les accès mémoire et jusqu'à deux opérations par multiplieur ou trois opérations par UAL ces chiffres se transforment en 10,9 milliards d'opérations 16 bits par seconde et par cluster ou 18,7 en 8 bits.

[33] présente aussi les résultats du portage de trois algorithmes clés dans le domaine ciblé : une DCT (*Discrete Cosine Transform*) sur des blocs 8x8 très utilisée en compression vidéo (MPEGx ou H.26x), une partie de la norme W-CDMA réalisant un désétalement complexe (*complex despreading*) et un algorithme d'autocorrélation que l'on trouve dans les codeurs de voix. Les résultats de ces expérimentations sont rassemblés dans le tableau 1.3. A noter qu'il est possible d'exécuter en parallèle les algorithmes de DCT et de désétalement complexe, ce qui amène à une performance globale de 4,9 millions d'opérations par seconde.

Application	DPR	operations	cycles	GOPS	inst. Read	data read	energy
Complex Despreading	2	2048	258	1,21	4	1032	435,8nJ
DCT 2-D	4	2048	72	3,69	5	256	64,7nJ
Autocorrelation	6	57600	1520	2,97	5	3040	3,15μJ

TAB. 1.3 – Performances de DART pour trois algorithmes multimedia [33].

Une seconde étude, faite en 2008 [87], présente des résultats et des comparaisons sur un algorithme de filtre à réponse impulsionnelle (FIR) et un algorithme issu de W-CDMA (*rake receiver*). Pour résumer, le FIR occupe 63% d'un cluster et l'efficacité énergétique atteinte est de 40 millions d'opérations par seconde et par milliwatt (MOPs/mW). Les opérateurs arithmétiques représentent alors plus de 80% de l'énergie consommée. Pour le deuxième algorithme, comparé à l'architecture XPP [10] proche de DART mais sans hiérarchie de mémoire ni d'interconnexion, la performance de DART est deux fois moins grande (au maximum) mais son efficacité énergétique est meilleure de 50%. Le tableau 1.4 donne les principales caractéristiques de DART.

1.4.2 Montium

Montium est une architecture reconfigurable hiérarchique intégrée comme tuile (ou TP pour *Tile Processor*) au sein du SoC chameleon [49] en tant que « cœur reconfigurable pour un domaine spécifique » ou DSRC (*Domain-Specific Reconfigurable Core*). Cette architecture est classifiée comme matrice d'unités fonctionnelles.

Technology	0,13 μm CMOS from ST Microelectronics
Supply voltage	1,2 V
Die size	2,68 x 8,3 mm^2
Clock frequency	200 MHz
Average total power	709 mW
Transistor count	2,4-million transistors
Computing performances	4800 MOPS on 32-bit data 9600 MOPS on 16-bit data

TAB. 1.4 – Caractéristique de DART [87] (version 2008).

1.4.2.1 Architecture

La tuile Montium, représentée sur la figure 1.12, ressemble à une architecture de type VLIW composée d'un ensemble de 5 UAL identiques utilisables de manière concurrente et un ensemble de 10 mémoires locales fournissant la bande passante nécessaire pour tirer partie de la puissance offerte par les UAL. Chaque UAL comporte aussi un ensemble de registres en entrée offrant un niveau hiérarchique de mémorisation supplémentaire permettant de supporter une localité importante, caractéristique clé d'une architecture basse consommation. Une partie calculatoire (PP pour *Processing Part*) est représentée par un segment vertical contenant une UAL, son banc de registres en entrée, deux mémoires locales et une partie du réseau d'interconnexion. Ce réseau est de type multi-bus, contenant 10 bus, reliant les PP au sein d'une tuile et pouvant être reconfigurés à chaque cycle. Un bus global est utilisé par l'unité de communication et de configuration (CCU pour *Communication and Configuration Unit*) pour accéder aux mémoires locales et gérer les flux de données assurant ainsi la liaison avec l'extérieur. Le séquenceur est responsable du contrôle de l'ensemble des parties calculatoires.

La granularité du chemin de données du Montium est de 16 bits et il supporte à la fois la représentation des nombres en entier et en virgule fixe. Chaque UAL comporte 4 entrées, chacune ayant son propre banc de registres permettant de stocker jusqu'à quatre opérandes, ainsi que 2 sorties de 16 bits. A noter que les bancs de registres ne sont pas contournables et que l'UAL ne contient pas de registre de pipeline. L'UAL du Montium, représentée figure 1.13 comporte deux niveaux, le premier traite les opérations arithmétiques et logiques classiques, sans la multiplication ni la division, et le second niveau contient une unité MAC bien utile pour les applications DSP notamment. Le second niveau d'une UAL comporte des canaux de communication de largeur 32 bits avec ses UAL voisines dans le sens est-ouest permettant d'accumuler le résultat du MAC du voisin à l'est au résultat du multiplieur, et cela sans introduire de délai. Les niveaux sont contournables au niveau fonctionnel.

1.4.2.2 Flot de compilation

Le flot de compilation de la tuile Montium [103], donné figure 1.14, permet la génération à partir d'une spécification en langage C des configurations pour la tuile. Après avoir vérifié que l'algorithme à accélérer n'est pas déjà présent dans la librairie, le code C est transformé en graphe de flot de données et de contrôle ou CDFG (*Control Data Flow Graph*). Cette représentation permet d'effectuer un certain nombre de traitements indépendants de l'architecture cible. Ensuite vient une phase de partitionnement ; les par-

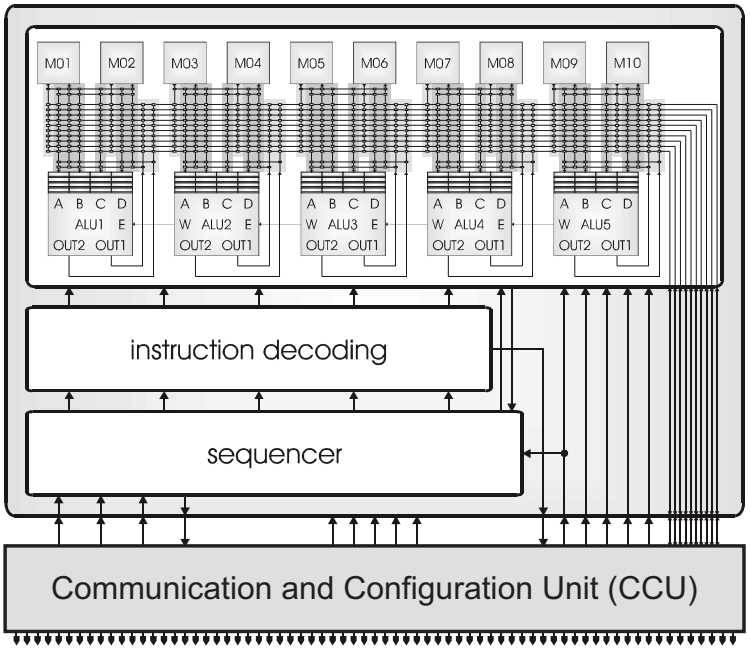


FIG. 1.12 – La tuile Montium [49].

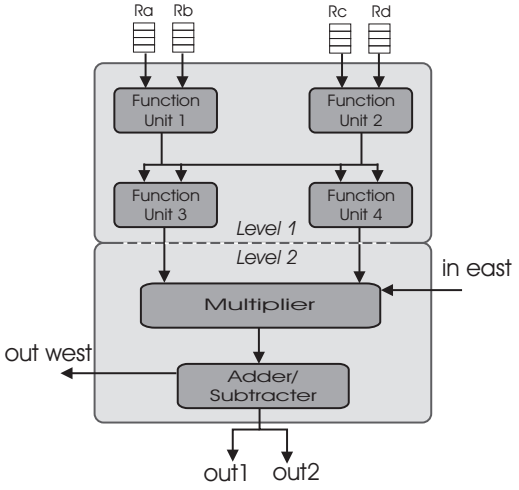


FIG. 1.13 – Unité Arithmétique et Logique de la tuile Montium [49].

titions obtenues sont ordonnancées et assignées aux 5 UAL de la tuile. La sortie de cette étape est un code lisible ressemblant à un assembleur (« MontionC ») qui est utilisé pour générer les configurations.

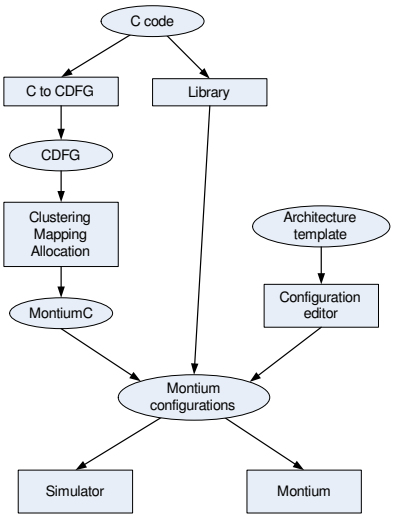


FIG. 1.14 – Flot de compilation pour la tuile Montium [103].

Le partitionnement (ou *clustering*) se fait par une méthode de couverture de graphe comportant une phase de génération de motifs de calculs et une autre de sélection de motifs pour couvrir le graphe représentant le chemin de données. Le but de cette phase est de couvrir le graphe efficacement avec le nombre minimum de motifs distincts et le nombre minimum d'appariements de ces motifs (dans le graphe). Pour ce faire, ils utilisent un algorithme itératif basé sur un graphe de conflit et la notion d'ensembles minimum indépendants notée MIS (*Minimum Independent Set*).

Trois algorithmes sont utilisés pour l'ordonnancement des partitions (*mapping*), problème défini sous le nom de *color-constrained scheduling*. Un algorithme de sélection de motifs sélectionne un ensemble non ordonné de motifs à partir du graphe en entrée, un algorithme de disposition (ou d'arrangement) de colonne minimise le nombre de configurations pour chaque UAL, et un algorithme d'ordonnancement *multi-pattern* (multi-motif) ordonnance le graphe en utilisant les motifs sélectionnés. Ces algorithmes sont des heuristiques basées sur des fonctions de priorité dont la définition et le raffinement représentent le principal axe amenant de bonnes performances.

Pendant la phase d'allocation, les variables sont allouées à des mémoires ou à des registres et les transmissions de données sont ordonnancées (nous utilisons plus communément le terme de routage). Cette étape est la dernière avant la génération des configurations pour le Montium. Cette étape complexe comporte de nombreuses contraintes et objectifs de minimisation. Il est (quasiment) impossible de les prendre en compte simultanément de par les détails de l'architecture c'est pourquoi elles sont considérées séparément : à chaque étape est considérée une seule optimisation. Dans le but de minimiser les effets négatifs causés par cette approche, à chaque étape est pris en compte les besoins des étapes suivantes.

Pour plus de détails, le lecteur peut se référer à la thèse de Guo [49].

1.4.2.3 Performances et caractéristiques physiques

Parmi les différents algorithmes portés sur la tuile Montium, l'implémentation de la 2-D IDCT (*Inverse DCT*) 8x8 [105], a retenu notre attention pour les mêmes raisons que pour DART. Trois cibles architecturales différentes permettent de comparer les résultats obtenus : un ASIC, un DSP Texas Instruments TMS320C6454 et un processeur RISC embarqué ARM 946E-S doté d'une extension optimisée pour l'opération ASIC. L'ensemble des résultats en termes d'efficacité énergétique et surfacique sont donnés par le tableau 1.5 et les graphiques associés par la figure 1.15. On peut facilement en déduire que Montium est une architecture plus efficace qu'un DSP, beaucoup plus efficace qu'un processeur embarqué même s'il est doté d'une unité spécialisée pour effectuer l'opération de base de l'IDCT (opération ASIC). Enfin, Montium rivalise avec un ASIC au niveau énergétique. Quant à la surface, évidemment plus faible pour l'ASIC, elle peut être utilisée pour d'autres applications dans le cas de Montium, on peut voir cela comme le prix de la flexibilité. En plus, le Montium offre pour l'exemple de l'IDCT des coûts plus faibles et une mise sur le marché (*time to market*) plus rapide que l'ASIC qui, rappelons le, est long et coûteux à développer.

	ASIC	Montium	TI	Arm
max freq. (MHz)	154	100	720	200
power @ max freq. (mW)	634,5	50	1967	92
technology (tm)	0,18	0,13	0,09	0,13
Voltage (V)	1,8	1,2	1,2	1,2
area (mm^2)	12,17	2,4	n/a	1,86
cycles/2-D 8x8 IDCT	30	96	495/6	2796
energy/clock cycle @ 0,13μm (nJ), 1,2V	1,34	0,5	3,95	0,46
area in 0,13μm (mm^2)	6,1	2,4	≈79	1,86
max. freq. 0,13μm (MHz)	216	100	514	200
# 2-D IDCT/s in 0,13μm	7,2M	1,0M	6,2M	0,072M
energy/2-D IDCT (nJ)	40	48	325	1286
# 2-D $IDCT/mm^2/s$	1182k	434	≈79k	38k

TAB. 1.5 – Comparaison des performances entre ASIC, Montium, TI et Arm pour l'algorithme IDCT 8x8 [105].

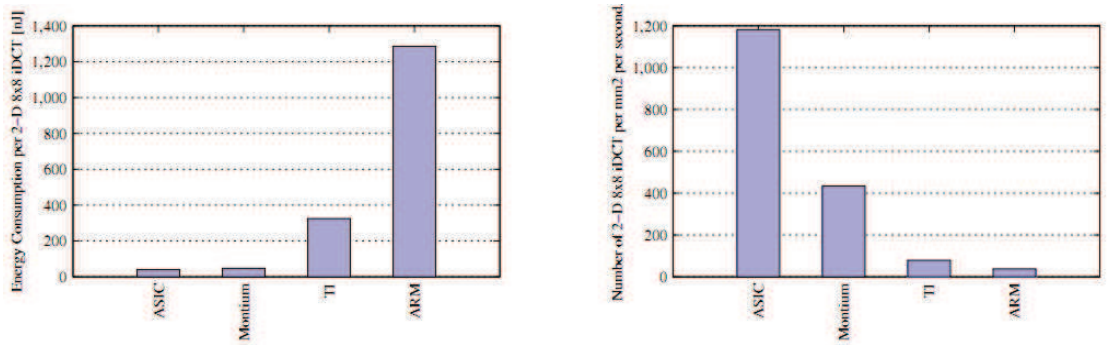


FIG. 1.15 – Consommation par 2-D IDCT 8x8 (nJ) et nombre de 2-D IDCT 8x8 $/mm^2/s$ [105].

1.4.3 ADRES

ADRES (*Architecture for Dynamically Reconfigurable Embedded Systems*) est une architecture hybride pour les systèmes embarqués reconfigurables dynamiquement ciblant les applications multimédia et plus généralement celles basées sur des boucles.

1.4.3.1 Architecture

ADRES se définit comme un modèle reconfigurable composé d'un processeur VLIW et d'une matrice reconfigurable à gros grain (figure 1.16). Cette dernière a un accès direct aux bancs de registres, aux caches et aux mémoires du système ce qui amène plusieurs avantages comme l'augmentation des performances, un modèle de programmation simplifié, un coût de communication réduit et un partage des ressources. Le couplage d'un processeur VLIW et d'une matrice reconfigurable permet de tirer au maximum partie du parallélisme de l'application. A la différence des autres approches, le processeur VLIW remplace le traditionnel cœur de processeur séquentiel pour l'exécution de la partie orientée contrôle ce qui permet d'exploiter le parallélisme au niveau instruction. Le parallélisme offert par la matrice reconfigurable est utilisé pour accélérer les parties critiques de type flot de données. Des liaisons de communications globales et locales permettent les transferts entre les cellules reconfigurables notées RC (*Reconfigurable Cell*) de la matrice. Les communications entre la partie VLIW et la matrice se font par l'intermédiaire de la file de registres et de la mémoire toutes deux partagées.

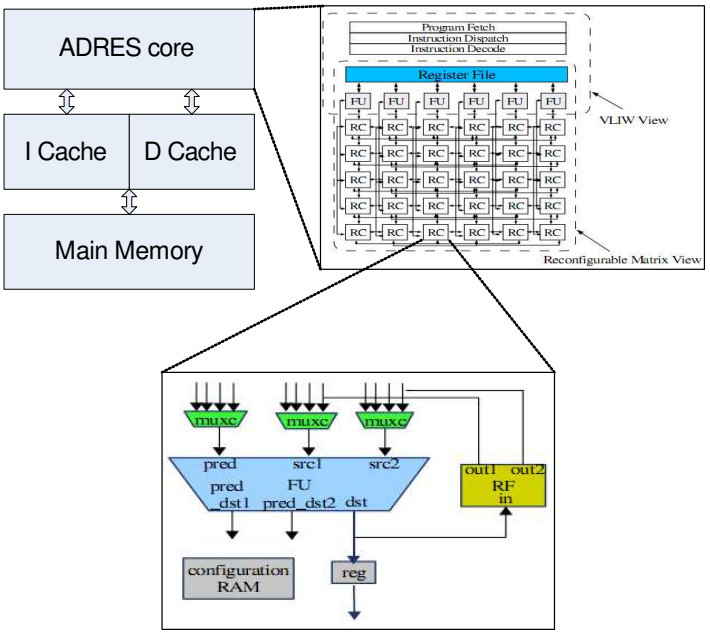


FIG. 1.16 – Architecture ADRES [76].

1.4.3.2 Flot de compilation

Le flot de compilation pour ADRES est donné figure 1.17 [75]. On y retrouve la phase de profilage et de partitionnement permettant d'extraire les parties critiques accélérables, puis leur code source est transformé pour en exacerber le parallélisme (*function inlining*, déroulage de boucle). Ensuite vient la phase de transformation de la spécification en une représentation intermédiaire (ici Lcode). Le flot se sépare alors en deux parties, une pour le VLIW, l'autre pour la matrice reconfigurable.

Concernant la partie reconfigurable, la clé de l'approche réside dans l'utilisation d'une technique de *modulo scheduling* adaptée au CGRA. Le *modulo scheduling* [93] est une

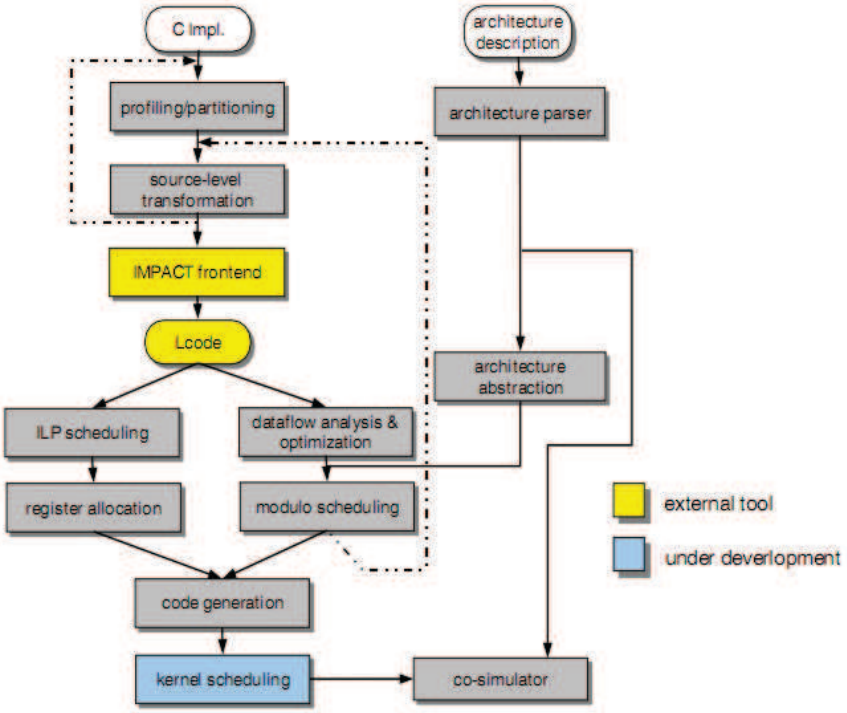


FIG. 1.17 – Flot de compilation pour ADRES [75].

technique bien connue permettant de paralléliser l'exécution d'un boucle en exécutant différentes itérations indépendantes en parallèle. Cette technique se complexifie dans le cas de CGRA car elle correspond alors à une combinaison de P&R et de contrainte modulo dans un espace 3D. Les chercheurs ont proposé à ce titre une représentation sous forme de graphe nommée MRRC (*Modulo Routing Resource Graph*) représentant une abstraction de l'architecture générée à partir d'une description de l'architecture en XML (*eXtensible Markup Language*). Cette représentation combine une table de réservation modulo notée MRT (*Modulo Reservation Table*) pour le *pipelining* logiciel, venant de la compilation pour VLIW, et un graphe de routage des ressources utilisé généralement pour le P&R des FPGA.

1.4.3.3 Performances

Il existe de nombreuses configurations ou instances du modèle d'architecture. La description de l'architecture concerne notamment la largeur des données traitées, le nombre d'unités fonctionnelles en précisant celles qui disposent d'une unité de chargement et de déchargement, le nombre de bancs de registres et enfin la topologie du réseau inter-RC. Cette grande flexibilité est intéressante pour explorer l'espace de conception comme dans [14] où 15 instances différentes d'ADRES (en jouant sur 7 paramètres différents) sont comparées, dont une modélisant un processeur RISC scalaire non pipeliné. Les algorithmes utilisés pour cette étude sont une FFT (*Fast Fourrier Transform*) et une IDCT. Les auteurs déterminent ainsi parmi toutes les instances étudiées, l'architecture ayant la consommation d'énergie la plus faible avec des performances raisonnables. Les résultats montrent une performance de 10,35 - 17,51 MIPS/mW, une puissance de 73,28 - 80,45 mW et une consommation d'énergie (la plus faible) de 0,619 - 37,72 μJ pour la FFT et l'IDCT respectivement. Tous les résultats détaillés sont donnés dans l'article. Une étude similaire mais plus récente [15] à permis d'améliorer ces résultats en définissant une instance d'ADRES ayant une efficacité de 60 MOPS/mW contre 40 MOPS/mW pour la précédente.

D'autres études montrent l'efficacité de l'approche utilisant l'architecture ADRES et la chaine d'outils DRESC dont celle publiée dans l'article [74] qui présente le processus et les résultats du déploiement de l'application de décompression vidéo H.264/AVC sur une instance d'ADRES formée d'une matrice 8x8 de 64 unités fonctionnelles incluant 16 multiplieurs, 40 UAL et 8 unités de chargement/déchargement ainsi qu'une topologie et une hiérarchie mémoire adaptées. En tout, 16 noyaux de calcul ont été accélérés grâce à la matrice reconfigurable. L'accélération obtenue, par rapport à une processeur VLIW 8 voies, est de 4,2 pour les noyaux de calcul et de 1,9 pour l'application entière. Une flux vidéo de format CIF[3] peut-être ainsi décompressé en temps réel à une fréquence de fonctionnement du circuit de seulement 100 MHz.

1.4.4 Silicon Hive

Silicon Hive est le nom d'une entreprise néerlandaise, start-up issue de Philips Netherlands, basée à Eindhoven. Les produits proposés par cette société sont des processeurs et accélérateurs spécifiques optimisés pour une application (ou un champ d'applications) et surtout programmables en C. Le domaine d'applications visé est celui des systèmes multimédia embarqués modernes. Silicon Hive propose différentes gammes de processeurs « HiveFlex » : une pour le traitement d'image basse consommation (ISP - *Image Signal Processing*), une pour les applications de télécommunication (CSP - *Communication Signal Processor*) et une pour la traitement vidéo HD (VSP - *Video Signal Processing*).

[3] *Common Intermediate Format* dont la résolution des images est de 360x288.

Dernièrement, un nouveau produit a été ajouté au catalogue, il s'agit de la plateforme de traitement configurable « HiveLogic » permettant de définir une plateforme efficace au niveau énergétique et surfacique à l'aide d'une bibliothèque de processeurs paramétrables par des blocs prédéfinis.

1.4.4.1 Architecture

Le modèle d'architecture des processeurs de Silicon Hive est le même pour tous les produits, il est représenté figure 1.18. Ce modèle d'architecture consiste en deux sous-systèmes : le cœur (*core* sur la figure) qui comporte un chemin de données VLIW, piloté par un simple séquenceur, et un module intégrant la mémoire et les interfaces d'entrée/sortie (*coreio*). L'une des idées exploitées ici est que ce modèle dispose d'un bloc de contrôle matériel le plus petit possible. Cette idée est une des bases dans la conception des architectures reconfigurables où la gestion du contrôle est principalement au niveau de la chaine d'outils.

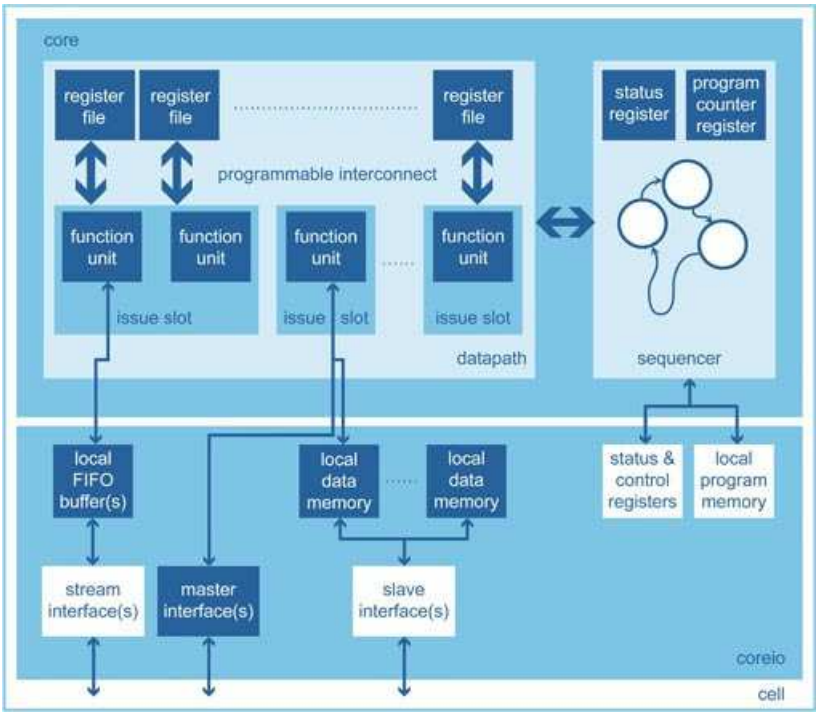


FIG. 1.18 – Modèle d'architecture générique Silicon Hive [5].

Le chemin de données VLIW, représenté sur la figure 1.19, consiste en plusieurs clusters (ou *issue slot*) comportant une ou plusieurs unités fonctionnelles, des bancs de registres et un réseau d'interconnexion programmable. Chaque cluster comporte sa propre unité de chargement/déchargement. Le passage à l'échelle est un point fort de ce modèle car l'architecture peut contenir n'importe quel nombre de clusters, de bancs de registres et d'unités fonctionnelles. Les processeurs peuvent contenir tellement d'unités en parallèle que l'on ne parle plus de VLIW mais d'ULIW signifiant *Ultra-Long Instruction Word*. Les unités fonctionnelles peuvent-être vectorielles ou SIMD (*Single Instruction on Multiple Data*) proposant ainsi un parallélisme élevé. De plus, les clusters sont spécialisables grâce à une bibliothèque qui fournit des clusters pour le traitement du signal, pour le traitement vidéo, les systèmes de télécommunication, etc. et des unités fonctionnelles spécialisées (MAC, dé-

calage, etc.).

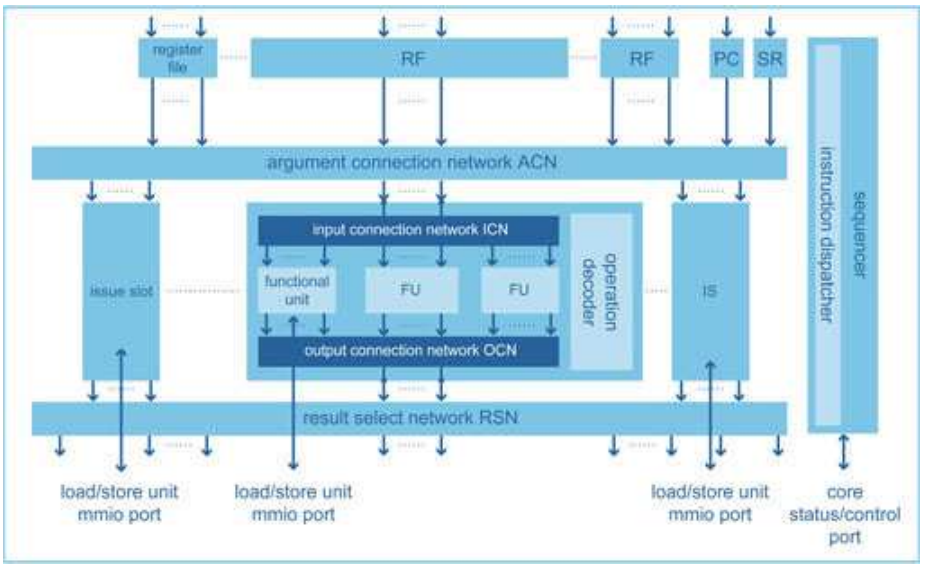


FIG. 1.19 – Modèle du chemin de données VLIW des processeurs Silicon Hive [5].

Pour plus de détails sur une architecture en particulier, se référer au site internet [5] et aux papiers [31, 112][4] pour le processeur HiveFlex ISP2100, [88] pour la série HiveFlex VSP2000 ou encore [51] pour les processeurs plus ancien (AVISPA) mais néanmoins représentatifs de l'approche utilisée.

1.4.4.2 Flot de compilation

Le modèle d'architecture proposé par Silicon Hive présente un fort degré de parallélisme avec peu de contrôle matériel. Pour pouvoir exploiter efficacement cette architecture il est nécessaire de disposer d'une chaine d'outils capables de placer de nombreuses opérations en parallèle dans le modèle ULIW. De plus, la grande flexibilité de l'approche qui consiste à proposer un modèle spécialisable et dimensionnable à souhait doit être supportée par la chaine d'outils.

La chaine d'outils de conception et de compilation de Silicon Hive, nommée HiveCC, est représentée figure 1.20 [51]. Le système est basé sur un langage propriétaire de description matérielle appelé TIM (*The Incredible Machine*) décrivant le paramétrage du modèle : nombre d'unités fonctionnelles, de banc de registres, etc. De cette simple description sont déduits des paramètres plus fins comme par exemple la largeur d'une instruction ULIW. Cette description permet d'instancier totalement une architecture en invoquant des blocs prédéfinis en VHDL ou Verilog. Mais la description TIM permet aussi la génération de la chaine de compilation et des outils associés : assembleur, éditeur de liens, compilateur C, simulateur de jeu d'instructions, et simulateur. Cela permet à l'entreprise de générer une implémentation optimisée de son architecture ULIW sous forme de code HDL synthétisable en quelques heures voire quelques jours selon la complexité demandée. A noter qu'à l'origine ce n'est pas le client ou l'utilisateur qui procède à cette étape. Ce flot utilise

[4]La première référence est un papier publié en 2006 dans une conférence, la seconde un « whitepaper » plus détaillé publié en 2007.

différents niveaux de simulation et de vérification permettant d'extraire des informations utilisées pour optimiser finement le *design* en modifiant sa microarchitecture et son jeu d'instructions. A noter aussi que le code compilé par la chaine d'outils générée n'est pas portable du fait que cette dernière ne cible qu'une instance particulière de l'architecture.

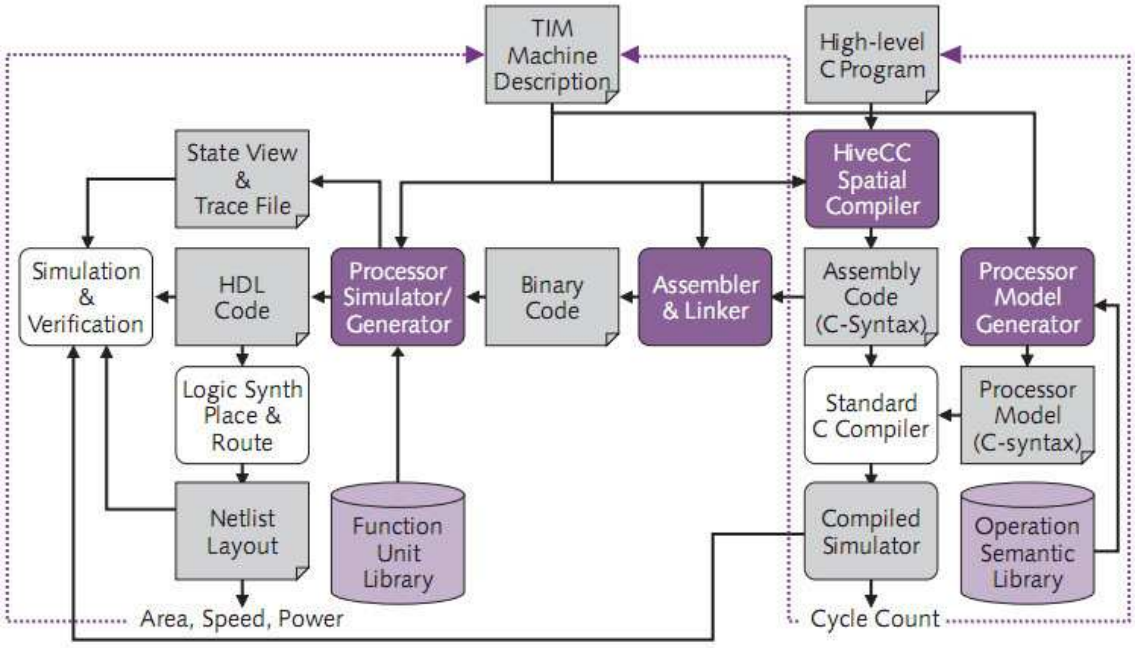


FIG. 1.20 – Flot de conception et de compilation Silicon Hive [51].

Le flot de compilation HiveCC comprend un compilateur spatial décrit dans [19] et représenté par le figure 1.21. Cet outil est particulièrement intéressant de par ses fonctionnalités et capacités. En effet, il utilise des techniques d'ordonnancement basées sur la résolution de contraintes déterministes qui sont capables de traiter un parallélisme massif, plus d'une centaine de bancs de registres et un réseau d'interconnexion partiel.

L'approche utilisée pour l'ordonnancement est représentative des deux tendances présentées précédemment (cf. 1.3.3). Ainsi, deux types de résolutions sont proposées selon le niveau d'optimisation souhaité : une basée sur la technique de *list scheduling* avec *pipelining* logiciel fournissant une solution rapidement mais pas optimale, et une autre dite « *Manifold* », qui contrairement à la première fournit un résultat moins rapidement mais garanti optimal. HiveCC est présenté comme le premier compilateur commercialisé basé sur le principe d'ordonnancement d'instructions sous contraintes architecturales [13].

1.4.4.3 Performances

Peu de chiffres sur les performances de ces architectures sont disponibles, cela est peut-être dû au fait qu'elles dépendent des paramètres utilisés pour instancier telle ou telle architecture. Néanmoins, quelques données commerciales sont fournies sur le site internet [5] : une architecture de type VSP2500 supportant la norme H.264 (*High Profile*) est capable, en temps réel, de décoder une vidéo au format 720p et d'encoder en format 576p avec une instance comportant 1 tuile de traitement de flux (*SP - Stream Processing*

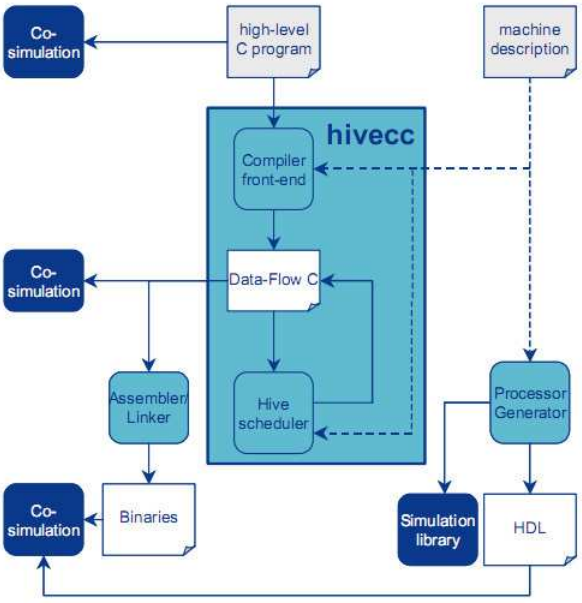


FIG. 1.21 – Compilateur spatial HiveCC [19].

tile) et une tuile de traitement vectoriel (*VP - Vector Processing tile*). Avec une tuile SP et trois tuiles VP, des résultats préliminaires promettent un encodage et décodage en haute définition (1080p) à 30 images/sec.

1.5 Synthèse

La conception et la programmation d'une architecture reconfigurable dynamiquement à gros grain sont des tâches complexes. L'approche principale est de concevoir, d'une part un modèle d'architecture simple, générique, et doté d'un contrôle matériel minimaliste, et d'autre part, de concevoir une chaine d'outils complexes permettant d'explorer l'espace de conception mais surtout d'exploiter au maximum, si possible de manière optimale, les propriétés de l'instance du modèle d'architecture optimisée pour un domaine, une classe d'applications ou pour une seule application.

Plusieurs challenges se présentent au niveau de la chaine d'outils qui concentre ainsi une grande partie de la complexité.

Le premier, indispensable, consiste à définir une instance spécialisée, optimisée de l'architecture. Pour cela, les caractéristiques des applications ciblées sont analysées puis les parties critiques sont extraites et utilisées pour dimensionner et définir les caractéristiques spécifiques de l'architecture.

Ensuite, si l'architecture cible un spectre applicatif large et non totalement défini, il est proposé un compilateur permettant de déployer une nouvelle application (ou partie) sur une instance de l'architecture. Ce dernier doit être flexible et efficace à la fois (comme le système global !). Deux tendances sont observées, la première est de fournir rapidement un résultat de bonne qualité, la seconde est de fournir le plus vite possible un résultat optimal.

Globalement, la recherche d'adéquation et de compromis, est au centre des problématiques de conception et de programmation d'une CGRA. L'architecture est optimisée pour

un domaine applicatif dont le spectre varie entre l'unique application et le vaste domaine comme les applications multimédia ; on parle d'adéquation architecture ⇆ application. Mais étant donné que le modèle d'architecture est conservé simple assurant efficacité et flexibilité à la fois, une nouvelle ère a commencé concernant les outils utilisés pour la conception, la spécialisation et la programmation de CGRA. Ces outils font face à des problèmes reconnus comme difficiles à résoudre, c'est pourquoi ils utilisent des techniques adaptées au modèle d'architecture pour lequel ils sont conçus. On peut parler alors d'adéquation Architecture ⇆ Flot de conception et de compilation.

Il en résulte un triangle représentatif des problématiques de conception et de compilation d'une architecture matérielle embarquée (représenté sur la figure 1.22). Ce triangle représente les adéquations entre un modèle d'architecture, un domaine applicatif et un flot de conception et de compilation. Cette représentation n'est pas nouvelle, on la trouve en 1997 dans le mémoire de thèse de Gwendal Le Fol [43] dont les travaux ont été menés dans le laboratoire « Architectures Parallèles Intégrées » de l'IRISA, d'ailleurs certains membres font maintenant partie de l'équipe CAIRN de l'IRISA. A l'époque, la problématique impliquait le langage de programmation qui dans nos travaux actuels est imposé, ce qui repousse le problème au niveau de la chaine d'outils.

Les travaux présentés dans cette thèse s'inscrivent aussi dans cette problématique dans laquelle sont visées les applications multimédia, les CGRA et un flot de conception et de compilation basé sur la méthodologie de programmation par contraintes, notée CP pour *Constraint Programming* et présentée dans le chapitre suivant.

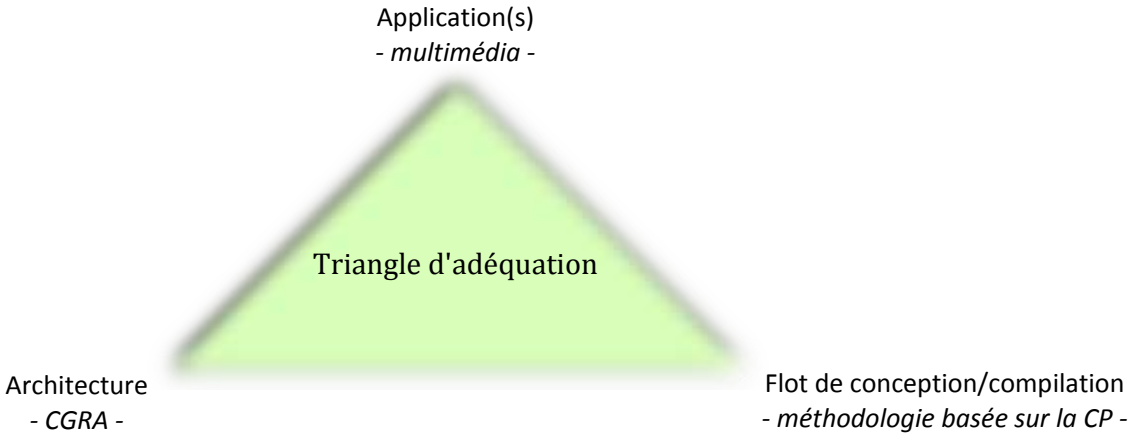


FIG. 1.22 – Triangle représentatif des problématiques de conception et de compilation d'une architecture matérielle embarquée, on parle d'adéquation application(s) / architecture / Flot de conception et de compilation.

Chapitre 2

La programmation par contraintes : une nouvelle approche pour la conception et la compilation pour architecture reconfigurable à gros grain

La programmation par contraintes (CP pour *Constraint Programming*) est une technique de résolution de problèmes dont les débuts datent des années 1970 et qui est issue de l'intelligence artificielle. Cette technique est particulièrement utilisée pour tenter de résoudre des problèmes combinatoires complexes venant d'applications réelles comme la planification et la gestion, le transport, la production industrielle etc. [95]. Il existe aujourd'hui de nombreux « solveurs » de contraintes tels que, pour les logiciels propriétaires, ILOG CP (bibliothèques C++, Java, .Net), Comet (langage orienté objet dédié à la résolution de problèmes de contraintes, gratuit pour usage académique) ou encore Artelys Kalis (C++, bibliothèques Java), pour les logiciels libres, Choco (bibliothèques Java), Gecode (bibliothèque C++) et surtout JaCoP [2] qui a déjà été utilisé par l'équipe CAIRN à l'occasion de travaux précédents.

Une citation largement utilisée pour introduire la CP est donnée ci-après :

« *Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it.* » [44]

La suite de ce chapitre présente dans un premier temps les bases de la programmation par contraintes, ensuite il est discuté de son utilisation pour la conception et la compilation pour des architectures embarquées. Enfin sont présentés deux systèmes, développés dans notre équipe de recherche, basés sur l'approche de programmation par contraintes et exploitant notamment les contributions proposées dans cette thèse.

2.1 Bases de la programmation par contraintes

2.1.1 Problème de satisfaction de contraintes

Définition 2.1 : *Un problème de satisfaction de contraintes, ou CSP pour Constraint Satisfaction Problem, est défini par un triplet* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *représentant une ensemble de variables* $\mathcal{X} = \{x_1, ..., x_n\}$*, de domaines* $\mathcal{D} = \{D(x_1), ..., D(x_n)\}$ *et de contraintes* $\mathcal{C} = \{C_1, ..., C_k)\}$*.*

Définition 2.2 : *Une* ***variable*** x_i *dans un CSP est associée à un* ***domaine*** $D(x_i)$ *qui définit l'ensemble des valeurs qu'il est possible d'affecter à cette variable. On distingue les variables booléennes dont le domaine se réduit à* $\{0, 1\}$*.*

Définition 2.3 : *Un* ***domaine*** $D(x_i)$ *représentant l'ensemble des valeurs pouvant être assignées à une variable* x_i *est généralement un ensemble fini et discret d'entiers. Un domaine peut être formé de plusieurs sous-domaines disjoints.*

Remarque : Dans notre utilisation de la CP, le domaine d'une variable est fini et est formé de nombres entiers.

Définition 2.4 : *Un tuple ou affectation est une liste de valeurs choisies pour chacune des variables du problème. Un tuple est dit consistant si il ne viole aucune contrainte. Une affectation est dite totale si elle instancie toutes les variables du problème et partielle si elle n'en instancie qu'une partie.*

Définition 2.5 : *Une solution à un CSP est un tuple qui est consistant sur l'ensemble des variables. Un problème peut-être sous-contraint (lorsqu'il existe plusieurs solutions possibles) ou sur-contraint (si il n'y a pas de solution).*

2.1.2 Contraintes

Les contraintes expriment les relations sur les variables du problème. Si une contrainte porte sur une variable, on dit qu'elle est unaire, sur deux variables elle est binaire et sur n variables elle est n-aire.

On peut distinguer trois grands types de contraintes : les contraintes primitives, logiques et conditionnelles ainsi que les contraintes globales.

2.1.2.1 Contraintes primitives

Par contrainte primitive, on entend celles exprimées à l'aide des opérations arithmétiques traditionnelles que sont l'addition, la soustraction, la multiplication et la division et les opérateurs relationnels égal, différent, inférieur, inférieur ou égal, supérieur et supérieur ou égal. Par exemple, $x_1 + 3 * x_2 \leq 10$.

Ce type de contrainte est utilisé comme argument des contraintes logiques et conditionnelles.

2.1.2.2 Contraintes logiques et conditionnelles

Les contraintes logiques sont exprimées à l'aide des opérateurs logiques non, ou, et, xor (par exemple $c_1 \wedge c_2 \vee c_3$) ainsi qu'avec la relation d'équivalence ($c_1 \Leftrightarrow c_2$) et de contenance ($x_i \subset D_j$).

Les contraintes conditionnelles sont du type **si** c_1 **alors** c_2 **sinon** $c3$.

A noter que le résultat d'une contrainte logique ou conditionnelle peut être affecté à une variable booléenne.

2.1.2.3 Contraintes globales

Il existe de nombreuses contraintes globales proposées par les solveurs. Nous nous limitons dans ce manuscrit à la présentation des contraintes les plus utilisées dans nos modèles en excluant les contraintes spécifiques développées pour nos propres besoins qui seront introduites par la suite.

Alldifferent Cette simple contrainte globale assure que toutes les variables ont une valeur différente.

Cumulative Cette contrainte assure pour un ensemble de rectangles, chacun défini dans un espace bidimensionnel par trois variables représentant son origine, sa longueur et sa hauteur, que la somme des hauteurs ne dépasse, à aucun moment, la limite imposée. Formellement cela se traduit par :

$$Rect_i = [start_i, duration_i, height_i]$$

$$\forall t \in [\min_{1 \leq i \leq n}(start_i), \max_{1 \leq i \leq n}(start_i + duration_i)] : \sum_{k:start_k \leq t \leq start_k + duration_k} height_k \leq Limite$$

Dans cette formule, les termes min et max représentent respectivement les valeurs minimum et maximum des variables. Cette contrainte impose qu'à tout instant t entre le début du première rectangle ($\min start_i$) et la fin du dernier rectangle ($\max(start_i + duration_i)$) l'accumulation des hauteurs des rectangles ne dépasse pas la limite « ($Limite$) » imposée.

La figure 2.1 représente graphiquement l'utilisation de cette contrainte pour l'ordonnancement de 5 tâches. Chaque tâche est caractérisée par une durée et un nombre de resssources nécessaires à son exécution. La contrainte cumulative impose que le nombre de ressources utilisées par les tâches ne dépasse, à aucun moment, la limite (ici égale à 8).

Cette contrainte globale est pratique pour modéliser l'utilisation d'un ensemble limité de ressources. Par exemple, nous avons utilisé cette contrainte pour modéliser l'occupation des cellules au sein d'une mémoire d'un processeur. Un rectangle $Rect_i$ modélise alors l'occupation d'une cellule mémoire ($height_i$ vaut donc 1) à partir d'un instant $start_i$ pendant une durée $duration_i$. La capacité d'une mémoire en termes de cellules représente la variable « $Limite$ ».

Diff2 La contrainte différentielle sur deux dimensions $Diff2$ impose pour un ensemble de rectangles défini dans un espace bidimensionnel, qu'il n'existe aucun recouvrement entre eux dans cet espace, c.-à-d. qu'aucun rectangle ne se chevauche. Comme le montre la figure

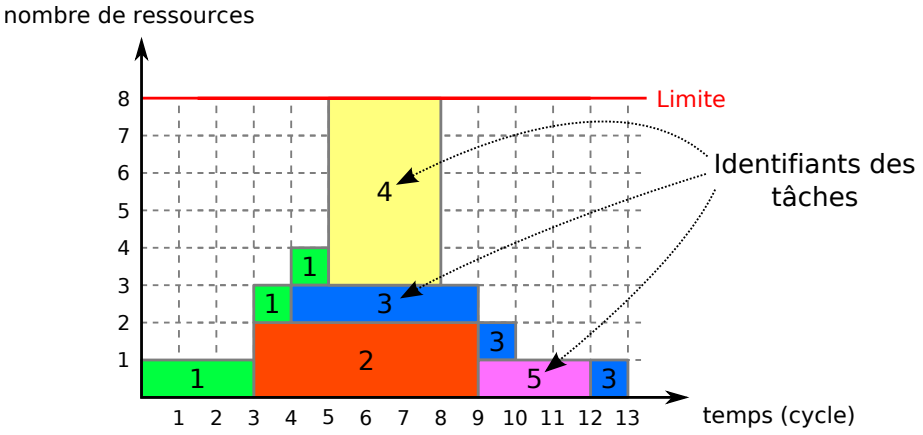


FIG. 2.1 – Utilisation de la contrainte cumulative pour l'ordonnancement de 5 tâches : le nombre de ressources utilisées ne doit pas dépasser, à aucun moment, le nombre maximum de ressources ici égal à 8.

2.2, chaque rectangle $Rect_i$ est défini par quatre variables, ses origines x_i et y_i, sa longueur Δx_i et sa hauteur Δy_i.

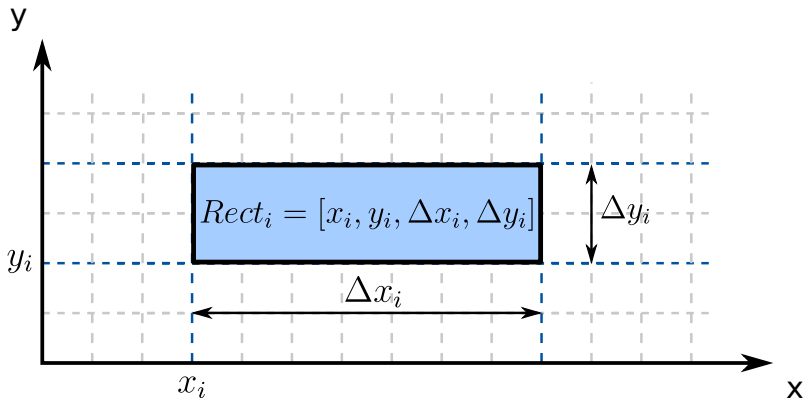


FIG. 2.2 – Définition d'un rectangle pour la contrainte différentielle bidemensionnelle.

La contrainte Diff2 permet de modéliser aisément le problème de partage de ressources lors de l'ordonnancement d'un ensemble de tâches. En effet, nous utilisons cette contrainte pour modéliser l'activité des ressources d'une architecture matérielle dans le temps, par exemple l'activité des opérateurs ou des mémoires d'un processeur. La figure 2.3 représente l'activité des mémoires d'un processeur dans un espace bidimensionnel, l'axe des abscisses représente le temps (en cycle) et l'axe des ordonnées l'identifiant de chaque mémoire. L'utilisation de cette contrainte permet d'assurer que tout au long de l'exécution d'un programme sur ce processeur aucun accès concurrent ne sera effectué sur une même mémoire.

ExtensionalSupport Cette contrainte permet de définir toutes les combinaisons possibles de valeurs qui peuvent être affectées à un ensemble de n variables.

La spécification d'une relation logique XOR ($a \oplus b = c$) est un exemple simple d'utilisation de cette contrainte. La table de vérité complète représentant toutes les combinai-

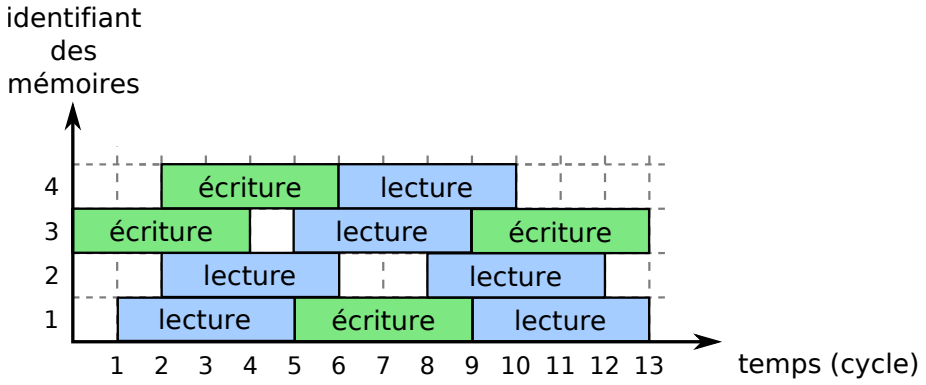


FIG. 2.3 – Exemple d'activité des mémoires d'un processeur dans un espace bidimensionnel. La contrainte Diff2 assure qu'il n'existe pas de recouvrement entre les rectangles représentant dans cet exemple une écriture ou une lecture en mémoire.

sons possibles et le vecteur spécifiant l'ordre des variables sont les paramètres de cette contrainte. Formellement cela se traduit de la façon suivante.

Soit trois variables booléennes a, b *et* $c \in \{0,1\}$, la contrainte permettant d'assurer une combinaison valide représentant la relation logique XOR pour le vecteur $v = [a,b,c]$ est défini par $ExtensionalSupport(v, \{\{0,0,0\},\{0,1,1\},\{1,0,1\},\{1,1,0\}\}$.

Cette contrainte est utilisée pour modéliser les connexions entre les éléments d'un réseau d'interconnexion point-à-point.

Remarque
Les contraintes globales représentent un des points forts de la CP. En effet, contrairement aux méthodes d'ILP et de MIP, elle est la seule approche permettant d'exprimer des contraintes globales spécifiques ainsi que des contraintes classiques (non globales) au sein d'un même environnement de résolution de problème. Cela permet d'exprimer facilement des problèmes complexes réels. De plus, pour chaque contrainte on associe un algorithme de réduction de domaine.

2.1.3 Algorithme de réduction de domaine

À chaque contrainte on associe un algorithme de réduction de domaine. Le rôle de cet algorithme est de supprimer des valeurs des domaines des variables de la contrainte pour lesquelles elle n'est pas satisfaite. Un algorithme de réduction de domaine fait appel à un mécanisme de propagation, appelé aussi technique de consistance.

2.1.4 Mécanisme de propagation

Lorsque le domaine d'une variable est réduit (par l'algorithme de réduction de domaine associé à la contrainte impliquant celle-ci) alors les conséquences de cette réduction sont étudiées pour les autres contraintes impliquant cette variable permettant ainsi de limiter encore plus son domaine. C'est le mécanisme de propagation de contrainte. D'après [95], « *L'idée sous-jacente à ce mécanisme est d'essayer d'obtenir des déductions globales. En effet, on espère que la conjonction des déductions obtenues pour chaque contrainte prise indépendamment conduira à un enchaînement de déductions. C'est-à-dire que cette*

conjonction est plus forte que l'union des déductions obtenues indépendamment les unes des autres ».

Les principales techniques de consistance sont la consistance par nœud, par arc et par chemin. Pour plus de détails, le lecteur peut se référer à [9].

2.1.5 Mécanisme de recherche de solutions

Un CSP a une solution lorsque toutes les variables ont une valeur et que ces valeurs respectent toutes les contraintes du problème. Selon le type de problème, il est possible de trouver une solution, toutes les solutions ou encore les solutions optimales qui minimisent (ou maximisent) une fonction de coût.

Le principal mécanisme de recherche de solutions utilisé est l'algorithme basé sur un parcours en profondeur (*Depth First Search* - DFS). Cet algorithme recherche une solution possible en organisant l'espace de recherche en arbre. A chaque nœud de cet arbre une variable se voit affecter une valeur de son domaine, le choix de cette valeur dans le domaine peut-être spécifié par l'utilisateur (cela peut être la plus petite valeur, la plus grande, la médiane ou encore un choix aléatoire). A chaque nœud, une décision est prise : soit le nœud est étendu (c.-à-d. la recherche continue), soit la recherche est coupée à ce nœud. Une coupure est effectuée si l'assignation effectuée ne remplit pas toutes les contraintes. Étant donné qu'une affectation déclenche la propagation de la contrainte et l'ajustement possible du domaine de la variable représentant la fonction de coût, la décision de continuer ou de couper la recherche à ce nœud peut facilement être effectuée.

Dans le solveur que nous utilisons, JaCoP [2], la minimisation (ou maximisation) d'une fonction de coût s'effectue en définissant une variable de coût et en utilisant un algorithme de recherche dit de séparation et évaluation (*Branch and Bound* - B&B). A chaque fois qu'une solution est trouvée avec $cost = costValue_i$, une contrainte $cost < costValue_i$ est imposée. Par conséquent, la recherche trouve des solutions avec un coût plus faible jusqu'à ce que finalement elle échoue à trouver une solution ce qui prouve que la dernière solution trouvée est optimale (c.-à-d. qu'il n'existe pas de meilleure solution).

Il existe aussi des mécanismes de recherche partielle comme la recherche par crédit (*credit search*) ou encore la recherche à divergence limitée (*Limited Discrepancy Search* - LDS).

Enfin il est possible de combiner plusieurs mécanismes de recherche en un unique mécanisme de recherche complexe (*Combining search*).

2.1.6 Modélisation

En programmation par contraintes, on distingue la modélisation du problème, c.-à-d. l'énonciation du problème, et sa résolution. La difficulté est double car le modèle doit non seulement répondre au problème lui-même mais aussi être construit de telle sorte que sa résolution soit efficace. D'après [95], l'une des difficultés majeures de la modélisation est l'identification des contraintes. Pour cela, l'auteur donne deux conditions amenant une résolution efficace :

1. les contraintes doivent être fortes afin d'engendrer des modifications des domaines des variables ;

2. les modifications dues à une réduction de domaine doivent pouvoir être utilisées par les autres contraintes.

2.2 Application à la conception et à la compilation pour architectures embarquées

L'utilisation de la CP pour la conception et la compilation pour des systèmes électroniques embarqués n'est pas nouvelle, d'autres laboratoires de recherche et certains laboratoires industriels de R&D ont déjà présenté des travaux sur ce sujet.

2.2.0.1 Application à la synthèse d'architecture

Renate Beckmann propose dans ses travaux présentés en 1994 dans [12] d'évaluer l'utilisation de la CP pour la conception de systèmes matériels à partir d'une spécification en langage de description d'architecture (*Hardware Description Language* - HDL) au niveau algorithmique représentant la structure et le comportement du circuit. Il démontre ainsi que cette approche est pertinente pour des problèmes complexes réels tels que la synthèse de haut niveau (HLS), la simulation, la génération de code et la synthèse mémoire. Avant cela, la majorité des travaux traitait de la conception de circuits à des niveaux d'abstraction plus bas (au niveau portes logiques voir encore plus bas).

Krzysztof Kuchcinski fait aussi partie des chercheurs à s'intéresser à l'utilisation de la CP pour la HLS. En 1998, il publie un premier article [65] qui propose une méthode pour la modélisation du problème de synthèse de chemins de données. Avec cette méthode sont modélisées les techniques de multi-cycles, chaînage, *pipelining*, *pipelining* algorithmique. Le prototype a été implémenté avec le solveur CHIP.

Ensuite, en 2003, il publie un autre article [67] sur le sujet qui peut être vu comme une extension de ses précédents travaux. Cette fois, un nouveau solveur *open source* nommé JaCoP [2] est développé, permettant d'ajouter de nouvelles contraintes spécifiques à un problème ou un domaine à l'inverse d'un solveur propriétaire fermé comme CHIP. Le solveur JaCoP a été spécialement implémenté pour résoudre des problèmes d'ordonnancement dans le domaine de la HLS et pour l'ordonnancement au niveau système. Ce second article exploite la représentation intermédiaire HCDG (*Hierarchical Conditional Dependency Graph* ou Graphe Hiérarchisé aux Dépendances Conditionnées) [8] qui représente à la fois la partie flot de données et flot de contrôle. De plus, des transformations formelles sur un HCDG permettent d'identifier un grand nombre d'exclusions mutuelles (typiquement l'identification de parties du programme qui peuvent être exécutées en parallèle) ce qui est très utile dans le cadre du partage conditionnel de ressources.

2.2.0.2 Application au placement d'applications sur architectures parallèles embarquées

Plusieurs contributions ont été proposées concernant le placement d'applications sur architectures embarquées.

Tout d'abord, les travaux de thèse de Cecilia Ekelin [38] ont permis d'aboutir à un environnement pour l'ordonnancement de systèmes temps réel embarqués intégrant la prise

en compte des contraintes de conception tout en permettant des optimisations. Elle propose aussi des techniques de parcours de l'espace des solutions pour réduire le temps d'exécution de l'algorithme d'optimisation passant d'un temps de l'ordre de la minute à la seconde. Les résultats obtenus grâce à ces techniques montrent une amélioration de la qualité des solutions et une exécution de l'algorithme d'optimisation aussi rapide, voire plus rapide, par rapport aux algorithmes de recuit simulé (*Simulated Annealing* - SA) et de B&B. Comme dans beaucoup d'études, le contexte architectural est très simple et ne tient donc pas compte de toutes les caractéristiques et contraintes d'une architecture réelle.

La société Thales, leader mondial dans les systèmes d'informations critiques pour les domaines de la défense et de la sécurité, de l'aéronautique, et du transport, a aussi travaillé sur l'utilisation de la CP essentiellement pour le placement d'applications de traitement du signal sur architectures parallèles.

Les premiers travaux sur le sujet ont été publié en 1997 à travers la thèse de Christophe Guettier [48] qui avait pour objectif de proposer une méthode de placement automatique d'applications de traitement du signal systématique en utilisant une modélisation concurrente et la CP. Cette première thèse a permis de valider la faisabilité d'une telle approche dans un contexte applicatif et architectural idéalisé.

Ces travaux ont été étendus à travers une seconde thèse effectuée par Nicolas Museau publiée en 2001 [82] qui avait pour ambition d'étendre ce contexte en ciblant des algorithmes et des architectures réels plus complexes. Il en résulte une modélisation d'une architecture multi-SPMD[1] et une étude formelle sur la détection des communications dépendantes du placement. Ces contributions ont été implémentées dans un outil d'aide au placement d'applications sur architectures parallèles appelé APOTRE.

Le solveur développé par Thales nommé « Eclair » repose sur le langage « Claire » [22], il contient depuis 2005 une bibliothèque appelée ToOls pour la conception d'algorithmes de recherche complexes arborescents (c.-à-d. par arbre) qui permettent l'obtention de meilleurs résultats en comparaison d'une recherche utilisant un parcours en profondeur (DFS) [35].

Le projet coopératif DREAM-UP exploite ces travaux en connectant l'outil de compilation PIPS [4] avec l'outil d'aide au placement APOTRE permettant notamment une exploration rapide de l'espace de recherche pour le placement d'applications multimédia embarquées sur SoC. La papier [58] présente le cas du placement d'une partie critique de la norme H.264 [58]. L'architecture modélisée dans ce projet est de type SIMD et les applications visées sont issues du domaine du traitement du signal systématique.

2.3 Application à la conception et à la compilation pour architectures reconfigurables

Cette thèse s'inscrit dans une démarche globale de modélisation d'architectures reconfigurables pour l'accélération de parties critiques d'applications de traitement du signal et de cryptographie notamment. Ces travaux ont abouti au développement de deux systèmes d'extension automatique de jeu d'instructions d'un processeur embarqué par des cellules reconfigurables formant ainsi un ASIP reconfigurable.

[1] Une architecture multi-SPMD comporte plusieurs unités de traitement de type *Single Program Multiple Data*, c.-à-d. exécutant un même programme sur plusieurs données en parallèle.

Dans le cadre de la conception et de la compilation pour architectures reconfigurables l'utilisation de la CP s'avère particulièrement bien adaptée et cela pour plusieurs raisons.

Tout d'abord parce que la CP permet de décrire aisément des problèmes d'optimisations combinatoires difficiles, voir prouvés NP-difficiles, et de résoudre plusieurs problèmes fortement corrélés simultanément. Ces propriétés sont celles des problématiques que nous traitons.

De plus, il est possible de prendre en compte, lors de la résolution d'un problème d'optimisation, de nombreuses contraintes architecturales, technologiques et applicatives exprimées à l'aide de « contraintes » au sens CP. A ce titre, des contraintes spécifiques aux problèmes de conception et de compilation pour architecture reconfigurable ont été proposées et sont maintenant disponibles dans les solveurs de contraintes comme JaCoP [2].

Enfin, il est possible d'optimiser différents aspects comme l'accélération d'une application ou bien l'utilisation des ressources matérielles nécessaires à son exécution. L'optimisation de plusieurs aspects dite « optimisation muti-objectifs » fait aussi partie des possibilités offertes par la méthodologie de CP. Cette possibilité n'étant pas étudiée dans les travaux de cette thèse, son utilisation représente une perspective de recherche.

2.3.1 Modélisation d'une application

Une application, spécifiée par un programme informatique écrit dans un langage de haut niveau peut être modélisée sous la forme d'un graphe.

La théorie associée à cette représentation, appelée « théorie des graphes », est un domaine largement étudié et cela est vrai aussi bien en informatique qu'en mathématiques. L'utilisation des graphes est donc très courante pour de nombreux domaines d'application.

Différentes représentations d'une application existent en fonction des informations dont on souhaite disposer. Le graphe de flot de données et de contrôle (CDFG) ou sans information de contrôle (DFG) mais aussi les représentations plus complexes comme le graphe hiérarchisé aux dépendances conditionnées (HCDG) sont communément employées pour représenter une application. On parle en compilation de représentation intermédiaire d'une application.

2.3.2 Modélisation d'une architecture

Une architecture peut être modélisée par un ensemble de contraintes appliquées sur la représentation intermédiaire de l'application. De plus, les unités de calcul et les réseaux d'interconnexion d'une architecture peuvent être aussi modélisés sous forme de graphes.

Enfin, les fortes contraintes technologiques imposées aux concepteurs de produits électroniques embarqués peuvent être elles aussi modélisées sous forme de contraintes (fréquence maximale, surface de silicium, etc.).

2.3.3 Notre méthodologie

La programmation par contraintes et la théorie des graphes sont deux approches qui, combinées, représentent un fort potentiel quant à la modélisation et la résolution de problèmes d'optimisation pour la conception et la compilation d'architectures reconfigurables dédiées à l'accélération d'applications pour des systèmes électroniques embarqués. C'est pourquoi nous avons choisi cette approche pour adresser le problème d'adéquation Architecture Application.

2.3.4 UPaK et DURASE deux systèmes pour la conception et l'utilisation d'extensions reconfigurables pour ASIP

2.3.4.1 UPaK

Le système UPaK (*Abstract Unified Patterns Based Synthesis Kernel for Hardware and Software Systems*) [117], est le premier système à intégrer les travaux de Krzystof Kuchcinski et de Christophe Wolinski sur la conception d'extensions reconfigurables pour ASIP en utilisant la CP et la représentation HCDG au sein d'un flot de conception complet. Ce flot, représenté sur la figure 2.4, permet d'identifier de nouvelles instructions, de sélectionner des instructions spécifiques pour accélérer une application et d'ordonnancer cette dernière en utilisant l'architecture reconfigurable ainsi spécialisée. Les extensions sont implémentées sous forme d'instructions séquentielles ou parallèles qui sont exécutées sur une (ou plusieurs) unité(s) reconfigurable(s) capable(s) d'exécuter toutes les instructions spécialisées. Le modèle d'ASIP généralisé défini dans le contexte UPaK est représenté sur la figure 2.5.

Pour cela, certaines contraintes globales spécifiques ont été développées : la contrainte *connected component* utilisée pour l'identification de nouvelles instructions, la contrainte de *(sub)graph isomorphism* qui permet d'effectuer l'isomorphisme de sous-graphe, utilisée surtout pour la sélection d'instructions. De plus, UPak intègre la fusion de motifs de calcul pour permettre d'implémenter efficacement et sous contraintes de conception ou technologiques toutes les instructions sélectionnées. Cette étape du flot qui représente la première contribution de cette thèse, est présentée dans le chapitre suivant (chapitre 3). A ce titre, une contrainte globale nommée *clique* et la méthode de consistance associée ont été développées, elle permet de spécifier la notion de clique au sein d'un graphe non orienté utilisée pour la fusion d'instructions. Toutes ces contraintes sont présentées en détails dans le papier intitulé « contraintes de graphe dans la conception de système embarqué » [116].

2.3.4.2 DURASE

Le système DURASE (*Generic Environment for Design and Utilization of Reconfigurable Application-Specific Processors Extensions*), issu des travaux de thèse de Kévin Martin [72], hérite de UPaK ; il est plus abouti et donne de meilleurs résultats. Le flot de ce système est donné figure 2.6. C'est le premier système à intégrer le *front end* de compilation générique nommé GeCoS [1] effectuant aussi certaines optimisations au niveau du CDFG (suppression de code mort, transformation dans la représentation en assignation unique statique, propagation de constantes, simplification algébrique, extraction et déroulage de boucle, etc.). Il a été récemment étendu par des optimisations utilisant conjointement le modèle polyédrique et la représentation intermédiaire HCDG.

De plus, le flot de DURASE permet d'effectuer du co-développement matériel/logiciel car il génère les extensions reconfigurables et le code assembleur permettant de les exploiter au mieux et cela pour un cœur de processeur embarqué existant, le NIOS II d'ALTERA par exemple. Les deux parties les plus importantes dans le flot DURASE sont la génération de motifs et la couverture du graphe d'application combinées à l'ordonnancement. A noter que dans ces travaux le modèle d'ASIP considéré est une extension reconfigurable comportant une seule cellule et non pas plusieurs en parallèle comme dans UPaK.

DURASE intègre aussi dans son flot la fusion de motifs de calcul en utilisant la contrainte *clique*.

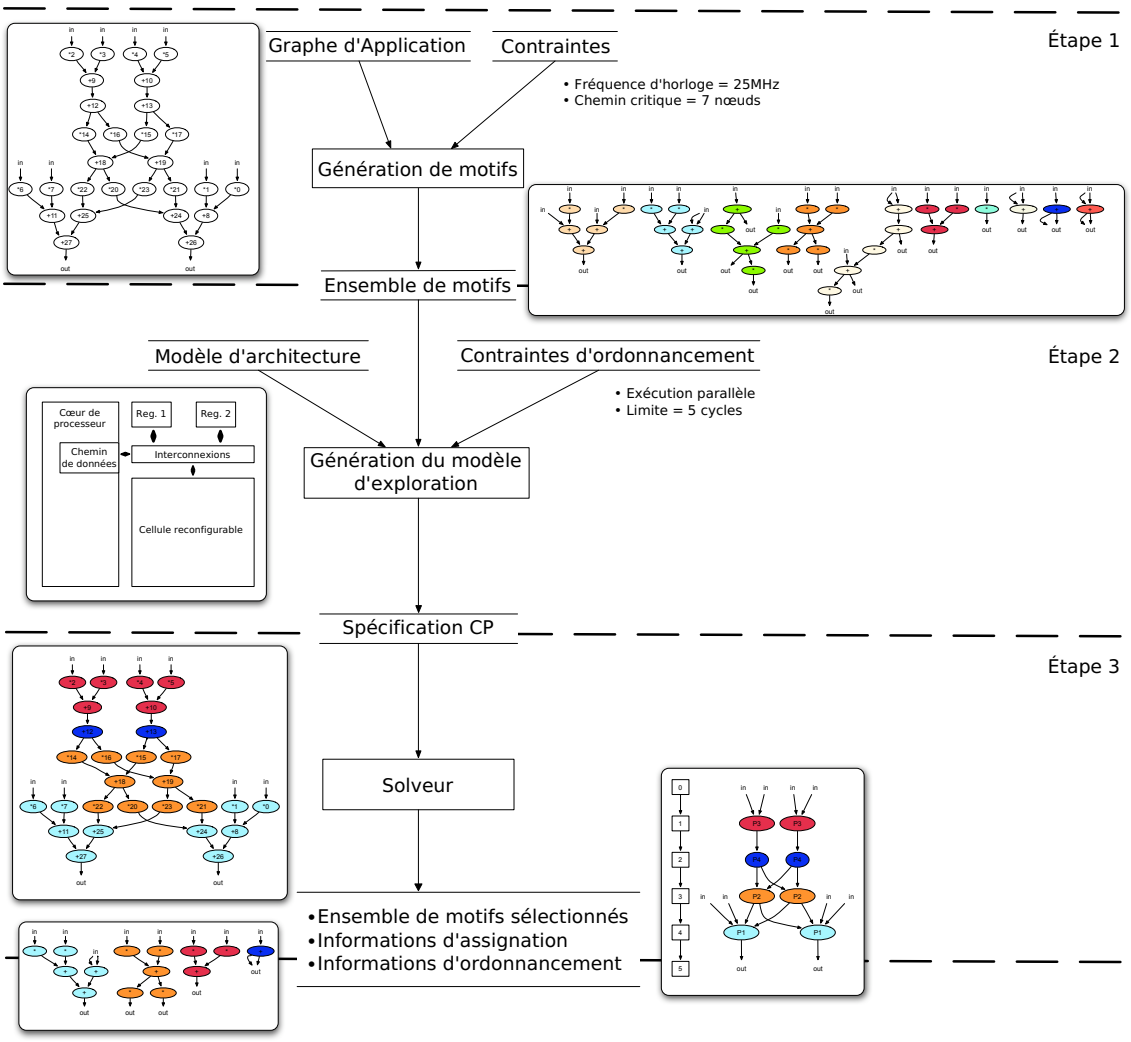


FIG. 2.4 – Flot de conception UPaK pour l'identification et la sélection d'instructions.

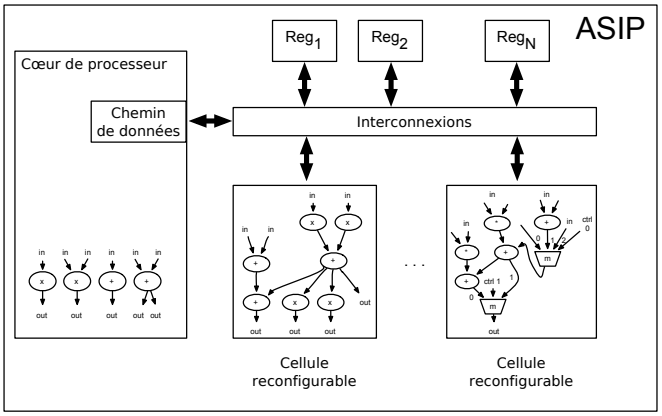


FIG. 2.5 – Modèle généralisé d'un ASIP dans le contexte UPaK.

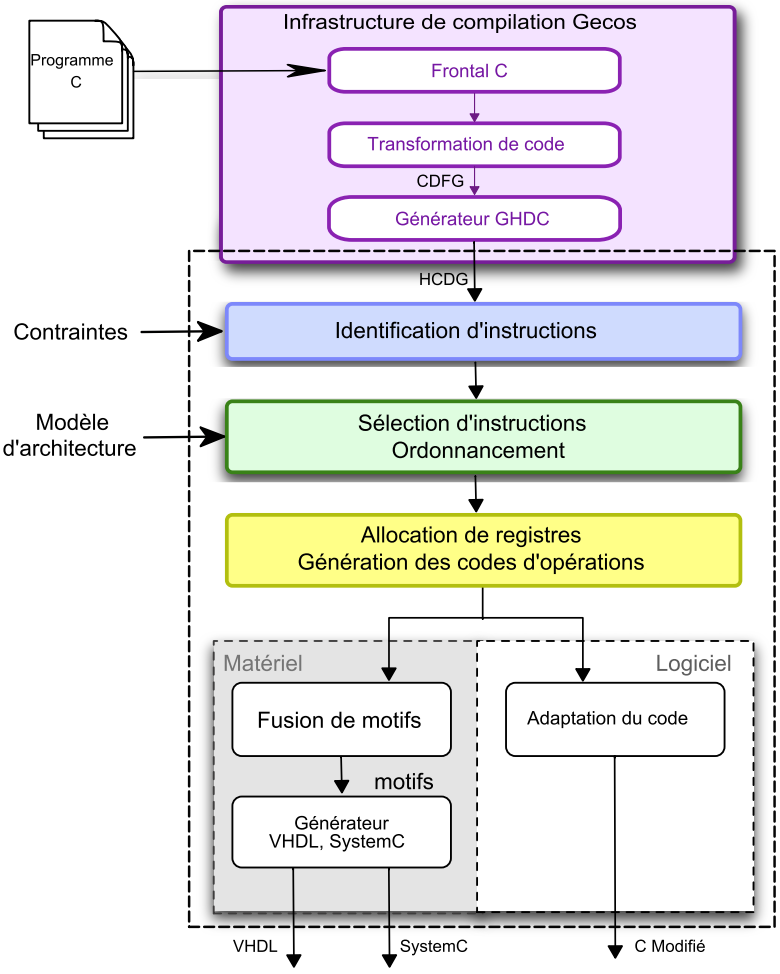


FIG. 2.6 – Le système DURASE [72].

Plus récemment, Antoine Floc'h a étendu le système DURASE par le support d'architectures parallèles génériques incluant les architectures de type VLIW [42]. Le flot et le modèle d'architecture sont représentés respectivement par la figure 2.7 et 2.8. La méthode proposée permet de modéliser une architecture parallèle reconfigurable, la sélection d'instruction et l'ordonnancement de l'application. Les instructions sélectionnées peuvent être existantes ou générées au sein du flot. De plus, la sélection d'instructions exploitées lors de l'ordonnancement pour une exécution parallèle utilise un nouveau modèle de couverture de graphe plus efficace pour ce type d'architecture.

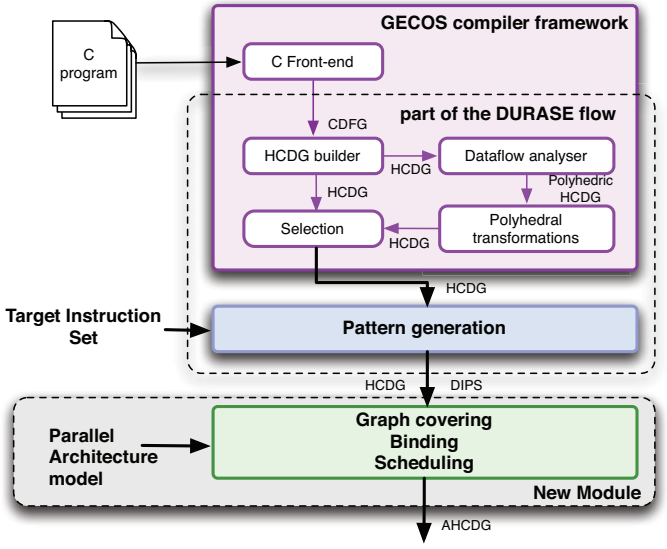


FIG. 2.7 – Le système DURASE étendu pour le support d'architecture parallèles génériques [42].

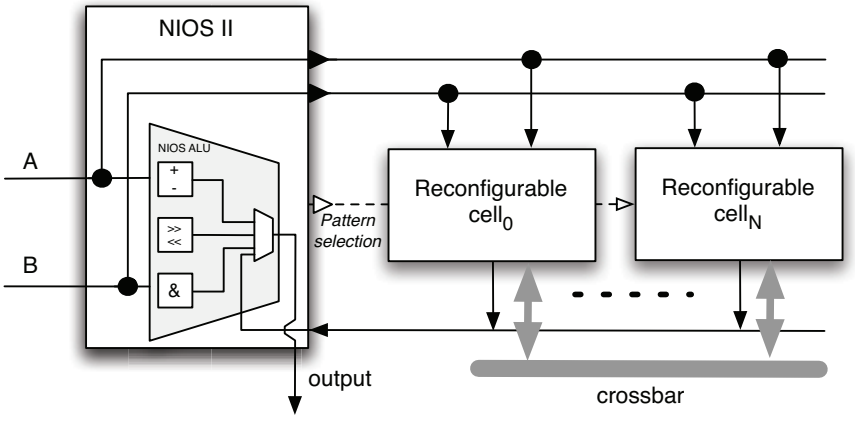


FIG. 2.8 – Modèle d'architecture d'ASIP considéré dans les travaux d'Antoine Floch [42].

2.4 Conclusion

La CP est une méthode utilisée pour résoudre des problèmes combinatoires complexes tels que la planification, la gestion et la production industrielle. Tout d'abord, nous avons présenté dans ce chapitre les bases de la CP, les différents types de contraintes ainsi que les principaux mécanismes de résolution et les problématiques générales de modélisation. Ensuite, nous avons montré que cette méthode a été utilisée pour résoudre des problèmes liés à la conception et à la compilation pour des architectures embarquées, particulièrement les problèmes de synthèse d'architecture à haut niveau et de placement d'applications pour des systèmes temps réels et parallèles. Enfin, nous démontrons à travers les systèmes UPaK et DURASE comment on peut exploiter la CP dans le contexte des architectures reconfigurables à gros grain.

L'aspect novateur de l'approche utilisée dans ces deux systèmes réside dans la capacité de la CP à exprimer/modéliser simplement, notamment à l'aide de contraintes globales spécifiques aux problèmes traités, et au sein d'un environnement unique, un ensemble de problèmes combinatoires complexes. Cela nous permet de modéliser les contraintes issues de l'application et de l'architecture ciblées tout en optimisant un ou plusieurs aspects comme la performance, la surface, etc. De plus la résolution est effectuée par un solveur, utilisant des méthodes elles aussi spécifiques aux problèmes, qui fournit dans la majorité des cas la preuve de l'optimalité de la solution.

La CP représente aujourd'hui une méthode prometteuse pour résoudre certains problèmes complexes rencontrés dans la conception et le compilation pour architectures reconfigurables à gros grain.

Chapitre 3

Modèle de contraintes pour la fusion d'unités fonctionnelles reconfigurables

La fusion de chemins de données s'inscrit parfaitement dans un flot de synthèse de haut niveau pour système reconfigurable à gros grain. En effet cette technique permet de fusionner plusieurs unités fonctionnelles spécialisées en une seule unité reconfigurable au niveau fonctionnel ; la reconfiguration se faisant au niveau système par le choix de l'instruction spécialisée à exécuter. Le but de cette étape est de synthétiser un matériel performant, flexible et ayant une faible surface.

La fusion de chemins de données peut être intégrée dans le flot UPaK [117] ou encore DURASE [73] comme le montre la figure 3.1. Celle-ci illustre la spécialisation du jeu d'instructions d'un processeur pour l'application ARF (*Auto Regression Filter* ou filtre d'auto régression). Dans un premier temps, le programme en langage C de l'application (ou une partie), représenté dans le bloc (A) de la figure, est transformé en HCDG (*Hierarchical Conditional Dependency Graph*) [8] par l'outil GeCoS [1]. Nous nous concentrons dans ces travaux sur la partie flot de données de cette représentation qui est un graphe de flot de données (DFG) que nous nommons graphe d'application (noté AG) et qui est représenté dans le bloc (B). Après l'extraction de tous les motifs de calcul non isomorphes de l'AG (bloc (C)), ceux-ci sont utilisés pour couvrir le graphe dans le but de trouver un temps d'ordonnancement minimum permettant d'accélérer au plus l'application ciblée. L'AG couvert par les motifs M_1, M_2 et M_3 est représenté dans le bloc (D). Il en résulte un ensemble réduit de motifs sélectionnés qui sont ensuite fusionnés en un unique motif (bloc (E)). Ce motif est ensuite synthétisé pour créer une unité fonctionnelle reconfigurable (bloc (F)) qui est utilisée comme extension du chemin de données d'un processeur. Du point de vue système, l'extension correspond à la mise à disposition de nouvelles instructions spécialisées, qui, dans le système DURASE sont intégrées automatiquement dans le code de l'application. L'architecture ainsi spécialisée est un processeur dont le jeu d'instructions est spécifique à une application. Ce type de processeur est couramment appelé ASIP (*Application Specific Instruction set Processor*).

Ce chapitre présente la synthèse de cellules reconfigurables au niveau système sous contraintes technologiques dans le contexte de DURASE. Dans un premier temps est présenté le positionnement des travaux au regard des solutions déjà connues sur le sujet. Dans un deuxième temps le modèle de contraintes utilisé pour résoudre le problème de

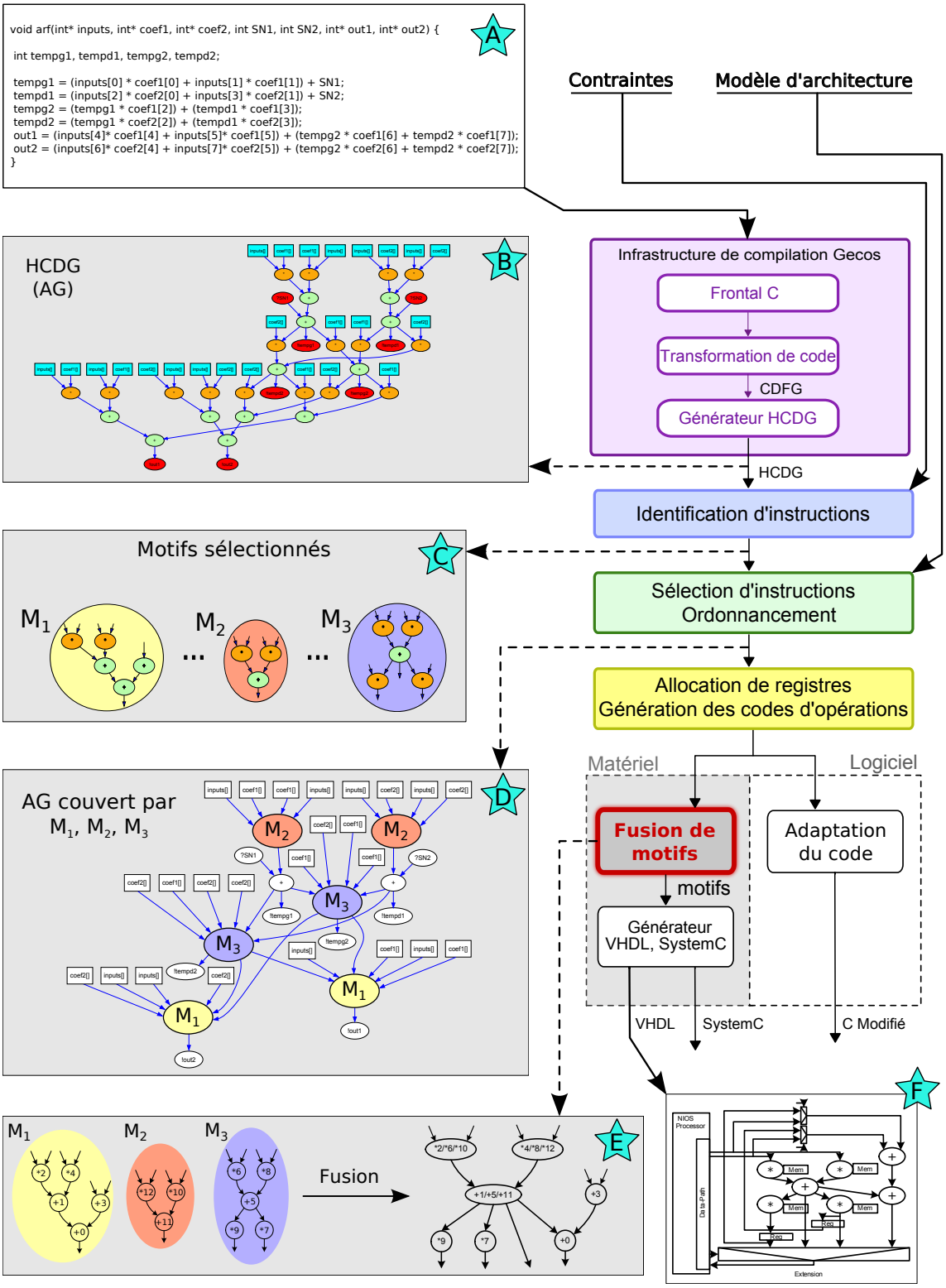


FIG. 3.1 – Positionnement de l'étape de fusion de chemins de données dans le flot DURASE (illustré sur l'exemple du filtre d'auto régression).

fusion de chemin de données est détaillé. Les résultats obtenus par ce modèle sont ensuite présentés. Une conclusion ainsi que des perspectives clôturent le chapitre.

3.1 État de l'art sur la fusion de chemins de données dans le cadre de la synthèse d'architecture matérielle

Dans la littérature, les travaux sur la fusion de chemin de données portent essentiellement sur les architectures reconfigurables et cela à tous les niveaux de granularité, depuis un grain fin [100] jusqu'à un gros grain [56, 45].

Le problème de fusion de chemin de données est le plus souvent modélisé comme un problème d'optimisation de graphe. Un chemin de données appelé aussi flot de données, représentant un motif de calcul complexe, est modélisable par un graphe acyclique orienté (noté ici AG) où chaque nœud représente une opération de base (opération arithmétique ou logique) ou une variable et chaque arc représente une dépendance de données entre deux nœuds (voir définition 3.1). Le problème de fusion de deux AG peut être considéré comme une réduction du problème d'isomorphisme de sous-graphe, prouvé comme étant NP-difficile [79].

Définition 3.1 : *Un graphe de flot de données (ou de chemin de données) est un graphe acyclique orienté noté* $AG = (V, E)$*,* V *désigne l'ensemble des nœuds et* E *l'ensemble des arcs.*

- *Un nœud* $v \in V$ *représente une opération ou une variable ;*
- *un arc* $e = (u, v) \in E$ *représente un transfert de données entre les nœuds* u *et* v*.*

Parmi les travaux réalisés sur la fusion de chemin de données et plus largement sur le problème d'isomorphisme de sous-graphe, deux techniques sont apparues comme le point de départ de la contribution présentée dans ce chapitre.

La première méthode concerne la fusion de chemin de données (*DataPath Merging*, DPM) pour architecture partiellement reconfigurable, thématique au combien proche de celle traitée dans cette thèse. De plus ces travaux s'inscrivent dans le projet MESCAL [3, 46], dont l'un des axes de recherche concerne les ASIP. Dans ces travaux, la technique de résolution utilise un algorithme basé sur la recherche d'une clique de poids maximum. Cette technique est présentée par Moreano dans [81].

3.1.1 Algorithme basé sur la recherche d'une clique de poids maximum

La technique présentée dans [81] est l'aboutissement de plusieurs travaux sur le sujet.

En effet, dans un premier temps, un article datant de 2001 [56] propose dans le cadre d'un système reconfigurable dynamiquement de fusionner des chemins de données représentant les boucles d'un programme en un seul chemin de données reconfigurable.

Dans le méthodologie proposée, le code (en langage C) de l'application visée (ou d'un ensemble d'applications) est partitionné en deux : une partie est exécutée sur un processeur à usage général et l'autre, représentant les boucles du programme, est placée sur un co-processeur reconfigurable. Ce dernier est composé d'un chemin de données reconfigurable formé d'unités fonctionnelles fixes et de registres reliés par un réseau d'interconnexion reconfigurable. Un bloc reconfigurable à grain fin de type FPGA est utilisé pour générer les signaux de contrôle du chemin reconfigurable. Le placement des boucles du programme

sur le coprocesseur reconfigurable consiste à définir les unités fonctionnelles ainsi que les interconnexions et les registres, et cela, à partir des chemins de données représentant les différentes boucles. Les auteurs proposent de fusionner ces chemins de données par une technique basée sur une correspondance bipartite de poids maximum qui, utilisée comme heuristique, permet de minimiser le réseau d'interconnexion reconfigurable.

Pour fusionner plus de deux chemins de données, chaque paire est combinée itérativement, en ajoutant un chemin de données à chaque fois.

Une approche similaire, toujours dans le domaine de la reconfiguration dynamique, avait été proposée à un niveau de granularité plus fin par Shirazi [100].

En 2002, N. Moreano présente dans [80] une nouvelle technique itérative basée sur l'utilisation de graphe pour effectuer la fusion de chemins de données. Celle ci est basée sur la recherche d'une clique de poids maximum qui permet de fusionner deux AG à la fois. En 2004, à travers leur collaboration dans le projet MESCAL, les équipes de Huang et de Moreano publient un article [57] présentant, dans un contexte très proche du nôtre, un flot complet incluant la fusion de chemins de données. L'année suivante, C. de Souza évalue les différentes heuristiques présentes dans la littérature pour résoudre ce problème, dont celle de Moreano en 2002 [36]. La conclusion de cet article présente l'heuristique de Moreano comme le meilleur algorithme suboptimal disponible pour le DPM, battant la populaire heuristique de la correspondance bipartite dans tous les tests.

Dans le dernier article de Moreano sur ce sujet [81], des améliorations sont apportées à la technique originale, notamment sur la fonction de coût représentant le gain en surface d'une fusion mais aussi la prise en compte de la commutativité des opérateurs.

Etant donné que ces travaux représentent la base de la contribution présentée dans ce chapitre, nous nous proposons d'illustrer les différentes étapes de l'algorithme décrit dans [81] à travers un exemple simple. Dans le but de ne pas avoir à re-définir plus tard dans le manuscrit les notions définies dans ces travaux, celles-ci ont été adaptées.

3.1.1.1 Étape 1 : Appariement de nœud et d'arc

Pour fusionner deux chemins de données représentés par deux AG, AG_i et AG_j , la première étape consiste à appareiller les nœuds ainsi que les arcs des deux graphes qui peuvent être fusionnés. Deux nœuds $u_i \in AG_i$ et $u_j \in AG_j$ sont appareillés s'ils représentent tous les deux une opération pouvant être exécutée par la même ressource matérielle. Deux arcs $(u_i, v_i) \in AG_i$ et $(u_j, v_j) \in AG_j$ sont appareillés si u_i et u_j le sont ainsi que v_i et v_j, c'est à dire que leurs nœuds source ainsi que leurs nœuds destination sont appareillés.

Par exemple, soient AG_1 et AG_2 deux AG à fusionner représentés sur la figure 3.2. Les nœuds u_1 et u_2 sont appareillés car ils représentent chacun une multiplication et peuvent donc être tous les deux exécutés par un seul multiplieur. De même, w_1 et v_2 peuvent être appareillés si l'on dispose d'un bloc matériel capable d'exécuter une addition et une soustraction. Concernant les arcs : l'arc (w_1, y_1) de AG_1 est appareillé avec l'arc (v_2, w_2) de AG_2.

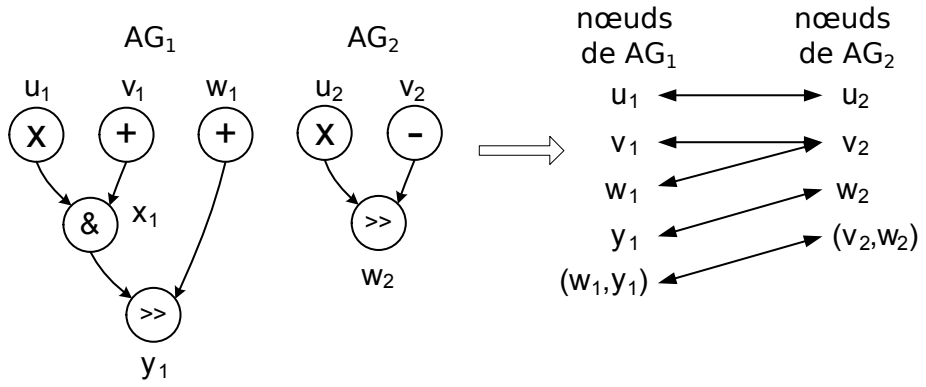


FIG. 3.2 – Exemple illustrant la première étape de l'algorithme de Moreano : appariement de nœuds et d'arcs.

3.1.1.2 Étape 2 : Construction du graphe de compatibilité

Dans la technique proposée par Moreano, la deuxième étape consiste à construire le graphe de compatibilité. La figure 3.3 illustre cette étape pour les appariements entre les deux AG de la figure 3.2.

Dans ce graphe, chaque appariement de nœuds ou d'arcs est représenté par un nœud. Un arc entre deux nœuds du graphe de compatibilité représente la compatibilité entre ces deux appariements, c.-à-d. que si ces deux appariements sont sélectionnés dans la solution alors ils n'amènent pas d'incohérence dans cette solution. Deux appariements ne sont pas compatibles si un nœud est présent dans les deux appariements comme le montre la figure 3.4 où le nœud v_2 est présent dans les deux appariements (v_1, v_2) et (w_1, v_2). Cela se traduit dans le graphe de compatibilité par une absence d'arc entre les deux nœuds représentant ces appariements (voir la figure 3.3). Une définition formelle du graphe de compatibilité est donnée ci-dessous (définition 3.2).

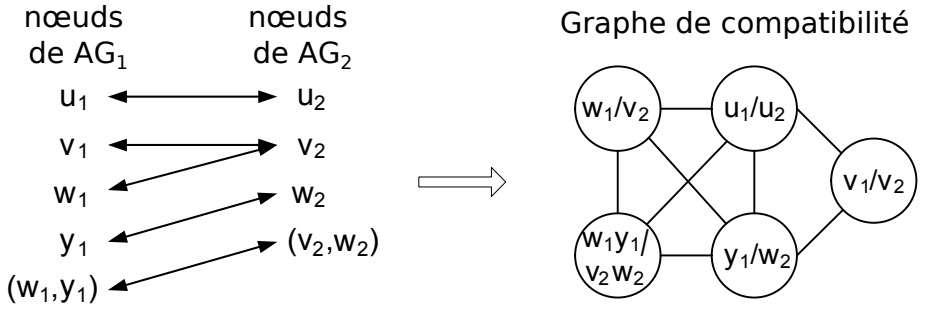


FIG. 3.3 – Exemple illustrant la deuxième étape de l'algorithme de Moreano : création du graphe de compatibilité.

Définition 3.2 : *Un graphe de compatibilité, correspondant aux graphes $AG_i = (V_i, E_i)$ et $AG_j = (V_j, E_j)$, est un graphe non-orienté $GC = (V_c, E_c)$, où V_c est un ensemble de nœuds et où $E_c \subseteq V_c \times V_c$ est un ensemble d'arcs. Un graphe de compatiblité comporte deux types de nœud :*

- *les nœuds réguliers, représentant un appariement de nœuds notés u_i/u_j, où $u_i \in V_i$ et $u_j \in V_j$,*
- *les nœuds d'arcs, représentant un appariement d'arcs notés $(u_i, v_i)/(u_j, v_j)$, où $(u_i, v_i) \in E_i$ et $(u_j, v_j) \in E_j$.*

Chaque arc $(u_c, v_c) \in E_c$ *du graphe de compatibilité représente la compatibilité entre les deux appariements* (u_c *et* v_c).

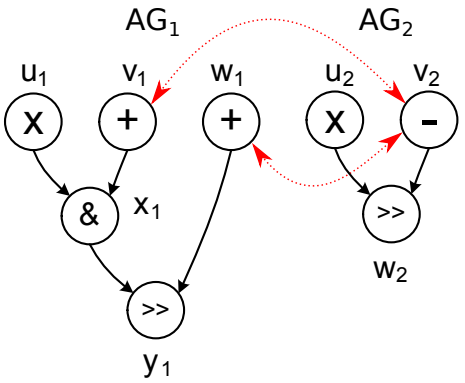


FIG. 3.4 – Exemple de deux appariements de nœuds incompatibles : (v_1, v_2) incompatible avec (w_1, v_2).

Le poids associé à chaque nœud du graphe de compatibilité représente le gain en surface obtenu si l'appariement est sélectionné dans la solution. Les équations 3.1 et 3.2 définissent le poids associé respectivement à un appariement de nœuds et d'arcs où u dénote le nœud du graphe de compatibilité, $Area(u)$ dénote la surface du bloc matériel capable d'exécuter les deux opérations représentées par u_i et u_j et $Area(Mux)$ dénote la surface d'un multiplexeur.

$$weight(u_i/u_j) = Area(u) - Area(Mux) \tag{3.1}$$

$$weight((u_i, v_i)/(u_j, v_j)) = Area(Mux) \tag{3.2}$$

Le calcul du gain en surface obtenu, si un nœud régulier est sélectionné, se décompose comme suit : la surface gagnée est égale à la somme des surfaces des deux blocs matériels nécessaires à l'exécution des deux nœuds issus de G_i et de G_j moins la surface du bloc matériel exécutant les deux nœuds dans le graphe fusionné, moins la surface du multiplexeur ajouté sur son entrée. Soit :

$$Weight(u_i/u_j) = Saved\ area = Area(u_i) + Area(u_j) - (Area(u) + Area(Mux)).$$

Si par simplification on considère que :

$$Area(u) = Area(u_i) = Area(u_j),$$

ce qui est le plus souvent le cas, alors on obtient la formule donnée par l'équation 3.1.

Le calcul du gain en surface obtenu, si un nœud d'arc est sélectionné, correspond au gain du multiplexeur sur l'entrée du bloc matériel exécutant les deux nœuds destinations v_i et v_j.

3.1.1.3 Étape 3 : Résolution de problème de clique de poids maximum

Afin de déterminer l'ensemble d'appariements de nœuds et d'arcs compatibles permettant de réduire au maximum la surface du chemin de données fusionné, Moreano utilise la technique de recherche de la clique de poids maximum. La notion de clique est définie dans la définition 3.3. L'algorithme utilisé est issu de « Cliquer » [83], implémentation libre des travaux présentés dans [84]. Cet algorithme a été modifié par l'équipe de Moreano pour borner son temps d'exécution en un temps polynomial égal à $|V_c|$; passé ce temps, la recherche est arrêtée.

Définition 3.3 : *Une clique est un ensemble de sommets deux à deux adjacents (notion de graphe complet). On utilise parfois le terme p-clique ou encore clique de cardinalité p pour désigner une clique contenant p nœuds.*

La figure 3.5 donne la clique de poids maximum pour fusionner les graphes AG_1 et AG_2 présentés précédemment (la clique de poids maximum est représentée en gras et les nœuds qui la composent sont grisés).

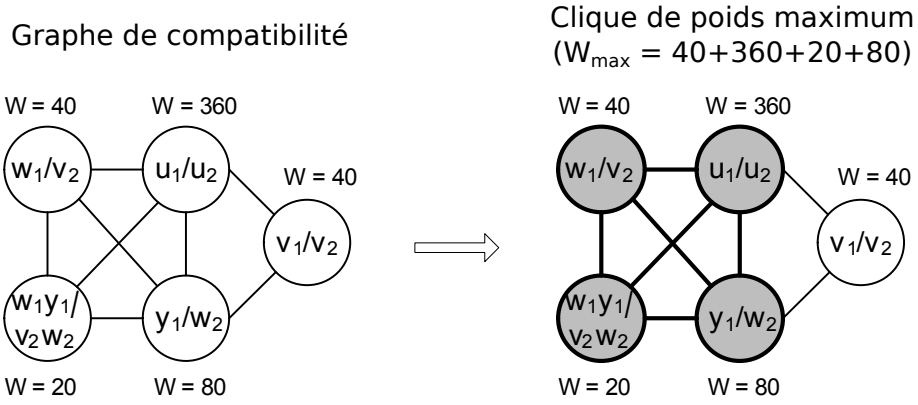


FIG. 3.5 – Exemple illustrant la troisième étape de l'algorithme de Moreano : résolution du problème de clique de poids maximum. La clique de poids maximum est représentée en gras et les nœuds qui la composent sont grisés.

3.1.1.4 Étape 4 : Reconstruction du graphe fusionné

Cette étape permet d'achever une itération de l'algorithme par la reconstruction du graphe fusionné à partir de la clique de poids maximum. Chaque appariement de nœuds de la clique correspond à un nœud dans le graphe fusionné. Chaque appariement d'arcs correspond à un arc dans ce même graphe. Les nœuds et les arcs des deux graphes fusionnés qui ne sont pas appareillés dans la clique de poids maximum sont ajoutés dans le graphe fusionné. Lorsqu'un nœud dans le graphe fusionné représentant un appariement comporte, sur une même entrée, deux arcs non appareillés, un multiplexeur est ajouté. La figure 3.6 représente le graphe fusionné à partir de la clique de poids maximum définie lors de l'étape précédente.

S'il reste des graphes en entrée, cette étape génère un graphe temporaire qui sera utilisé comme graphe d'entrée pour la prochaine itération.

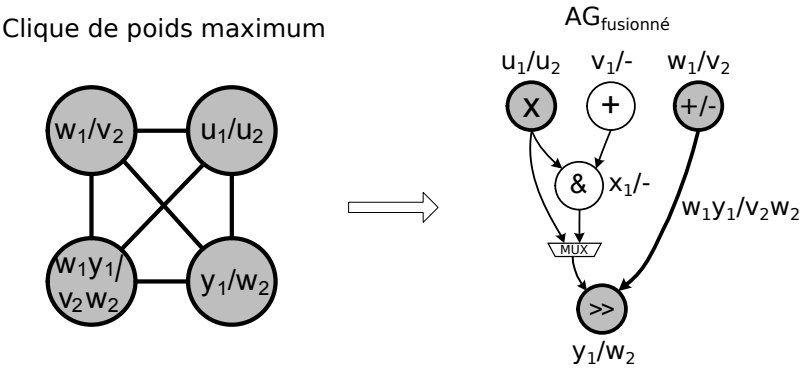


FIG. 3.6 – Exemple illustrant la dernière étape de l'algorithme de Moreano : reconstruction du graphe d'application fusionné à partir de la clique de poids maximum.

3.1.1.5 Discussion

Les travaux cités ci-dessus s'inscrivent totalement dans notre démarche et présentent de très bon résultats en termes de réduction de surface. Par contre il est important de noter ici que le but de ces travaux est d'obtenir le maximum de réutilisation des ressources matérielles (unités fonctionnelles et interconnexions). En aucun cas la performance du chemin de données résultant de la fusion n'est prise en compte. Or il est clair que pour une architecture reconfigurable il y a une relation entre le partage de ressources matérielles et les performances de celle-ci. Certains choix peuvent amener à baisser significativement la performance du chemin de données au profit d'un gain en surface.

A travers l'étude et l'implémentation de la méthode proposée dans [81], il est rapidement apparu que le gain en surface croit évidemment lors du partage de blocs fonctionnels mais ce gain est largement diminué lorsque les interconnexions ne le sont pas. En effet, partager un bloc sans partager ses entrées oblige à ajouter des multiplexeurs, et ceux-ci ont une surface proportionnelle à leur nombre d'entrées. Ce problème amène la question suivante : Comment partager partiellement deux chemins de données en réduisant cet effet de bord ?

3.1.2 Concept de contournement d'opérateur

Les travaux présentés par Corazao dans [30] permettent de répondre partiellement à cette question. Dans cet article est proposé une nouvelle approche de synthèse de haut niveau pour circuit intégré spécifique à une application comportant de nombreux chemins de données (*data-path-intensive ASIC*). Cette approche, orientée pour la performance, fait appel à des motifs déjà optimisés en tentant de les reconnaitre dans le flot de données d'une application. La clef de leur algorithme réside dans l'introduction de la notion de contournement d'opérateur permettant une correspondance partielle entre un motif et une partie de l'application (voir la définition 3.4).

Définition 3.4 : *Un nœud de motif est dit contournable sur une certaine entrée si sa valeur de sortie peut être égale à cette entrée en affectant aux autres entrées des constantes sans induire d'effet de bord.*

La figure 3.7 donne un exemple simple mettant en œuvre le contournement d'opérateur. Dans cet exemple, le graphe fusionné est identique au graphe le plus grand en termes

de nombre de nœuds. Cela est possible par le contournement de l'opérateur en grisé, un additionneur, en plaçant la constante zéro sur son entrée (valeur qui peut être obtenue en plaçant la constante zéro sur une des entrées du multiplieur en amont).

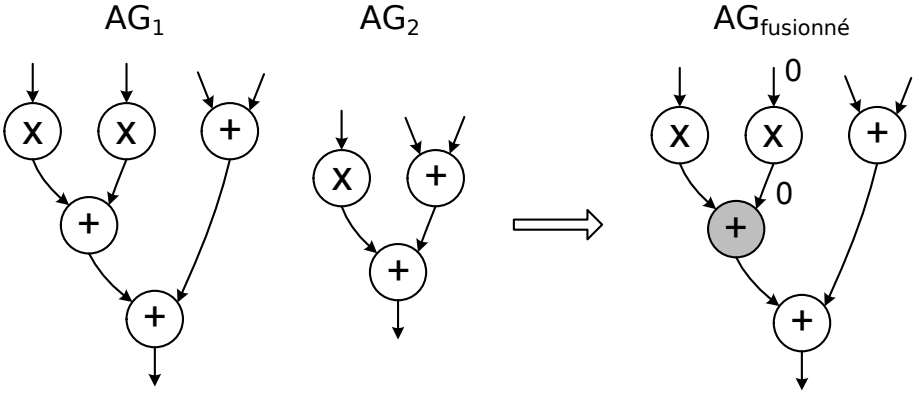


FIG. 3.7 – Exemple illustrant le concept de contournement d'opérateur.

On peut noter que le contournement d'un opérateur dyadique (c.-à-d. à deux entrées) sur une certaine entrée peut se faire par affectation de l'élément neutre sur l'autre entrée. Il est facile de montrer que tous les opérateurs arithmétiques et logiques dyadiques sont contournables. Ainsi, on peut définir la notion de chemin d'inhibition dans le but de contourner un opérateur sans avoir à ajouter un multiplexeur pour y affecter son élément neutre sur son entrée. Cette nouvelle notion, même si elle n'est pas exploitée ici, permet de définir si un opérateur complexe, c.-à-d. comportant plusieurs opérations arithmétiques et/ou logiques chaînées, comporte des opérateurs contournables et, si c'est le cas, les valeurs à affecter sur ses entrées pour les contourner.

Par contre, d'autres types d'opérateurs (comme par exemple un opérateur mono-adique, c.-à-d. à une entrée) ne sont pas contournables en utilisant leur élément neutre.

Ainsi, il est possible de contourner un opérateur de plusieurs manières.
- La première est d'affecter directement sur la bonne entrée de l'opérateur son élément neutre.
- La deuxième est d'affecter la (les) bonne(s) valeur(s) en entrée du chemin d'inhibition permettant d'appliquer l'élément neutre de l'opération sur l'entrée de l'opérateur à contourner sans avoir à ajouter un multiplexeur.
- La dernière est d'ajouter, dans l'opérateur lui-même, la logique nécessaire à son contournement.

Nous avons utilisé et étendu le concept de contournement d'opérateur pour la fusion de chemin de données afin de limiter l'ajout de multiplexeurs et ainsi d'optimiser la surface et la performance du chemin de données fusionné. Malheureusement, ce concept ne résout pas le problème d'augmentation non maitrisée du chemin critique.

3.1.3 Positionnement de la contribution au regard de l'état de l'art

Comme nous l'avons dit précédemment, les méthodes issues de la littérature permettent de fusionner un ensemble de chemins de données en maximisant le gain en surface. De plus, nous avons identifié la possibilité d'accroître les gains en surface en utilisant la technique de

contournement d'opérateur. Mais aucune méthode ne permet aujourd'hui de contraindre le processus de fusion par des aspects de performance, plus généralement, d'effectuer la fusion d'opérateurs reconfigurables gros grain sous contraintes architecturales et technologiques. C'est pour cette raison que nous nous sommes intéressés à la mise en place d'une méthode permettant à la fois, (1) de fusionner un ensemble de chemins de données en maximisant le gain en surface, (2) tout en donnant la possibilité d'ajouter des contraintes sur la performance de la solution.

Pour atteindre cet objectif, il était nécessaire de modéliser le problème global de fusion de chemins de données pour la minimisation de la surface tout en ajoutant des contraintes modélisant les limitations technologiques. Une limitation de ce type peut être, par exemple, de ne pas dépasser, dans le chemin de données final, la longueur du chemin critique le plus long parmi les chemins de données à fusionner.

3.2 Modèle de contraintes pour la fusion de chemins de données

3.2.1 Aperçu de l'algorithme

L'algorithme développé est schématisé par le figure 3.8. L'approche itérative de Moreano a été conservée. En effet les motifs sont fusionnés deux à deux et le résultat d'une itération est un motif temporaire utilisé comme entrée de l'itération suivante.

Lors de l'étape de fusion, on retrouve la génération du graphe de compatibilité à partir des appariements. Ensuite vient l'étape de génération des contraintes pour la recherche de la clique de poids maximum ainsi que pour les deux options que sont les problèmes du chemin critique et de l'ajout de multiplexeurs sur le chemin critique.

La génération des contraintes pour le problème du chemin critique permet d'assurer que la longueur du chemin critique de la solution recherchée ne dépasse pas le chemin critique le plus long en entrée. Quant au problème d'ajout de multiplexeurs sur le chemin critique, cela permet d'assurer que le chemin critique de la solution recherchée ne comporte pas plus d'un certain nombre de multiplexeurs. De plus, il est possible de spécifier le nombre maximum d'opérateurs contournés sur un même chemin.

Enfin l'étape dite d'optimisation consiste à résoudre le problème de la clique de poids maximum sous les contraintes générées précédemment.

Dans le cas où la longueur du chemin critique ne doit pas augmenter (option du problème du chemin critique), les motifs en entrée sont ordonnés. L'ordre défini est décroissant en termes de chemin critique c'est à dire du motif ayant le plus grand chemin critique jusqu'au motif ayant le plus petit chemin critique.

La procédure de mise à jour est appliquée à la fin d'une itération au nouvel ensemble de motifs composé de tous les motifs déjà fusionnés. Cette procédure injecte à ces motifs les informations concernant le partage des nœuds et la position du ou des multiplexeurs ajoutés sur les chemins critiques. Cette procédure n'est utilisée que pour l'option du problème d'ajout de multiplexeurs sur le chemin critique.

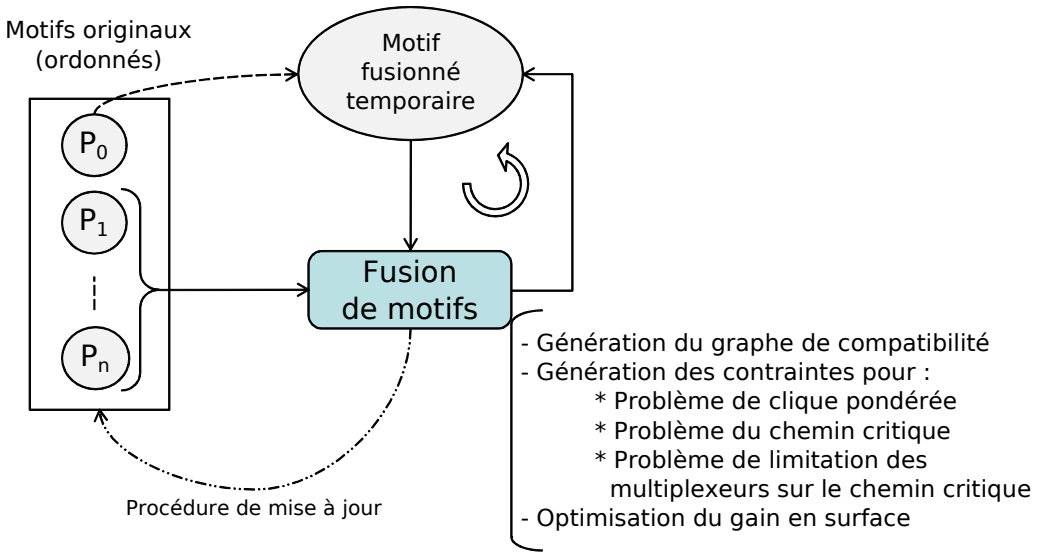


FIG. 3.8 – Aperçu de l'algorithme de fusion de chemins de données

3.2.2 Intégration et généralisation du concept de contournement d'opérateur

Le concept de contournement d'opérateur à été intégré sous la forme d'un nouveau type de nœud dans le graphe de compatibilité dit nœud « de chemin » et noté $(u_i, w_i, ..., v_i)/(u_j, v_j)$. Ce type représente la même idée que pour le type nœud « d'arc » mais cette fois l'arc à gauche de l'appariement (appartenant donc au motif temporaire) est un chemin qui part du nœud source u_i et qui arrive au nœud destination v_i en traversant des opérateurs contournables $(w_i, ...)$.

Le poids associé à ce nouveau type est de fait très proche de celui d'un nœud d'arc tout en prenant en compte le surcoût en surface lié au possible ajout de logique pour contourner l'opérateur (voir section 3.1.2). Si aucune logique n'est nécessaire, ce poids est égal à un multiplexeur. Il peut arriver que ce poids soit négatif si la surface de la logique nécessaire au contournement est supérieure à celle d'un multiplexeur. Ce poids est donné par la formule 3.3 où $Area((u_i, w_i, ..., v_i))$ représente le surcoût en surface nécessaire au contournement des opérateurs entre les nœuds source et destination.

$$weight((u_i, w_i, ..., v_i)/(u_j, v_j)) = Area(Mux) - Area((u_i, w_i, ..., v_i)) \tag{3.3}$$

3.2.3 Conditions de compatibilité entre appariements

Dans le graphe de compatibilité, un nœud représente un appariement et un arc la compatibilité entre deux appariements. La notion de compatibilité, introduite pécédemment (3.1.1.2), assure la cohérence de la solution. Le tableau 3.1 donne les conditions de compatibilité entre les différents types de nœuds du graphe de compatibilité dont le nouveau type nœud de chemin. Rappelons qu'il y a trois types d'appariement, les appariements de nœuds, d'arc et de chemin.

3.2.3.1 Compatibilité entre deux appariements de nœuds

Deux appariements de nœuds (u_i/u_j) et (u'_i/u'_j) issus des graphes AG_i et AG_j sont compatibles si les nœuds sources de AG_i (u_i et u'_i) et les nœuds destinations de AG_j (u_j et u'_j) sont différents. Cela impose qu'il n'existe pas d'arc dans le graphe de compatibilité entre deux appariements si un nœud appartenant au même graphe est présent dans les deux appariements. Ceci est aussi valable pour la compatibilité entre les autres types d'appariements.

3.2.3.2 Compatibilité entre un appariement de nœuds et d'arcs

La compatibilité entre un appariement de nœuds et d'arcs peut être vu comme la composition de deux compatibilités de nœuds. Ainsi, un appariement de nœuds (u_i/u_j) et d'arcs $(u'_i, v'_i)/(u'_j, v'_j)$ sont compatibles si les deux appariements de nœuds formant l'appariement d'arc $((u'_i, u'_j)$ et $(v'_i, v'_j))$ sont compatibles avec l'appariement de nœuds (u_i/u_j). Cela se traduit par la formule suivante :

$$(u_i \neq u'_i \wedge u_j \neq u'_j) \wedge (u_i \neq v'_i \wedge u_j \neq v'_j)$$

De plus, il y a compatibilité entre un appariement de nœuds et d'arcs, si un des deux appariements de nœuds formant l'appariement d'arc est identique à l'appariement de nœuds. Cela se traduit par la formule suivante :

$$(u_i = u'_i \wedge u_j = u'_j) \vee (u_i = v'_i \wedge u_j = v'_j)$$

3.2.3.3 Compatibilité impliquant un appariement de chemin

La compatibilité impliquant un appariement de chemin est quasi identique à celle utilisée pour un appariement d'arc. La seule différence est que l'on spécifie que tous les nœuds sur le chemin sont différents des nœuds du même AG impliqués dans l'autre appariement. Cela se traduit, dans le cas de deux appariements de type chemin $(u_i, w_i, ..., v_i)/(u_j, v_j)$ et $(u'_i, w'_i, ..., v'_i)/(u'_j, v'_j)$, par la formule suivante :

$$\begin{gathered}[(u_i \neq u'_i \wedge u_j \neq u'_j) \vee ((v_i \neq v'_i \wedge v_j \neq v'_j)] \wedge \\ (\forall_{n \in (u'_i, w'_i, ..., v'_i)} (n \neq u_i) \wedge n \neq v_i) \wedge \\ (\forall_{n \in (u_i, w_i, ..., v_i)} (n \neq u'_i) \wedge n \neq v'_i)\end{gathered}$$

	appariement de nœuds (u'_i/u'_j)	appariement d'arcs $(u'_i, v'_i)/(u'_j, v'_j)$	appariement de chemin $(u'_i, w'_i, ..., v'_i)/(u'_j, v'_j)$
appariement de nœuds (u_i/u_j)	$u_i \neq u'_i \wedge$ $u_j \neq u'_j$	$(u_i = u'_i \wedge u_j = u'_j) \vee$ $(u_i = v'_i \wedge u_j = v'_j) \vee$ $(u_i \neq u'_i \wedge u_j \neq u'_j \wedge$ $u_i \neq v'_i \wedge u_j \neq v'_j)$	$(u_i = u'_i \wedge u_j = u'_j) \vee$ $(u_i = v'_i \wedge u_j = v'_j) \vee$ $[u_i \neq u'_i \wedge u_j \neq u'_j \wedge$ $u_i \neq v'_i \wedge u_j \neq v'_j \wedge (\forall_{n \in (u'_i, w'_i, ..., v'_i)} n \neq u_i)]$
appariement d'arcs $(u_i, v_i)/(u_j, v_j)$		$(u_i \neq u'_i \wedge u_j \neq u'_j) \vee$ $(v_i \neq v'_i \wedge v_j \neq v'_j)$	$[(u_i \neq u'_i \wedge u_j \neq u'_j) \vee ((v_i \neq v'_i \wedge v_j \neq v'_j)] \wedge$ $(\forall_{n \in (u'_i, w'_i, ..., v'_i)} (n \neq u_i) \wedge n \neq v_i)$
appariement de chemin $(u_i, w_i, ..., v_i)/(u_j, v_j)$			$[(u_i \neq u'_i \wedge u_j \neq u'_j) \vee ((v_i \neq v'_i \wedge v_j \neq v'_j)] \wedge$ $(\forall_{n \in (u'_i, w'_i, ..., v'_i)} (n \neq u_i) \wedge n \neq v_i) \wedge$ $(\forall_{n \in (u_i, w_i, ..., v_i)} (n \neq u'_i) \wedge n \neq v'_i)$

TAB. 3.1 – Conditions de compatibilité entre les différents types d'appariements.

3.2.4 Modèle pour la recherche de clique de poids maximum

Pour être capable de déterminer la clique de poids maximum au sein du graphe de compatibilité $GC = (V_c, E_c)$, chaque nœud $u \in V_c$ est modélisé par une variable Sel_u définie sur l'ensemble $\{0, 1\}$. Cette variable Sel_u vaut 1 si le nœud u fait partie de la clique de poids maximum, sinon elle est égale à 0.

Une clique dans le graphe de compatibilité est définie par la contrainte 3.2.1. Cette contrainte impose, pour chaque paire de nœuds non connectés dans le graphe de compatibilité, car non compatibles, qu'ils ne peuvent pas être sélectionnés dans la clique. Donc, forcément une des deux variables *Sel* associées à ces nœuds doit être différente de 1.

Pour déterminer la clique de poids maximum dans le graphe de compatibilité la variable *Sum*, définie par la contrainte 3.2.2, doit être maximisée. Rappelons que le poids $weight(u)$ associé à un appariement, défini par les équations 3.1, 3.2 et 3.3, représente le gain de surface obtenu si l'appariement est sélectionné.

Contrainte 3.2.1

$$\forall (u_c, v_c) \notin E_c : Sel_{u_c} \neq 1 \vee Sel_{v_c} \neq 1$$

Contrainte 3.2.2

$$Sum = \sum_{u \in V_c} Sel_u \cdot weight(u)$$

Suite à ces travaux, pour améliorer l'efficacité de la recherche de clique de poids maximum, une contrainte spécifique (*Clique*) a été développée au sein du solveur JaCoP. Cette contrainte prend deux paramètres, un graphe non-orienté et une variable (à domaine fini) définissant une taille de clique. La contrainte assure que les variables *Sel* associées à chaque nœud du graphe valent 1 si les nœuds concernés font partie d'une même clique. Un mécanisme de propagation spécialement dédié à cette nouvelle contrainte, basé sur l'algorithme proposé par Régin [94], permet d'améliorer l'efficacité de celle-ci par rapport à la contrainte 3.2.1. A titre d'exemple, il n'a fallu qu'environ deux secondes à la contrainte *Clique* pour trouver une clique dans un graphe comportant 2 122 nœuds et 2 116 470 arcs.

Pour assurer la cohérence de la solution, les contraintes 3.2.3 et 3.2.4 sont définies. La première de ces contraintes modélise le fait qu'un appariement d'arcs est sélectionné seulement si les appariements de nœuds source et destination sont aussi sélectionnés. La seconde contrainte modélise la même idée pour un appariemement de chemins. Sans ces contraintes, une connexion entre deux blocs matériels (avec ou sans contournement d'opérateurs) peut être partagée alors que les blocs ne le sont pas, ce qui n'a aucun sens.

Contrainte 3.2.3

$$\forall u_c = (u_i, v_i)/(u_j, v_j) \in V_c : Sel_{u_c} \Leftrightarrow Sel_{(u_i,v_i)} = 1 \wedge Sel_{(u_j,v_j)} = 1$$

Contrainte 3.2.4

$$\forall u_c = (u_i, w_i, ..., v_i)/(u_j, v_j) \in V_c : \textit{If } Sel_{u_c} = 1 \textit{ then } Sel_{(u_i,w_i,...,v_i)} = 1 \wedge Sel_{(u_j,v_j)} = 1$$

La figure 3.9 représente le graphe de compatibilité pour la fusion des deux graphes d'application AG_1 et AG_2. La clique de poids maximum est en gras et les nœuds qui la composent sont grisés. Le graphe fusionné, généré à partir de la clique en gras, est aussi

représenté sur cette figure.

Il est important de noter que plus les deux AG à fusionner sont similaires, tant au niveau structurel qu'au niveau des opérations qu'ils représentent, plus le graphe de compatibilité est grand et dense et donc plus la clique de poids maximum est difficile à déterminer. De plus, l'utilisation du contournement d'opérateurs implique l'ajout des appariements de chemin et donc augmente la taille du graphe de compatibilité.

3.2.5 Modèles pour le problème d'augmentation du chemin critique

Le fait est que la longueur du chemin critique du motif fusionné est rarement égale au plus long chemin critique des motifs à fusionner. Il y a deux raisons qui amènent une augmentation du chemin critique du motif fusionné : la première est le contournement d'opérateur et la seconde l'ajout de multiplexeur. Nous proposons un modèle de contraintes permettant de limiter l'augmentation de la longueur du chemin critique du motif fusionné dans chacun de ces deux cas.

3.2.5.1 Problème d'augmentation du chemin critique par contournement d'opérateur

Quand l'algorithme de fusion utilise des opérateurs contournés, la longueur du chemin critique peut augmenter. C'est le cas, par exemple, pour le second motif AG_2 de la figure présentée précédemment 3.7, dont le chemin critique à été allongé par le nœud contourné en gras représentant un additionneur.

La longueur du chemin critique du motif fusionné est calculé à partir des motifs déjà fusionnés. Ainsi, le chemin critique le plus long parmi les motifs déjà traités correspond au chemin critique du motif fusionné. L'objectif dans notre approche est d'ajouter, au niveau des motifs initiaux déjà fusionnés, le délai correspondant à l'ajout d'un nœud contourné sur un de ses chemins. Cet objectif a été atteint en remplaçant un motif initial par un nouveau motif contenant ce délai. Ainsi, dans ce nouveau motif, chaque arc $(u_j, v_j) \in E_j$ venant d'un nœud de chemin est remplacé par un chemin contenant un nœud additionnel appelé nœud spécial ou *special node* noté SN. Ensuite, une contrainte est utilisée pour limiter la longueur du chemin critique au niveau des motifs déjà fusionnés.

La figure 3.10 montre l'exemple d'un nœud de chemin et une partie du motif modifié en conséquence de cette règle. Dans cet exemple, le nœud t_2 est contourné dans le motif fusionné temporaire pour être appareillé avec l'arc (i_1, i_2) du motif en entrée. Dans ce cas, le retard introduit par le calcul du nœud contourné doit être prise en compte si le nœud de chemin $(t_1, t_2, t_3)/(i_1, i_2)$ est sélectionné dans la clique de poids maximum. Il en résulte que le nouveau motif en entrée contient les mêmes nœuds que dans sa forme originale plus le super nœud SN.

Chaque nœud $u \in V_j$ du nouveau motif est modélisé par deux variables : $Start_u$ et $Delay_u$. La variable $Start_u$ représente la date de début d'exécution de l'opération représentée par le nœud u. La variable $Delay_u$ représente, pour les nœuds réguliers, le retard introduit par le calcul modélisé par u (notée $Latency(u)$). Pour les nœuds des deux autres types $Delay_{SN_u}$ vaut 0 si le nœud chemin correspondant dans le graphe de compatibilité n'est pas sélectionné dans la clique de poids maximum, sinon, cette variable est égale au retard introduit par le chemin correspondant (chemin issu du motif temporaire) comme le

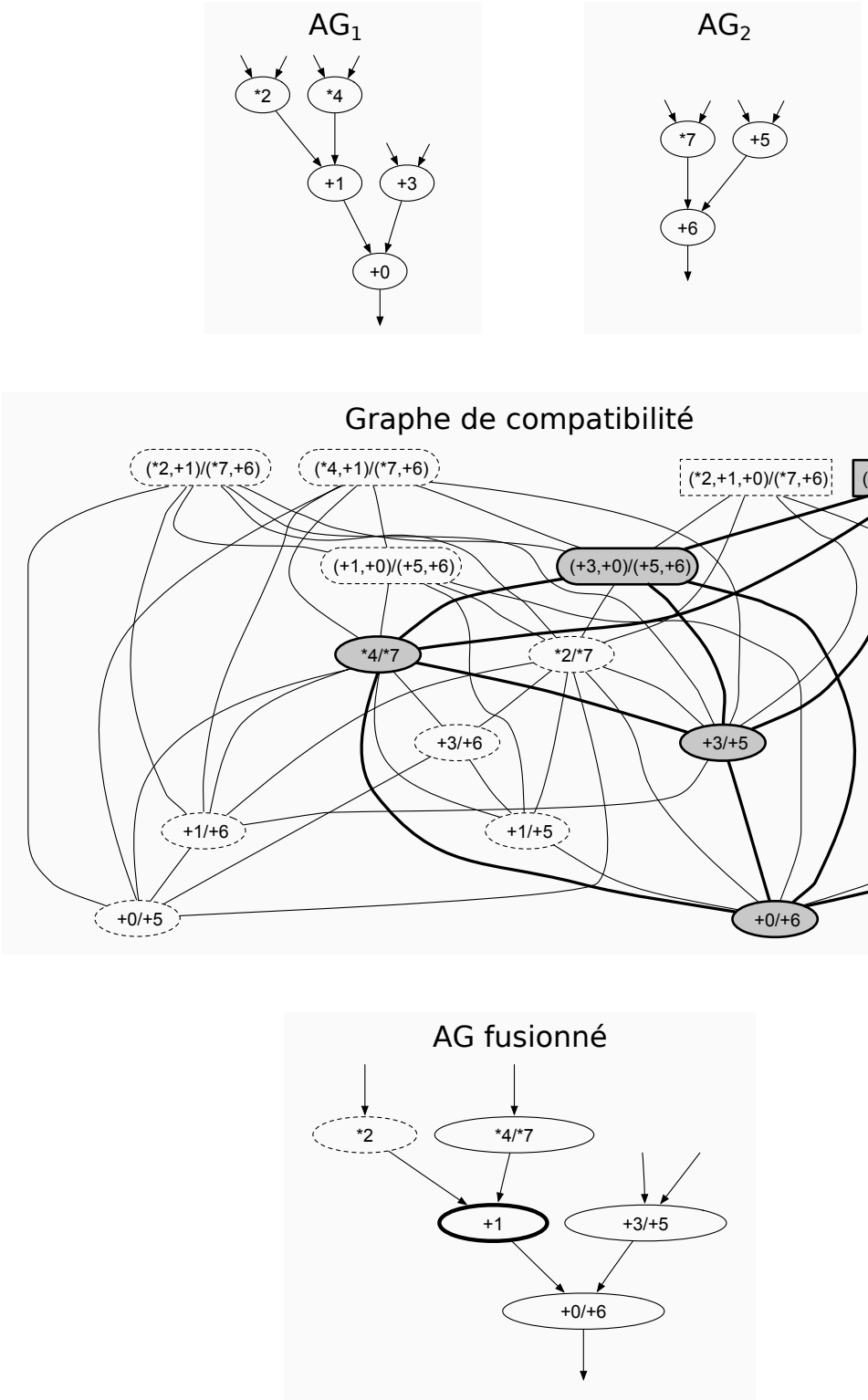


FIG. 3.9 – Graphe de compatibilité pour la fusion de deux graphes d'application, la clique de poids maximum (en gras et grisée) et le graphe d'application fusionné.

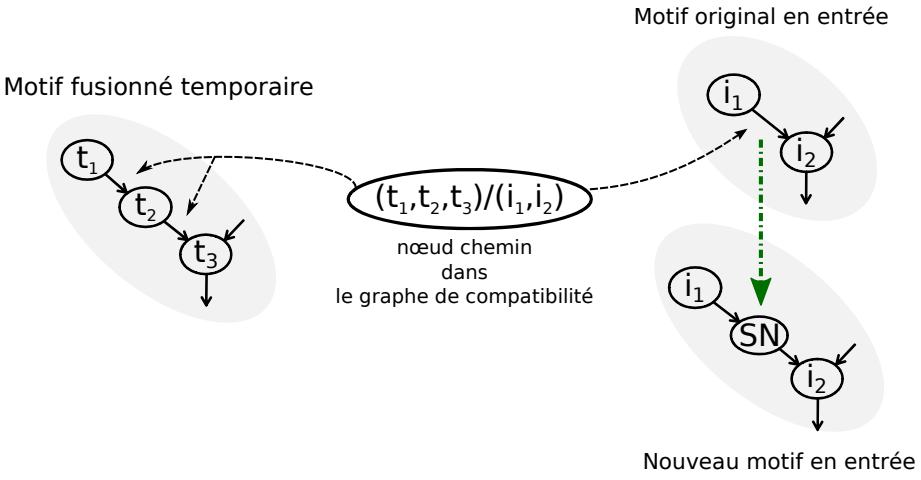


FIG. 3.10 – Exemple d'ajout d'un super nœud SN

définit la contrainte 3.2.5.

Les dépendances de données concernant le nouveau motif sont imposées par la contrainte 3.2.6. La latence du chemin critique est limitée par la contrainte 3.2.7 où ONS est l'ensemble des nœuds de sortie du nouveau motif (ONS signifie *Output Node Set*) et CPL_{max} est la latence limite imposée du chemin critique (CPL signifie *Critical Path Latency*).

Contrainte 3.2.5

$$\forall u \in V_c,\ u = (u_i, w_i, ..., v_i)/(u_j, v_j) :$$
$$Delay_{SN_u} = Sel_u \cdot \sum_{n \in u_i, w_i, ..., v_i} Latency(n)$$

Contrainte 3.2.6

$$\forall (u_j, v_j) \in E'_j : Start_{u_j} + Delay_{u_j} \leq Start_{v_j}$$

Contrainte 3.2.7

$$\forall u \in ONS : Start_u + Delay_u \leq CPL_{max}$$

3.2.5.2 Problème d'allongement du chemin critique par ajout de multiplexeur

Lorsque deux nœuds issus de deux motifs sont appareillés et partagent donc le même nœud dans le motif fusionné, des multiplexeurs peuvent être ajoutés. Cela se produit typiquement si les arcs en entrée de ces nœuds ne sont pas appareillés.

La figure 3.11 montre un exemple de deux appariements. Le premier nécessite un multiplexeur mais pas le second, car il partage l'arc entier. Cela peut aussi arriver que des multiplexeurs soient ajoutés sur le chemin critique. Dans le but de minimiser la latence du motif fusionné, il est nécessaire d'imposer une condition sur le nombre de multiplexeurs sur le chemin critique. Nous avons modélisé cela par l'ajout de nouvelles contraintes.

Tout d'abord, la latence initiale du chemin critique le plus long parmi les motifs en entrée de l'algorithme, CPL_{init}, est calculée pour tous les motifs $P_1, ..., P_k$ de la façon suivante. $Latency(P_i)$ représente le calcul de la latence du motif P_i.

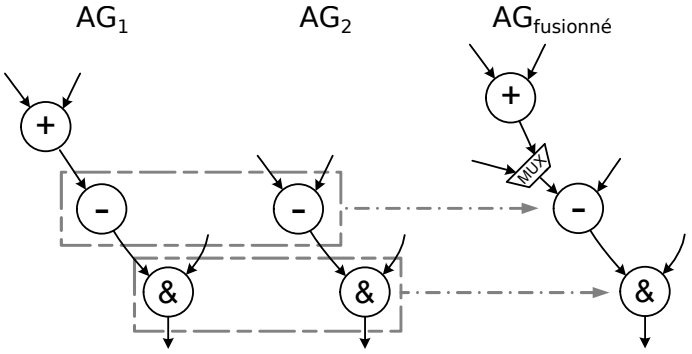


FIG. 3.11 – Exemple d'ajout d'un multiplexeur

$$CPL_{init} = Max\{Latency(P_1), ..., Latency(P_k)\},$$

Ensuite, à chaque itération de l'algorithme on compare la longueur du chemin critique passé en paramètre, CPL_{max}, avec la latence de chaque motif déjà fusionné ainsi que le motif à fusionner. Puis, pour chaque motif P_i avec $Latency(P_i) = CPL_{max}$, un ensemble $SNCP_i$ est généré. Cet ensemble contient tous les nœuds se trouvant sur le chemin critique du motif P_i ($SNCP$ signifie *Set of Node on Critical Path*).

A noter que CPL_{max} ne peut pas être plus petit que CPL_{init}. De plus, CPL_{max} ne correspond pas exactement à la longueur du chemin critique à ne pas dépasser car celle-ci peut augmenter en fonction du nombre de multiplexeurs pouvant être ajoutés, noté MNM (*Maximum Number of Multiplexer*). La longueur maximale du chemin critique est exactement égale à $CPL_{max} + MNM * Delay_{mux}$ où $Delay_{mux}$ représente le délai de traversée d'un multiplexeur 2 vers 1.

Pour exprimer les conditions sur les multiplexeurs, deux autres ensembles sont définis pour chaque nœud $u_i \in SNCP_i$. Ces ensembles, notés SSN_{u_i} et SSN'_{u_i} (SSN signifie *Set of Selected Nodes*), contiennent les variables Sel_{u_i} associées aux nœuds du graphe de compatibilité GC et sont générés par l'algorithme 1. Si au moins une variable de l'ensemble SSN_{u_i} est égale à 1, cela signifie que le nœud correspondant dans le GC a été sélectionné et le nœud u_i est partagé dans le motif fusionné résultant. De manière similaire, si la valeur d'une variable de l'ensemble SSN'_{u_i} vaut 1, cela signifie que le nœud correspondant dans le GC a été sélectionné et que l'arc entier est fusionné (le nœud u_i est le nœud destination dans l'arc fusionné).

Chaque nœud $u_i \in SNCP_i$ est modélisé par deux variables : $NbMuxEntre_{u_i}$ et Mux_{u_i}. La variable $NbMuxEntre_{u_i}$ représente le nombre de multiplexeurs sur le chemin critique avant u_i. La variable multiplexeur Mux_{u_i} est définie par la contrainte 3.2.8. La valeur de cette variable est égale à 1 si le multiplexeur est ajouté, 0 sinon.

La première condition de la contrainte 3.2.8 spécifie qu'un multiplexeur existe déjà. En effet, nous déterminons l'existence d'un multiplexeur en fonction de la variable notée TNM_{u_i} (*Temporary Number of Multiplexer*). Cette variable représente le nombre temporaire de multiplexeurs sur les entrées d'un nœud u_i se trouvant sur le chemin critique dont la longueur ne peut augmenter. Elle est affectée pendant l'étape de mise à jour de l'algorithme. Si un multiplexeur a déjà été ajouté en entrée du nœud u_i, alors TNM_{u_i} est supérieure à 0, sinon TNM_{u_i} vaut 0.

Algorithme 1 Génération des ensembles SSN

// $SNCP_i$: ensemble de nœuds sur le chemin critique
// SSN : ensemble de nœuds réguliers sélectionnés dans le graphe de compatibilité
// SSN' : ensemble de nœuds d'arc et de chemin sélectionnés dans le graphe de compatibilité
// V_c : ensemble de nœuds du graphe de compatibilité
pour tout u_i tel que $u_i \in SNCP_i$ **faire**
  $SSN_{u_i} = \emptyset,\ SSN'_{u_i} = \emptyset$
  pour tout u tel que $u \in V_c$ **faire**
    si $(u = u_i/u_j)$ **alors**
      $SSN_{u_i} = SSN_{u_i} \cup \{Sel_u\}$
    sinon si $u = (v_i, u_i)/(u_j, v_j) \wedge v_i \in SNCP_i$ **alors**
      $SSN'_{u_i} = SSN'_{u_i} \cup \{Sel_u\}$
    sinon si $u = (v_i, w_i, .., u_i)/(u_j, v_j) \wedge v_i \in SNCP_i$ **alors**
      $SSN'_{u_i} = SSN'_{u_i} \cup \{Sel_u\}$
    fin si
  fin pour
fin pour

La deuxième condition signifie qu'il n'y a pas de multiplexeur pour le moment ($TNM_{u_i} = 0$), de plus, le nœud u_i est impliqué dans un appariement de nœuds mais pas d'arc ni de chemin ($SSN_{u_i} \neq \emptyset \wedge SSN'_{u_i} = \emptyset$). Dans ce cas, un multiplexeur sera ajouté si au moins un des appariements de nœuds impliquant u_i dans le graphe de compatibilité est sélectionné. Cette condition est modélisée par la variable booléenne R_1.

La troisième condition définit une situation où un multiplexeur n'existe pas et u_i est impliqué à la fois dans un appariement de nœuds et d'arcs (ou de chemin). Formellement cela se traduit par $TNM_{u_i} = 0 \wedge SSN_{u_i} \neq \emptyset \wedge SSN'_{u_i} \neq \emptyset$. Ici c'est la variable booléenne R_2 qui définit l'ajout d'un multiplexeur, elle vaut 1 si $R_1 = 1$ et si aucun des appariements d'arc ou de chemin dont u_i est le nœud destination n'est sélectionné (c.-à-d. lorsque la variable booléenne $R = 1$).

La dernière condition exprime le cas où il n'y a pas de multiplexeur et il ne sera pas nécessaire d'en ajouter un car u_i n'est pas appareillé avec un autre nœud. En effet, l'ensemble SSN_{u_i} est vide ce qui signifie qu'il n'existe pas de nœud de la clique impliquant u_i.

Contrainte 3.2.8

$$Mux_{u_i} = \begin{cases} 1 & si\ TNM_{u_i} > 0 \\ R_1 & si\ TNM_{u_i} = 0 \wedge SSN_{u_i} \neq \emptyset \wedge SSN'_{u_i} = \emptyset \\ R_2 & si\ TNM_{u_i} = 0 \wedge SSN_{u_i} \neq \emptyset \wedge SSN'_{u_i} \neq \emptyset \\ 0 & si\ TNM_{u_i} = 0 \wedge SSN_{u_i} = \emptyset \end{cases}$$

$$R_1 \Leftrightarrow \sum_{Sel \in SSN_{u_i}} Sel > 0$$

$$R_2 = R_1 \cdot R,\ where\ R \Leftrightarrow \sum_{Sel \in SSN'_{u_i}} Sel = 0$$

Un exemple simple reprenant les deux motifs de la figure 3.11 permet de mieux appréhender la définition de la variable Mux_{u_i}, il est illustré sur la figure 3.12. Dans cet exemple, le chemin critique maximum autorisé est égal au chemin critique du motif AG_1, soit $CPL_{max} = CPL_{init} = Latency(AG_1)$, donc $SNCP_1 = \{u_1, v_1, w_1\}$. Le graphe de compatibilité est représenté sur la figure, tous les nœuds de ce graphe sont compatibles entre eux, ils sont donc reliés par des arcs.

A partir de l'ensemble des nœuds sur le chemin critique, $SNCP_1$, les deux ensembles SSN et SSN' sont créés pour chacun de ces nœuds. Les contenus de ces ensembles après avoir exécuté l'algorithme 1 sont les suivants :

$$SNCP_1 = u_1, v_1, w_1$$

$$u_1 \left\| \begin{array}{l} SSN_{u_1} = \emptyset \\ SSN'_{u_1} = \emptyset \end{array} \right. \qquad v_1 \left\| \begin{array}{l} SSN_{v_1} = \{Sel_{cn_1}\} \\ SSN'_{v_1} = \emptyset \end{array} \right. \qquad w_1 \left\| \begin{array}{l} SSN_{w_1} = \{Sel_{cn_2}\} \\ SSN'_{w_1} = \{Sel_{cn_3}\} \end{array} \right.$$

Maintenant, nous allons étudier pour cet exemple les variables Mux_{u_1}, Mux_{v_1} et Mux_{w_1} qui permettent de déterminer si il y a ajout de multiplexeur sur le chemin critique (u_1, v_1, w_1). L'hypothèse que nous faisons est qu'il n'existe pas déjà de multiplexeur sur ce chemin donc le nombre temporaire de multiplexeurs en entrée des nœuds sur le chemin critique vaut 0 ($TNM_{u_1} = TNM_{v_1} = TNM_{w_1} = 0$).

- Concernant le nœud u_1, comme il n'est appareillé à aucun nœud de l'AG_2 ($SSN_{u_1} = \emptyset$) et qu'il n'y a pas déjà de multiplexeur sur une de ses entrées ($TNM_{u_1} = 0$) alors d'après la contrainte 3.2.8 $Mux_{u_1} = 0$ donc aucun multiplexeur ne peut être ajouté en entrée de u_1.

- Pour le nœud v_1, il est impliqué dans l'appariement de nœuds cn_1 donc $SSN_{v_1} \neq \emptyset$ mais dans aucun appariement d'arc ni de chemin donc $SSN'_{v_1} = \emptyset$. Selon la contrainte 3.2.8, la variable Mux_{v_1} est donc égale à R_1. Dans ce cas, un multiplexeur est ajouté en entrée de v_1 si l'appariement de nœuds cn_1 est sélectionné ($Sel_{cn_1} = 1$), ce qui signifie en fait que dans l'AG fusionné, v_1 est fusionné mais pas ses arcs d'entrée.

- Pour le nœud w_1, il est impliqué dans l'appariement de nœuds cn_2 et d'arcs cn_3 donc $SSN_{w_1} \neq \emptyset$ et $SSN'_{w_1} \neq \emptyset$. Selon la contrainte 3.2.8, la variable Mux_{w_1} est donc égale à R_2. Dans ce cas, un multiplexeur est ajouté en entrée de w_1 dans le cas où l'appariement de nœuds cn_2 est sélectionné mais pas l'appariement d'arcs cn_3 ($Selcn_2 = 1 \wedge Selcn_3 = 0$). Dans tous les autres cas de figure, aucun multiplexeur n'est ajouté.

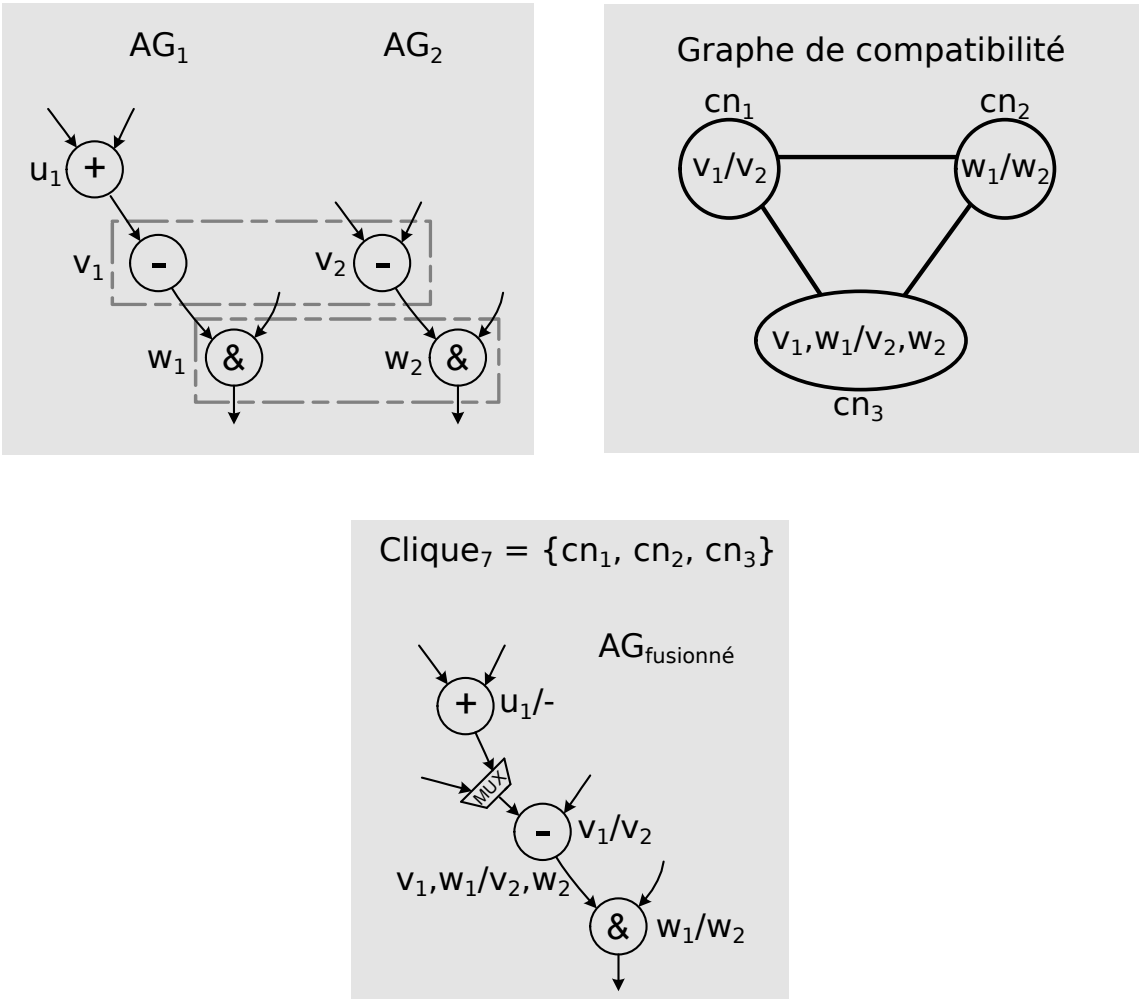


FIG. 3.12 – Exemple illustrant le modèle de contraintes pour le problème d'augmentation du chemin critique par ajout de multiplexeur. Le modèle permet de sélectionner la $Clique_7$ si au moins un multiplexeur peut être ajouté sur le chemin critique $\{u_1, v_1, w_1\}$.

Dans le but de satisfaire la condition relative au nombre de multiplexeurs sur le chemin critique, les contraintes 3.2.9 et 3.2.9 sont imposées. La contrainte 3.2.9 prend en compte la dépendance de données entre les nœuds contenus dans l'ensemble $SNCP_i$. Ces contraintes permettent de calculer le nombre de multiplexeurs sur le chemin critique du motif P_i. La contrainte 3.2.10 limite le nombre de multiplexeurs. ONS' représente l'ensemble des nœuds de sortie (*Output Node Set*) dans $SNCP_i$ et MNM le nombre maximum de multiplexeurs (*Maximum Number of Multiplexer*) admis sur le chemin critique du motif P_i.

Contrainte 3.2.9

$$\forall u_i, v_i \in SNCP_i, (u_i, v_i) \in E_i : NbMuxEntre_{u_i} + Mux_{u_i} \leq NbMuxEntre_{v_i}$$

Contrainte 3.2.10

$$\forall u \in ONS' : NbMuxEntre_u + Mux_u \leq MNM$$

Si nous reprenons l'exemple précédent et que l'on spécifie qu'aucun multiplexeur ne doit être ajouté sur le chemin critique (MNM vaut 0), alors notre modèle de contraintes assure qu'aucune clique ne sera sélectionnée. En conséquence, les deux AG ne partagent aucun nœud, ni aucun arc. En effet, il y a sept cliques dans le graphe de compatibilité de la figure 3.12 :

- $Clique_1 = \{cn_1\}$
- $Clique_2 = \{cn_2\}$
- $Clique_3 = \{cn_3\}$
- $Clique_4 = \{cn_1, cn_2\}$
- $Clique_5 = \{cn_1, cn_3\}$
- $Clique_6 = \{cn_2, cn_3\}$
- $Clique_7 = \{cn_1, cn_2, cn_3\}$

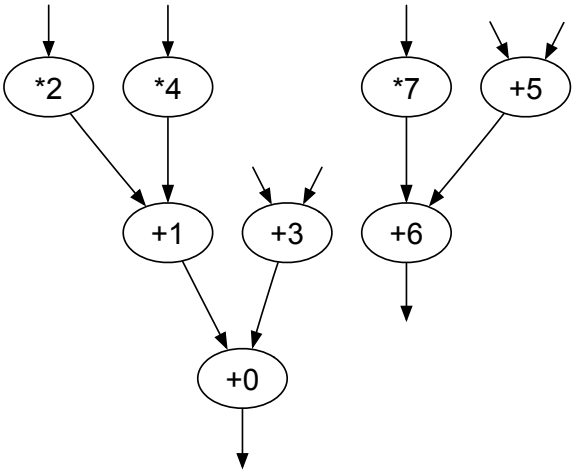
Dans les cas où cn_1 est sélectionné, un multiplexeur est ajouté en entrée de v_1, donc les cliques 1, 4, 5 et 7 ne peuvent pas être sélectionnées. La clique 2 ne peut pas l'être non plus car cela implique l'ajout d'un multiplexeur en entrée de w_1. Enfin les cliques 3 et 6 ne sont pas sélectionnées car elles ne remplissent pas la contrainte 3.2.3 qui impose qu'un appariement d'arcs ne peut être sélectionné que si les deux appariements de nœuds qui le composent le sont aussi. En d'autres termes, si cn_3 est sélectionné, alors cn_1 et cn_2 le sont aussi.

Si par contre, on spécifie qu'un multiplexeur peut être ajouté sur le chemin critique (MNM vaut 1), alors il est clair que c'est la clique 7 qui est sélectionnée car son poids est forcément le plus grand.

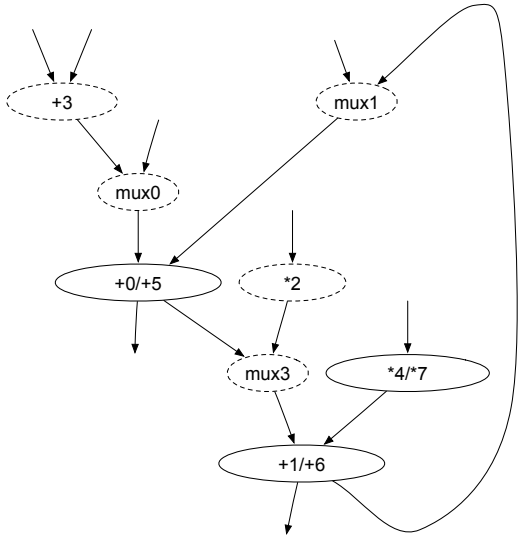
3.3 Résultats

Le modèle de contraintes pour la fusion de chemins de données présenté permet d'obtenir des résultats aussi bons, voire meilleurs, que ceux présentés par Moreano dans [81] et cela notamment grâce à l'utilisation et à la généralisation de la notion de contournement d'opérateur mais surtout par la modélisation de contraintes sur l'ajout de multiplexeur. Pour illustrer cela, prenons un exemple représentatif de l'amélioration apportée par notre approche (figure 3.13). Dans cet exemple on peut observer la différence entre le résultat obtenu par l'approche de Moreano (3.13(b)) et la nôtre (3.13(c)). Cet expérimentation a été effectuée avec les contraintes suivantes :

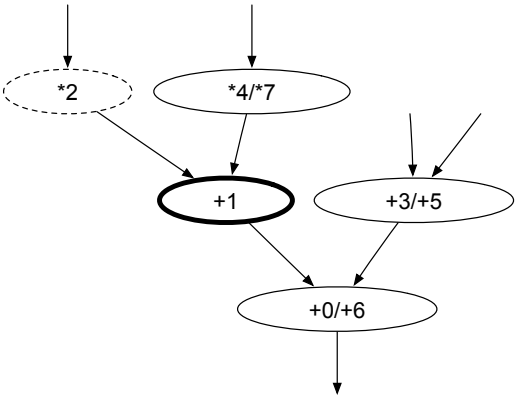
– la longueur du chemin critique ne doit pas augmenter,
– aucun multiplexeur ne peut être ajouté sur le chemin critique,
– il est possible de contourner un nœud (le nœud +1 est effectivement contourné dans le chemin de données fusionné).



(a) Chemin de données à fusionner



(b) Résultat obtenu avec l'approche de Moreano



(c) Résultat obtenu avec notre système

FIG. 3.13 – Comparaison entre l'approche de Moreano et la notre : exemple représentatif.

Dans le but d'évaluer la qualité de notre méthode de fusion de chemins de données nous avons procédé à plusieurs expérimentations dont les résultats ont été présentés à la conférence internationale DSD (*Digital System Design*) en 2009 [118]. Toutes les expérimentations ont été exécutées sur un processeur Intel Core Duo à 2GHz sous le système d'exploitation Mac OSX. Les expérimentations ont été faites en deux temps.

3.3.1 Comparaison avec l'approche de Moreano

Dans un premier temps, nous avons comparé les résultats obtenus avec notre approche avec ceux présentés par [81].

Le tableau 3.3 rassemble les résultats détaillés pour trois expérimentations faites sur l'application *EPIC decoder* issue de [81]. Cette application est représentée par trois AG comportant chacun une dizaine de nœuds et d'arcs. Ces expérimentations ont été effectuées en utilisant différentes options d'optimisation (données dans le tableau 3.2). Elles illustrent la possibilité offerte par notre système d'effectuer une exploration de l'espace de conception sous contraintes de synthèse d'architecture.

La première expérimentation (Exp. 1) correspond à celle que l'on trouve à l'origine dans [81] où seuls les nœuds et les arcs (N et E) sont partagés. Dans ce cas, notre système obtient les mêmes résultats que dans [81].

Dans la deuxième expérimentation (Exp. 2), l'option de partage des chemins (utilisant le contournement d'opérateur) a été utilisée acceptant ici un maximum de deux nœuds contournés sur un chemin (P=2). Cela permet de réduire le nombre de multiplexeurs de 40% et le nombre d'arcs de 16%.

Dans la dernière expérimentation (Exp. 3), le chemin critique ne peut pas augmenter à cause de l'insertion de nœuds contournés (CP=Yes) et le nombre de multiplexeurs est limité à deux sur ce chemin (NM=2). Le motif fusionné résultant comporte en plus deux nœuds, un arc et un multiplexeur comparé à la deuxième expérimentation.

Expérimentation	N	E	P	CP	NM
Exp. 1	√	√			
Exp. 2	√	√	2		
Exp. 3	√	√	2	√	2

TAB. 3.2 – Détail des expérimentations effectuées sur l'application *EPIC decoder*.

Le système permet d'obtenir des cliques de poids maximum optimales. Par ailleurs, il prouve leur optimalité.

Expérimentation		Graphe de compatibilité				Appariements sélectionnés dans la clique de poids max.			Temps d'exécution (seconde)	Caractéristiques du motif fusionné			Gain en surface (en %)
		Appariements			Arcs								
		de nœuds	d'arcs	de chemin		de nœuds	d'arcs	de chemin		Nœuds	Arcs	Mux	
Exp. 1	fusion 1	31	15	0	858	10	6	0	1.02	18	30	5	48
	fusion 2	49	31	0	2717	15	9	0	0.85				
Exp. 2	fusion 1	31	15	8	1151	10	6	1	3.71	18	25	3	50
	fusion 2	49	28	12	3225	15	9	2	1.24				
Exp. 3	fusion 1	31	15	2	1151	9	6	0	1.18	20	29	4	45
	fusion 2	49	28	12	3225	13	8	1	2.15				

TAB. 3.3 – Résultats pour l'application *"EPIC decoder"* issue de [81].

3.3.2 Résultat sur un ensemble d'applications multimédia

Dans un second temps, nous avons réalisé des expérimentations sur des ensembles de motifs identifiés par notre système sur les applications de traitement du signal numérique

de la suite de tests MediaBench [68].

Les graphiques (3.14, 3.16, 3.17) montrent les différents résultats obtenus pour l'ensemble de motifs identifiés par notre système UPaK[1] pour les applications DSP de la suite de tests Mediabench [68].

Pour chaque application venant de la suite de tests Mediabench, cinq expérimentations ont été effectuées avec différentes options comme le montre le tableau 3.4 (N, E, P, CP et NM sont les mêmes que précédemment).

Expérimentation	N	E	P	CP	NM
Exp. 1	√				
Exp. 2	√	√	2		
Exp. 3	√	√			0
Exp. 4	√	√	2	√	0
Exp. 5	√	√	2	√	2

TAB. 3.4 – Détail des expérimentations effectuées sur les applications multimédia sélectionnées.

Tout d'abord, le graphique 3.14 donne les gains en surface par rapport à la surface de l'ensemble des chemins de données à fusionner. Les surfaces ont été obtenues après une synthèse logique avec comme cible un FPGA Altera Stratix2 (EP2560 FC675 -4). Ces surfaces sont données en nombre d'atomes combinatoires (noté CA) pour des opérateurs sur des données de 32 bits. Entre parenthèses, après chaque nom d'application, est donné le nombre de chemins de données fusionnés.

Le graphique 3.15 montre les gains moyens en surface (sans la surface de l'interconnexion) pour les cinq expérimentations. Toutes les cliques trouvées durant la fusion de motifs ont été prouvées comme étant de poids maximum optimal.

Une première analyse de ces résultats démontre qu'en moyenne le meilleur gain en surface est obtenu lorsque l'on effectue tous les types d'appariements possibles par l'expérimentation 2 qui obtient 67,67% de gain. Le plus faible gain, de 41.07%, est obtenu avec l'expérimentation 3 dans laquelle aucun contournement d'opérateur, ni aucun ajout de multiplexeur sur le chemin critique n'est possible. Entre ces deux extrêmes, se trouve l'expérimentation 1 dans laquelle seuls les nœuds peuvent être appareillés sans aucune restriction sur la dégradation des performances temporelles du chemin de données fusionné. L'expérimentation 4 est la plus contraignante car elle combine le plus grand nombre de contraintes, celle de ne pas augmenter la longueur du chemin critique ni par le contournement d'opérateur (même si il est possible d'utiliser cette technique sur d'autre chemin), ni par l'ajout de multiplexeur. Néanmoins, un gain moyen en surface de 50,47% est obtenu avec cette expérimentation. La cinquième et dernière expérimentation est la plus intéressante car elle montre qu'il est possible d'obtenir en moyenne des gains en surface presque aussi bon qu'avec l'expérimentation 2 et meilleurs qu'avec l'expérimentation 1 tout en assurant que la longueur du chemin critique ne peut augmenter, au plus, de 2 multiplexeurs.

Le graphique 3.16 permet de mieux comprendre les résultats de surface en fonction des expérimentations en donnant le nombre d'arcs dans le chemin de données fusionné.

[1] *Abstract Unified Patterns Based Synthesis Kernel for Hardware and Software Systems*, voir 2.3.4.1.

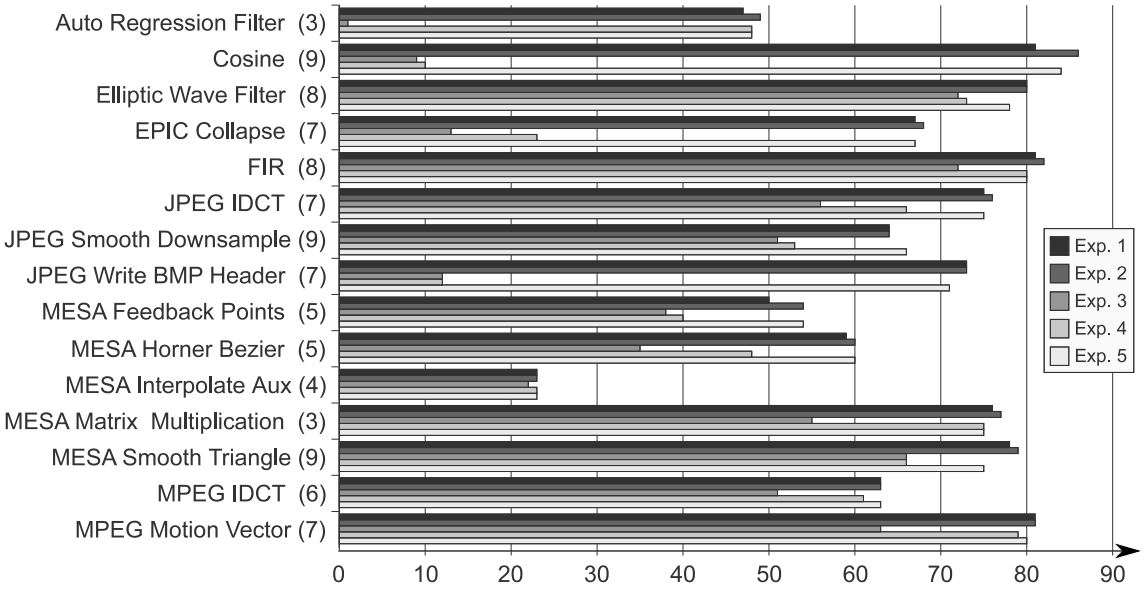


FIG. 3.14 – Gains en surface obtenus (en %) pour le FPGA Stratix2 EP2560 FC675 -4.

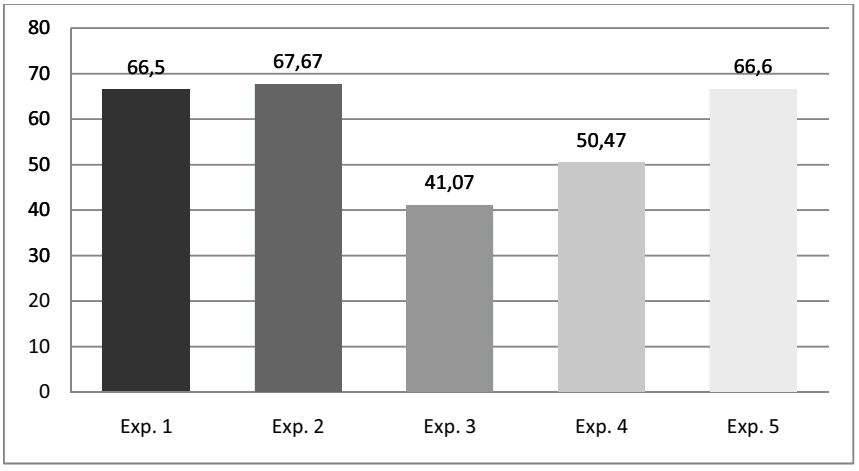


FIG. 3.15 – Gains moyens en surface en % (sans la surface de l'interconnexion) pour les cinq expérimentations pour le FPGA Stratix2 EP2560 FC675 -4.

Il est clair que la différence de gain en surface entre les expérimentations 3 et 4 est due principalement à la réduction du nombre d'arcs par le contournement d'opérateur et cela tout en assurant que cette réduction n'induit pas de perte de performance. Cette différence représente l'intérêt de notre approche qui minimise la surface lors de la fusion en utilisant le contournement d'opérateur mais sous contrainte de ne pas augmenter la longueur du chemin critique.

On peut observer également que le nombre d'arcs dans le chemin de données fusionné, représentant l'interconnexion entre opérateurs, est très lié à l'inverse du gain en surface obtenu. Globalement, on peut en déduire que plus le partage des connexions est important (en utilisant notamment le contournememt d'opérateur), moins le graphe fusionné comporte d'arc, et meilleur est le gain en surface.

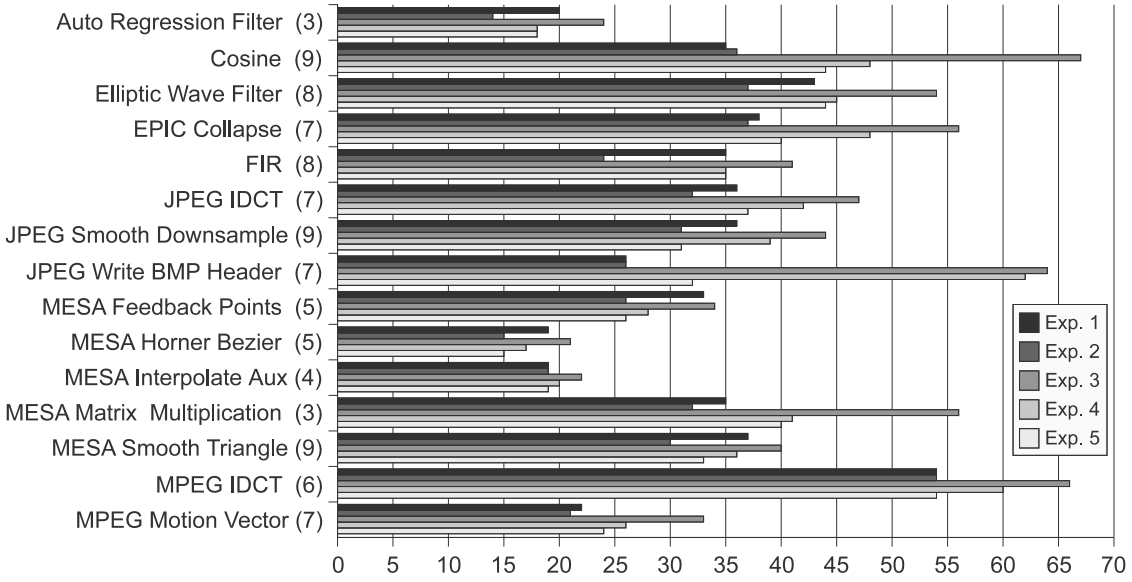


FIG. 3.16 – Nombre d'arcs dans le chemin de données fusionné

Enfin, les temps d'exécution du solveur pour déterminer les solutions optimales pour la plus restrictive des expérimentations (Exp. 4) sont donnés par la figure 3.17 (incluant le temps pour prouver l'optimalité des solutions). Le temps d'exécution moyen est de seulement 0,1 seconde et ne dépasse pas les 300 ms. Ces temps d'exécution ont été obtenus en utilisant la contrainte spécifique *Clique* et le mécanisme de propagation associé qui permet une résolution du problème de clique de poids maximum optimal dans des temps acceptables même pour de grands graphes de compatibilité.

3.4 Conclusion et perspectives

L'état de l'art sur la fusion de chemin de données montre un grand intérêt pour cette technique dans le cadre de la conception d'accélérateur reconfigurable, en particulier à gros grain. Cette technique peut s'inscrire dans les deux flots de conception et de compilation pour ASIP, DURASE et UPaK, présentés au paragraphe 2.3.4. Dans ce contexte, la fusion de chemin de données est utilisée pour synthétiser une ou plusieurs unités reconfigurables au niveau fonctionnel étendant ainsi le jeu d'instructions d'un processeur à usage général

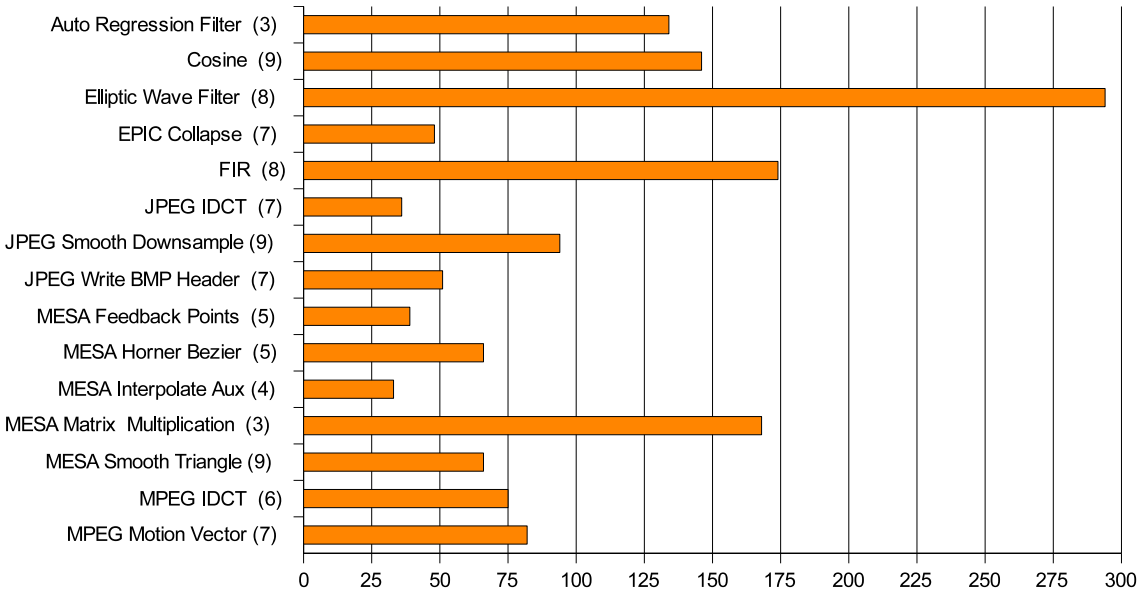


FIG. 3.17 – Temps d'exécution (en ms) de l'algorithme de fusion de motifs en ms pour l'expérimentation la plus restrictive (Exp. 4).

par des instructions spécifiques à l'application à accélérer.

La littérature au sujet de la fusion de chemins de données propose des solutions efficaces. Celles-ci permettent de minimiser la surface du chemin de données fusionné comme celle de Moreano [81] que nous avons pris comme point de départ dans nos travaux, car c'est la solution la plus aboutie et qui s'inscrit parfaitement dans notre approche. Les travaux de Corazao [30], qui proposent le concept de contournement d'opérateur, permettent de réduire l'ajout de multiplexeur lors de la fusion et donc d'accroitre le gain en surface et la performance du résultat.

Mais il subsiste une lacune dans les différentes approches proposées. Seule la minimisation de la surface est visée, aucune contrainte n'est imposée quand à la performance du chemin de données résultant de la fusion. Or, ceci pose problème dans le cadre de la conception d'une extension reconfigurable de processeur. En effet, il est intéressant d'assurer par exemple que l'extension ne ralentisse pas le processeur, c.-à-d. que la longueur du chemin critique d'une extension purement combinatoire ne dépasse pas celle du processeur.

Nous avons proposé dans ce chapitre une méthode basée sur celles de Moreano et Corazao et sur la programmation par contrainte qui comble cette lacune. Notre système permet, grâce à une méthodologie basée sur la CP, de résoudre le problème de fusion de chemins de données en maximisant le gain en surface sous contraintes de performances. Pour cela, nous avons modélisé le problème de recherche de la clique de poids maximum sous forme d'une contrainte associée à son mécanisme de propagation qui est intégrée dans le solveur JaCoP. Le modèle de contraintes proposé intègre et généralise le concept de contournement d'opérateur. Ce modèle comporte aussi les contraintes permettant de limiter l'augmentation de la longueur du chemin critique de la solution par le contournement d'opérateur(s) et/ou par l'ajout de multiplexeur(s).

Les résultats de nos expérimentations sur la suite de test Mediabench sont prouvés optimaux. Ils démontrent que notre approche est meilleure que celle de Moreano en termes de surface grâce au concept de contournement d'opérateur. Le plus important est le fait que nous obtenons des cellules reconfigurables qui respectent les contraintes de performances imposées avec un gain entre 41,07% et 67,67% selon les contraintes. Enfin, nous avons montré la capacité de notre système à déterminer une solution en moins d'une demie seconde et cela même pour la résolution de problèmes fortement contraints.

Un paradoxe subsiste néanmoins dans l'utilisation d'un graphe de compatibilité. En effet, plus les deux chemins de données à fusionner sont similaire, plus le graphe de compatibilité est grand et dense. Cela conduit à une augmentation de la complexité à déterminer la clique de poids maximum.

Les perspectives envisagées à court terme sont, d'une part, l'appariement des nœuds d'entrée/sortie qui ne sont pas modélisés pour le moment, ce qui permettrait de contourner les opérateurs en entrée/sortie. D'autre part, il est possible de simplifier le graphe de compatibilité, et donc la recherche de clique de poids maximum, en y excluant les nœuds qui représentent les appariements d'arcs et de chemins. En effet, les contraintes 3.2.3 et 3.2.4 assurent déjà la prise en compte des gains de surface issus de la sélection de ces appariements. Cette seconde perspective permettrait de fusionner des chemins de données plus grands ou avec de plus fortes contraintes technologiques (longueur du chemin critique, consommation énergétique, etc.).

La principale perspective à plus long terme est la possibilité de fusionner un ensemble de chemin de données en plusieurs cellules reconfigurables. Il est possible d'ajouter la minimisation du temps d'exécution de l'application lors de l'ordonnancement (sur l'architecture comportant les cellules fusionnées). Ce problème peut être modélisé en un seul problème d'optimisation multi-objectif ou en un problème de minimisation de la surface sous contrainte de temps d'exécution (ou le contraire). Ce type de système peut s'inscrire dans le contexte des applications embarquées temps-réel sur architecture VLIW par exemple.

D'autres perspectives intéressantes correspondent à l'ajout de contraintes sur la largeur des données, ou encore la synthèse de cellules reconfigurables pipelinées voir de type SWP.

Enfin, il subsiste certains aspects, en particulier le paradoxe sur la taille et la densité du graphe de compatibilité décrit précédemment. Cela pourrait éventuellement remettre en cause l'approche utilisée dans le cas de la fusion de grand graphes quasi-similaires.

Chapitre 4

Modèles de contraintes pour le déploiement d'applications sur CGRA

4.1 Introduction

Le problème du déploiement d'une application sur une CGRA[1] consiste à déterminer sur quelle ressource et à quel moment seront effectuées les opérations de calcul et les transferts de données nécessaires à l'exécution de l'application. Ce problème se décompose communément en trois sous-problèmes : l'ordonnancement des opérations, l'allocation de ressources et le routage des transferts de données sur le(s) réseau(x) d'interconnexion. Le terme de compilation est aussi communément employé pour désigner cela.

Dans de nombreux travaux [32, 49, 59], ces sous-problèmes (ou un sous-ensemble) sont résolus séparément et cela parce qu'ils sont chacun très complexes à résoudre, ils sont connus pour être NP-complets. Dans notre approche basée sur la CP[2], nous avons cherché à résoudre le problème de déploiement en une unique étape d'optimisation. Ce choix a été fait car ces sous-problèmes étant fortement corrélés, la recherche d'une solution optimale est difficilement approchable en les adressant séquentiellement. Dans une approche itérative, le risque de rester bloqué sur un minimum local ou d'itérer à l'infini est bien connu.

Plusieurs travaux antérieurs ont proposé de résoudre simultanément les différents problèmes sous-jacents au déploiement d'une application. Brenner propose une méthode basée sur la programmation linéaire en nombre entier (ILP pour *Integer Linear Prgramming*) pour résoudre de manière optimale le déploiement d'un graphe de flot de données sur une CGRA [16]. L'architecture ciblée est formée de processeurs élémentaires identiques connectés par un réseau en *mesh* où chaque processeur peut communiquer avec ses voisins. Une formulation par programmation linéaire mixte (MIP pour *Mixed Integer Programming*), proposée par Sadiq dans [96], adresse le problème pour des algorithmes de traitement du signal et une architecture VLIW[3] mais la problématique de routage n'est pas adressée.

[1] *Coarse Grain Reconfigurable Architecture*, architecture reconfigurable à gros grain.
[2] *Constraint Programming*, programmation par contraintes.
[3] *Very Long Instruction Word*, processeur adressant plusieurs unités fonctionnelles en parallèle dans la même instruction, exploitant ainsi le parallélisme d'instruction.

D'autres approches basées sur des heuristiques (ou métaheuristiques) comme l'utilisation d'algorithmes génétiques [98, 107] ou plus récemment l'utilisation d'un algorithme de colonies de fourmis [40] permettent d'obtenir de très bons résultats pour des graphes de petite taille. Le niveau d'abstraction des graphes utilisés varie entre le graphe de tâches et le graphe de flot de données. Une étude comparative présentée dans [89] permet d'appréhender les avantages et inconvénients de trois techniques phares, le *list scheduling*, la CP et un algorithme génétique. Même si le problème de routage des données sur un ou plusieurs réseaux d'interconnexion n'est pas adressé, cette étude montre que le *list scheduling* est efficace mais peu précis, que la CP permet d'obtenir des résultats prouvés optimaux mais au prix d'une recherche plus longue et enfin que l'algorithme génétique a une efficacité raisonnable avec une meilleure précision que le *list scheduling*.

Le contribution présentée dans ce chapitre permet, pour la première fois, de résoudre simultanément les problèmes de déploiement (ordonnancement, allocation de ressources et routage) d'un graphe de flot de données pour un modèle de CGRA générique en utilisant la CP. Notre approche permet aussi bien la génération de configuration que l'exploration de l'espace de conception.

Flot de déploiement proposé

Pour résoudre ce problème de déploiement avec la CP, il est nécessaire de disposer d'un modèle d'architecture et d'une représentation de l'application. Dans notre approche, l'architecture cible est modélisée sous la forme d'un ensemble de paramètres décrivant ses différentes caractéristiques et l'application est représentée sous la forme d'un graphe de flot de données.

Le résultat du déploiement est un graphe de flot de données déployé, c.-à-d. qu'il comporte toutes les informations nécessaires à son exécution sur l'architecture modélisée. Dans les travaux présentés dans ce manuscrit, nous avons choisi d'optimiser le temps d'exécution mais il est possible avec notre approche basée sur la CP d'optimiser d'autres aspects comme le nombre de configurations ou encore plusieurs aspects en même temps.

La figure 4.1 représente le flot générique de déploiement d'une application sur une CGRA (flot présenté dans [91]). On y retrouve les deux entrées que sont l'application (programme C) et le modèle d'architecture reconfigurable ciblée. La première partie de ce flot, issue du système DURASE [72], est basée sur l'environnement de compilation GeCoS [1], elle permet principalement de transformer le programme C de l'application en un HCDG[4]. La sortie de l'étape de déploiement (composée de l'ordonnancement, l'allocation de ressources et le routage) est un HCDG annoté (AHCDG). La dernière étape est l'étape de génération des configurations et du code de contrôle pour la CGRA cible.

Organisation du chapitre

Ce chapitre commence par décrire le contexte des travaux réalisés, le projet ROMA, l'architecture reconfigurable à gros grain conçue dans le cadre de ROMA et la problématique de déploiement d'une application sur cette architecture. Ensuite, nous présentons deux modèles de contraintes : le premier pour le déploiement d'un graphe d'application

[4]Un HCDG (*Hierarchical Conditional Dependency Graph* ou Graphe Hiérarchisé aux Dépendances Conditionnées) est une représentation intermédiaire qui représente à la fois la partie flot de données et flot de contrôle d'un programme [8].

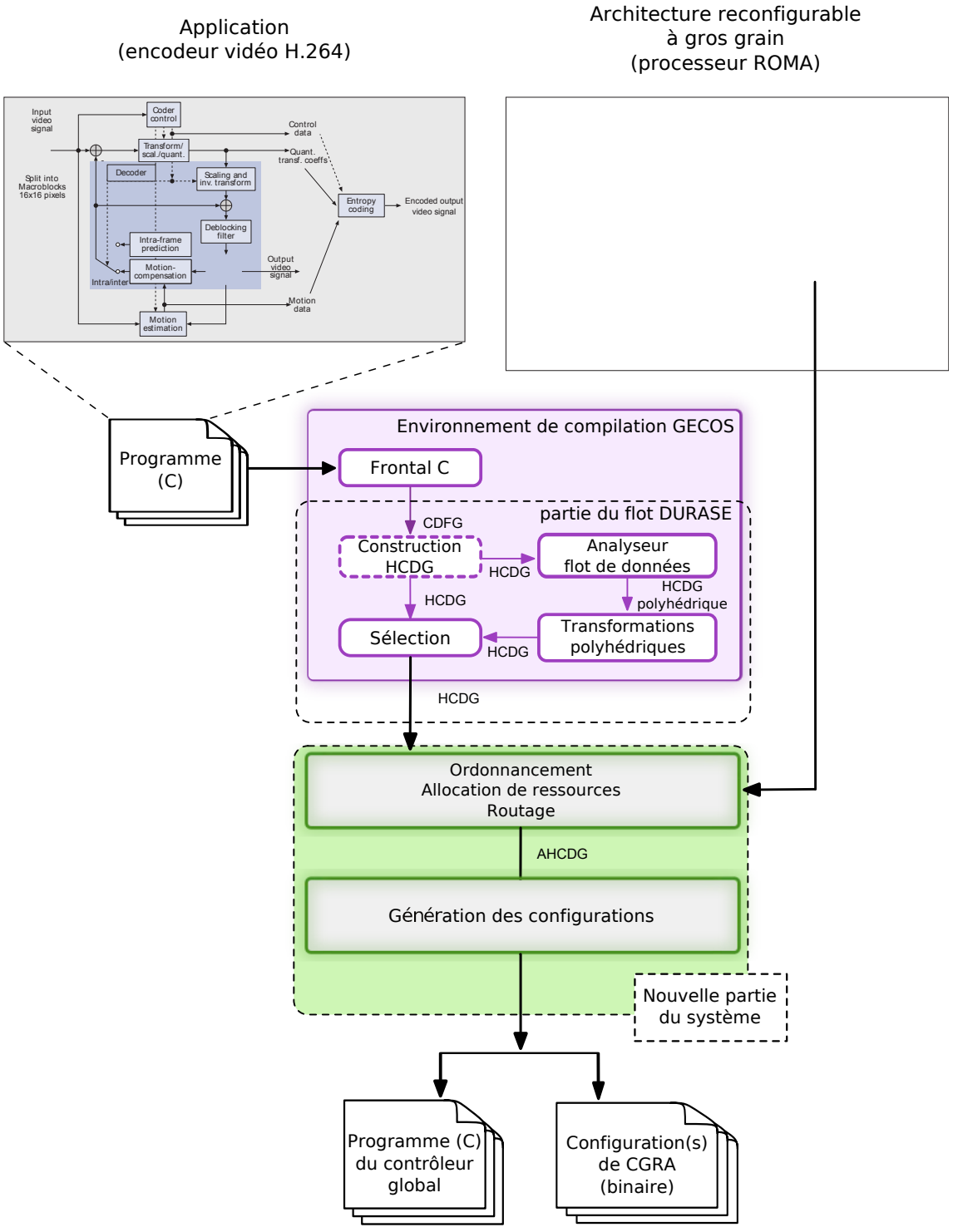


FIG. 4.1 – Flot générique de déploiement d'une application sur une CGRA.

sur un modèle d'architecture non pipeliné et le second pour un modèle d'architecture pipelinée. Les résultats obtenus sont discutés après la description de chaque modèle. Enfin, une brève présentation de l'étape de génération des configurations et de la validation du flot de déploiement clôtureront ce chapitre.

4.2 Contexte : Architecture ROMA

Les travaux présentés dans cette thèse ont été effectués dans un cadre industriel, à Technicolor R&D France (ex Thomson R&D France), en collaboration étroite avec l'équipe de recherche CAIRN de l'IRISA/INRIA Rennes. Cette collaboration a été élargie à travers le projet collaboratif ROMA (*Reconfigurable Operators for Multimedia Applications* - Processeur reconfigurable pour applications multimédia) [23] qui associe aussi les équipes de recherche du CEA LIST[5] et du LIRMM[6]. Ce projet a pour objectif d'accélérer les applications multimédia. Pour cela, les partenaires ont travaillé ensemble sur la définition et le développement d'une architecture reconfigurable dynamiquement à gros grain (CGRA) et d'une chaîne d'outils basée sur la CP. Le projet ROMA a débuté en 2007, il répond à l'appel à projets de l'Agence Nationale de la Recherche (ANR) « architectures du futur » et est de type « Recherche industrielle ».

4.2.1 Applications ciblées

L'objectif général du projet ROMA est d'accélérer les applications issues du domaine du multimédia. Certaines caractéristiques de ces applications sont inhérentes à ce domaine.

Tout d'abord, la règle des « 80/20 » qui définit que 80% du temps d'exécution d'une application est passé dans 20% du code, est particulièrement vraie dans le domaine du multimédia qui comporte des noyaux de calcul intensif (qui sont pour la plupart des algorithmes de traitement du signal). En compression vidéo par exemple, certains algorithmes comme l'estimation de mouvement ou à plus bas niveau la transformée en cosinus discrète (notée DCT pour *Discrete Cosine Transform*) ou encore la transformée d'Hadamard sont des noyaux de calcul bien connus (ceci est illustré dans l'introduction concernant l'estimation de mouvement). Ainsi, les parties critiques des applications multimédia sont aujourd'hui bien identifiées, elles sont l'objet de nombreuses études dont le but est de les accélérer en tenant compte de l'architecture sur laquelle elles sont implémentées.

De plus, les noyaux de calcul des applications multimédia comportent pour la plupart un parallélisme important qui facilite leur accélération sur des architectures parallèles.

Ainsi, des architectures spécialisées pour les noyaux de calcul des applications multimédia ont fait leur apparition. On trouve des architectures totalement dédiées (ASIC) ou des processeurs spécialisés (les processeurs DSP et les ASIP) ou encore des accélérateurs reconfigurables à grain fin (FPGA) ou à gros grain (CGRA). Le point commun de toutes ces cibles architecturales réside dans leur spécialisation, plus ou moins importante en fonction du niveau de flexibilité. En effet, elles sont parallèles, comportent des opérateurs spécialisés connectés par des réseaux de communication dont les topologies sont adaptées, etc.

[5]Commissariat à l'Energie Atomique et aux Energies Alternatives, Laboratoire d'Intégration des Systèmes et des Technologies.

[6]Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier.

Quant aux outils permettant de compiler des applications multimédia, ils ont eux aussi été dotés de fonctionnalités permettant d'identifier, d'extraire et de tirer partie des caractéristiques des noyaux de calcul de ces applications. Ceci est particulièrement vrai pour le parallélisme qui peut se situer à plusieurs niveaux. Par exemple, il est souvent possible de traiter plusieurs lignes d'une même image en parallèle ou plusieurs images en parallèle, etc. Un autre aspect important est l'identification de certains motifs de calcul récurrents comme l'utilisation de la valeur absolue (qui se traduit par un appel à une fonction dédiée dans la majorité des programmes) ou encore l'identification d'accumulations. Ces deux exemples sont représentatifs car ces motifs sont largement utilisés, à tel point que dans le projet ROMA ils ont fait l'objet d'une attention particulière.

La partie suivante décrit l'architecture du processeur ROMA spécialisé pour répondre à la problématique d'accélération des applications multimédia. Un des aspects privilégiés est l'obtention d'un compromis performance, flexibilité et efficacité des outils de compilation associés (triangle d'adéquation, présenté à la fin de l'introduction).

4.2.2 Architecture ROMA

Le processeur ROMA est une architecture reconfigurable dynamiquement à gros grain. Ce modèle d'architecture est spécialisable pour le ou les applications à accélérer par le biais de ses opérateurs (simple unité arithmétique et logique, opérateur complexe pipeliné, etc.) et de ses interconnexions dont la topologie peut être adaptée aux besoins. Une instance de ce modèle peut être considérée comme un ADSS[7] ou un ACSS[8] selon son degré de spécialisation. En effet, il est tout à fait possible de définir une instance dont les opérateurs et l'interconnexion sont spécialisés pour une classe d'application totalement connue, dans ce cas on parle d'ACSS. Mais il est aussi possible de définir une instance plus générique comportant des opérateurs génériques orientés multimédia et une interconnexion adaptée et flexible, dans ce cas on parle plutôt d'ADSS. Dans ce manuscrit comme dans le cadre du projet, la méthodologie de déploiement est vue sous l'angle d'un ADSS spécialisé pour le domaine du multimédia.

L'architecture ROMA a été définie et réalisée par le CEA LIST en collaboration avec les autres partenaires du projet. La description de l'architecture fournie ci-dessous est donc inspirée de la contribution de ce laboratoire pour la dissémination scientifique du projet à travers les publications aux conférences internationales ARC en 2009 [77] et DASIP en 2010 [91].

L'architecture proposée est représentée sur la figure 4.2. On y retrouve toutes les caractéristiques d'une CGRA : l'architecture consiste en un ensemble d'opérateurs gros grain reconfigurables (*Operators* sur la figure), de mémoires de données (*Data memory banks*), de mémoires de configurations, d'un réseau d'interconnexion pour les données (*Interconnection network*), d'un réseau de contrôle et d'un contrôleur principal (*Global CTRL*).

[7] *Application Domain-Specific System*, système spécifique à un domaine d'application (spectre applicatif large).
[8] *Application Class-Specific System*, système spécifique à une classe d'application (spectre applicatif réduit).

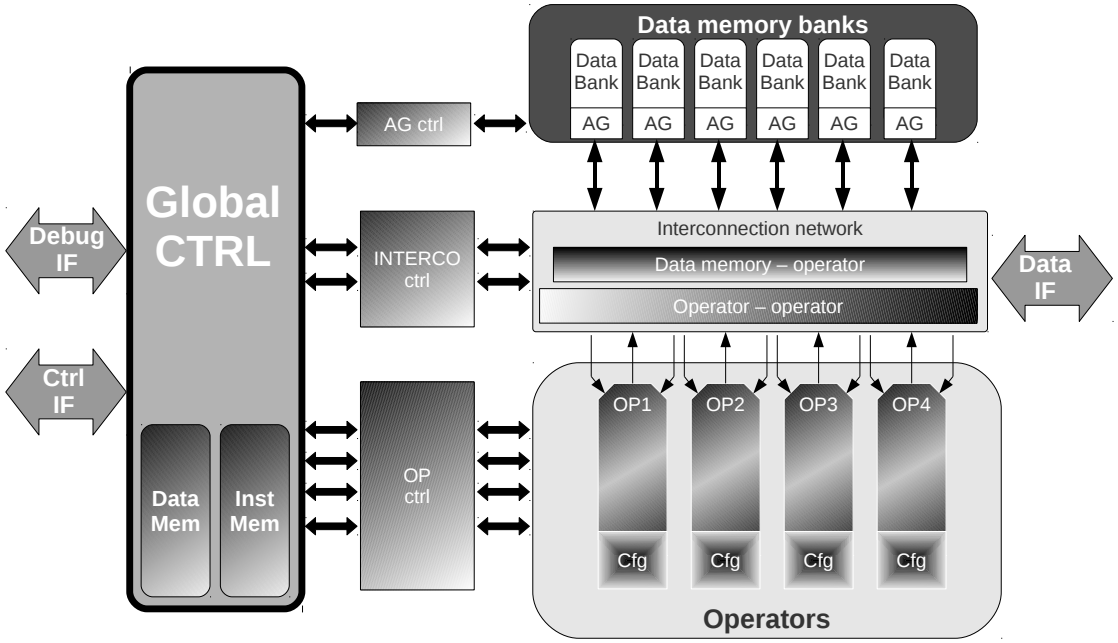


FIG. 4.2 – Architecture du processeur ROMA.

4.2.2.1 Chemin de données reconfigurable et son contrôle

Le chemin de données reconfigurable du processeur ROMA est constitué de bancs de mémoire, de deux réseaux d'interconnexion et d'un ensemble d'opérateurs reconfigurables. Il peut comporter de 4 à 12 opérateurs pouvant être homogènes ou hétérogènes ; ces paramètres sont définis selon la puissance de calcul nécessaire pour le spectre applicatif ciblé. Chaque opérateur dispose de sa propre mémoire de configuration et de sa propre interface avec le contrôleur permettant d'être indépendant en termes de contrôle et de reconfiguration.

Les opérateurs reconfigurables (« OP » sur la figure) sont connectés entre eux par un réseau d'interconnexion dédié appelé « réseau opérateur - opérateur » et avec la mémoire par un autre réseau appelé « réseau mémoire de donnée - opérateur ».

La reconfiguration du chemin de données est effectuée par le contrôle, à chaque cycle, des éléments le formant exploitant ainsi la flexibilité de l'architecture de manière efficace. Cela est géré par le système de contrôle, formé du contrôleur principal contenant une mémoire de données et une mémoire d'instructions et par les unités de contrôle associées à chaque élément du chemin de données : une pour le générateur d'adresses (« AG ctrl » sur la figure), une par réseau d'interconnexion (« INTERCO ctrl ») et une pour chaque opérateur (« OP ctrl »).

4.2.2.2 Opérateur reconfigurable à gros grain

Trois modèles d'opérateurs ont été définis dans le cadre de nos travaux. Le premier est un opérateur combinatoire générique capable d'exécuter toutes les opérations de base né-

cessaires à l'exécution d'une application. Le deuxième est un opérateur capable d'exécuter un ou plusieurs motifs de calcul complexes dédiés à l'accélération d'une ou de plusieurs applications. Le dernier opérateur, que nous présentons dans cette partie, est dédié à l'accélération des applications multimédia et a été conçu dans le cadre du projet ROMA.

L'architecture interne de l'opérateur reconfigurable gros grain ROMA est représentée sur la figure 4.3. Cet opérateur comporte un *pipeline* de 4 étages et est capable d'exécuter les opérations arithmétiques standards (addition, soustraction, multiplication). Il est aussi capable d'exécuter des opérations d'accumulation simples, de valeur absolue et des motifs de calcul complexes orientés multimédia (comme la somme des différences absolues notée SAD pour *Sum of Absolute Differences* ou le MAC). Chaque opérateur de l'architecture comporte deux entrées et une sortie d'une largeur de 40 bits.

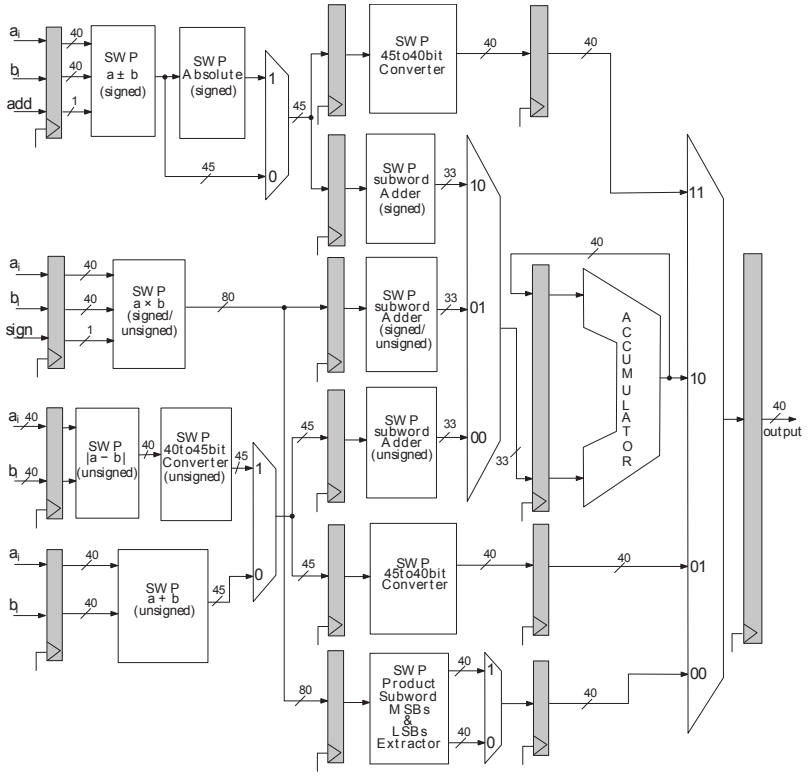


FIG. 4.3 – Architecture de l'opérateur reconfigurable gros grain SWP conçu dans le cadre du projet ROMA pour le domaine du multimédia.

L'opérateur est de type SWP (*Sub-Word Parallelism*), ce qui lui permet d'effectuer des opérations sur des données, typiquement des pixels, dont la largeur peut être de 8, 10, 12, 16 ou 40 bits (cette dernière est aussi utilisée pour représenter les données sur 32 bits). Le principe de base est simple, chaque opérateur interne est capable d'effectuer une opération sur des données de 40 bits ou deux opérations en parallèle sur des données de 16 bits ou encore cinq opérations sur des données de 8 bits, etc. De plus, la vitesse des différentes opérations arithmétiques est accrue par l'utilisation d'une représentation redondante des données appelée *borrow save*. Tous les détails sur cet opérateur (caractéristiques, performances etc.) sont disponibles dans l'article [63].

Enfin, l'opérateur ROMA a trois modes de fonctionnement appelés veille, configuration

et exécution. Ces trois modes sont totalement exclusifs. Durant le mode de veille, les interfaces de données et de configuration sont figées ce qui permet de minimiser l'activité de l'opérateur et donc de réduire la consommation énergétique dynamique du processeur. Le mode de configuration a une durée variant de 1 à 4 cycles selon la complexité de l'opération à affecter à l'opérateur.

4.2.2.3 Réseaux d'interconnexion

Les transferts de données dans le processeur ROMA sont effectués en fonction de leur type comme dans les architectures Harvard des processeurs DSP qui séparent le transfert des données et le transfert des signaux de contrôle. Dans le cas de ROMA on trouve en plus des transferts de configurations. Les réseaux d'interconnexion sont donc fractionnés en trois réseaux, un par type de transfert. Ces réseaux sont reconfigurables partiellement, ce qui permet de préparer le motif de communication suivant sans avoir à arrêter l'exécution en cours et cela en un cycle.

Le réseau d'interconnexion dédié aux transferts de données se décompose en deux sous-réseaux indépendants permettant de minimiser sa complexité matérielle et surtout d'éviter d'implémenter un réseau point-à-point coûteux comme un réseau *crossbar*. Un motif de communication est défini par deux configurations, une par sous-réseau.

Le réseau d'interconnexion opérateur - opérateur a une topologie dans laquelle chaque sortie d'opérateur est connectée aux $\frac{N}{2}$ entrées des opérateurs suivants. L'unique sortie d'un opérateur peut être connectée à plusieurs entrées des opérateurs suivants. Cette organisation permet d'obtenir un réseau particulièrement optimisé au niveau matériel et de sa latence.

Le réseau d'interconnexion mémoire de données - opérateur est flexible car il permet d'effectuer en parallèle des transferts données entre les bancs de mémoire et les opérateurs. Cette flexibilité permet en plus de simplifier les problèmes liés au placement des données dans les bancs de mémoire. Ce réseau peut être considéré comme un réseau totalement connecté (*full crossbar*). Dans le cas où toutes les applications sont connues (ACSS), il est possible de l'optimiser en utilisant un réseau multi-bus (le nombre maximum de transferts concurrents est dans ce cas connu) ou encore un réseau dédié avec des liens directs.

4.2.2.4 Espace mémoire

L'espace mémoire est divisé en deux segments indépendants ayant leur propre usage donc leur propre structure d'accès. On trouve un segment pour les données traitées par les opérateurs et un autre pour les configurations de tous les modules formant le chemin de données reconfigurable.

L'espace mémoire de données est réparti en plusieurs bancs de mémoire dont le nombre peut varier entre 3 et 3 fois le nombre d'opérateurs. La définition du nombre de bancs est un paramètre clé dans le dimensionnement de l'architecture. En effet si l'architecture ne comporte que 3 bancs de mémoire alors l'accès à ceux-ci peut devenir un goulot d'étranglement, en particulier si il n'est pas possible de chaîner les opérations. Par contre si elle comporte le maximum de bancs possibles alors chaque opérateur peut faire deux lectures et une écriture simultanément.

A chaque banc de mémoire est associé un générateur d'adresse programmable particulièrement adapté aux accès mémoire que l'on trouve dans les applications multimédia [47].

L'espace mémoire de configuration est réparti en plusieurs segments, un par type d'élément formant le chemin de données reconfigurable (opérateur, réseau et générateur d'adresse). Dans un esprit d'économie de mémoire, les duplications sont évitées permettant à un banc de reconfigurer tous les éléments du même type (diffusion de la même configuration à tous les générateurs d'adresse par exemple) ou seulement un élément de ce type.

La taille de ces bancs de mémoire est faible de par le grain de reconfiguration élevé. En effet, seulement 32 bits sont nécessaires pour décrire une configuration des réseaux et seulement de 32 à 128 bits pour décrire une opération en fonction de sa complexité. Ainsi il est possible de stocker plusieurs configurations dans ces mémoires ce qui implique une grande flexibilité.

4.2.2.5 Contrôleur global

Le contrôleur global (« Global CTRL » sur la figure) est un petit processeur généraliste de type RISC comportant un *pipeline* de 5 étages. Le *pipeline* et le jeu d'instructions ont été spécialisés pour des opérations de contrôle. Ce dernier comporte des instructions dont la largeur est mixte, 16 ou 32 bits, ce qui permet de minimiser l'empreinte du code embarqué. De plus, il est principalement constitué de deux opérations arithmétiques (addition et soustraction) dans différents formats, des opérations de manipulation de bit (décalage, insertion, extraction, etc.) et des opérations logiques classiques (et, ou, ou exclusif et non).

4.2.3 Adéquation application, architecture et CP

La clé du problème de déploiement d'un graphe d'application sur un modèle de CGRA en utilisant la CP réside dans l'adéquation entre l'application et l'architecture. En effet, il y a un lien direct entre cette adéquation et l'efficacité de recherche de solution par le solveur de contraintes. Prenons deux cas extrêmes pour l'illustrer.

Dans le cas simple, l'architecture est générique, elle est surdimensionnée (beaucoup d'opérateurs, de mémoire, un réseau totalement connecté, etc.) et l'application est très simple et régulière (calculs séquentiels sur peu de données). Dans ce cas, la résolution du problème est rapide et la solution est souvent prouvée optimale.

Dans le cas compliqué, l'architecture est très contraignante (peu d'opérateurs et tous spécialisés pour des types d'opérations différentes, peu de mémoire, un réseau d'interconnexion non orthogonal, etc.) et l'application est très complexe (nombreux calculs parallélisables, irrégularités, nombreuses variables temporaires, etc.). Dans ce cas, la résolution est longue voire impossible dans un temps acceptable lorsque l'on utilise la CP.

Heureusement, la spécialisation de l'architecture d'une CGRA est effectuée en adéquation avec le domaine ou la classe d'application ciblée ce qui permet d'éviter les cas compliqués où les caractéristiques de l'application ne sont pas ou peu compatibles avec celles de l'architecture. Malgré cela, certaines caractéristiques, de l'architecture ou de l'application, peuvent engendrer une résolution difficile du problème de déploiement.

Dans l'architecture ROMA, les opérateurs reconfigurables sont peu nombreux et hétérogènes (ils peuvent exécuter chacun différents types de motifs de calcul), les réseaux de

communication sont eux aussi reconfigurables. En ce qui concerne le réseau entre opérateurs, il est particulièrement intéressant de le limiter (d'un point de vue de la surface) en fonction des applications ciblées ce qui contraint encore plus la résolution.

4.2.4 Déploiement pour l'exploration de l'espace de conception

Le flot de déploiement proposé dans le projet ROMA permet aussi bien d'effectuer une exploration de l'espace de conception (DSE pour *Design Space Exploration*) que de générer le code binaire représentant les différentes configurations de l'architecture pour l'accélération d'une application. Le modèle d'architecture ainsi que le modèle de contraintes diffèrent selon la tâche à réaliser : dans le cas du DSE, le niveau d'abstraction est plus élevé, ce qui implique un modèle d'architecture simplifié et un modèle de contraintes moins contraignant que dans le cas de la génération de configurations où il est nécessaire de considérer les caractéristiques précises de l'architecture.

4.3 Modélisation de CGRA

Dans cette partie nous présentons deux modèles de CGRA, un premier peu détaillé pour l'exploration architecturale et le second plus détaillé permettant la génération de code.

Une CGRA est un modèle d'architecture simple qui peut être abstrait en utilisant les caractéristiques discutées dans la partie 1.2.2. Le modèle abstrait générique adapté à la modélisation du processeur ROMA est représenté sur la figure 4.4. On y retrouve un ensemble d'opérateurs et de mémoires locales, ainsi que deux réseaux d'interconnexion. Le réseau entre les mémoires et les opérateurs est renommé pour la suite du document en **« réseau mémoire »** et le réseau entre opérateur est renommé en **« réseau fonctionnel »**. Ce modèle est paramétrable de telle manière qu'il puisse servir à la fois pour l'exploration de l'espace de conception et pour la génération de code.

4.3.1 Modèle d'architecture pour l'exploration de l'espace de conception

Dans le cadre de l'exploration de l'espace de conception (DSE pour *Design Space Exploration*), le modèle d'architecture se doit d'être concis et générique (au sens paramétrable). En effet, si certaines caractéristiques sont imposées comme hypothèse alors l'espace de conception est réduit et certaines solutions potentiellement intéressantes sont écartées sans être véritablement évaluées. Le modèle se doit d'être paramétrable car après l'évaluation des premières itérations de déploiement des applications visées, il est intéressant de converger vers l'instance du modèle d'architecture répondant à toutes les contraintes de conception et technologiques en positionnant les différents paramètres.

Le modèle doit permettre de répondre aux questions suivantes :

- Quel est le nombre de mémoires et leur taille pour assurer que les données d'entrée/sortie et les données temporaires puissent être stockées dans les mémoires locales ?
- Quel est le nombre d'opérateurs nécessaires pour déployer une application de manière à obtenir l'exécution la plus rapide ?
- Quelle est la topologie idéale des réseaux d'interconnexion ?

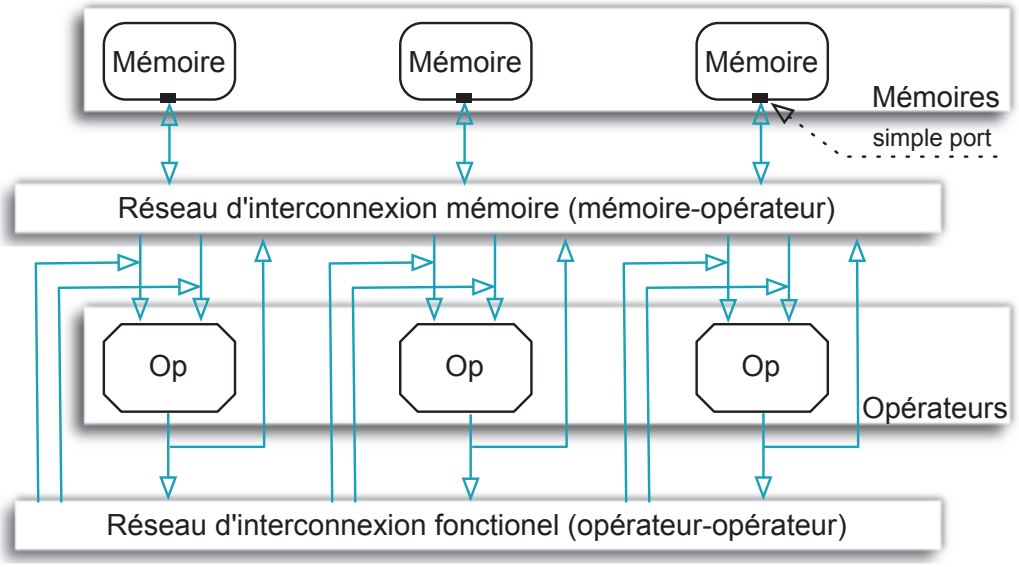


FIG. 4.4 – Modèle abstrait générique d'une CGRA.

Afin de pouvoir apporter une réponse à toutes ces questions, notre modèle pour le DSE est caractérisé par les paramètres suivants.

- L'ensemble des mémoires locales ainsi que leur taille. Les mémoires sont modélisées comme une unique mémoire locale multiport ayant un nombre de cellules mémoires représentant la somme des tailles des mémoires vues séparément. Cette modélisation se prête bien aux cas où le réseau mémoire est totalement connecté, sinon, le problème de placement des données dans les bancs de mémoire impose une modélisation plus précise.

- Le nombre d'opérateurs. On considère dans un premier temps que les opérateurs peuvent exécuter toutes les opérations nécessaires à l'exécution des applications. Ils peuvent être combinatoires ou pipelinés et comporte des registres en entrée.

- Le type et la topologie du ou des réseaux d'interconnexion.

- Les latences. Elles sont modélisées grossièrement, c.-à-d. par exemple, dans le cas où l'on dispose de deux réseaux et que l'un est plus rapide sa latence (représentant sa traversée) sera plus petite que celle de l'autre réseau. Les latences à prendre en compte sont principalement : les temps d'exécution des opérations, des accès mémoires, le temps de traversée des réseaux d'interconnexion et le temps de reconfiguration. La prise en compte simultanée de toutes ces latences n'est pas nécessaire en général, il est préférable de se concentrer sur certains aspects et de ne considérer que les latences impliquées.

De notre point de vue il y a deux manières ou étapes pour dimensionner une architec-

ture avec la CP.

Soit on ne limite pas (ou peu) les ressources, c.-à-d. qu'on considère que l'on dispose d'un nombre infini de mémoires (avec une taille infinie), d'opérateurs et un réseau totalement connecté. Dans ce cas, la résolution optimale du déploiement des applications visées permet de connaître le nombre d'accès concurrents à la mémoire, l'espace mémoire local maximum utilisé, le taux d'utilisation du ou des réseaux, le nombre d'opérateurs utilisés en parallèle. Cette approche donne une bonne idée des ressources nécessaires à l'exécution optimale d'une application sur une architecture ayant peu de contraintes quantitatives mais plutôt des contraintes structurelles. On peut voir cette approche comme un profilage quantitatif sous contraintes structurelles, c.-à-d. architecturales.

Soit on définit des contraintes quantitatives plus fortes (borne min et max des paramètres) représentant finalement des contraintes de conception au vu des limitations technologiques par exemple. Puis, de manière itérative, on place les applications sur des instances précises de l'architecture comme cela a été fait pour l'architecture ADRES par exemple [76]. La comparaison des déploiements optimaux sur ces instances permet au concepteur de trouver des compromis et de converger vers une instance idéale.

Le modèle de contraintes défini pour l'exploration de l'espace de conception est une simplification du modèle présenté par la suite. C'est pourquoi il n'est pas décrit dans ce manuscrit. Néanmoins, pour illustrer les possibilités offertes par cette approche, nous présentons dans le tableau 4.1 les résultats obtenus avec trois différents types de réseau lors du déploiement de l'IDCT (calcul sur les colonnes de la matrice) de l'application JPEG. Les réseaux utilisés sont un réseau tout connecté de type *crossbar* complet, le réseau ROMA (présenté dans le paragraphe 4.2.2.3) et l'absence de réseau (aucune liaison directe entre opérateurs).

Application	Graphe d'Application	Nœuds	Arcs	E/S	Type de réseau	Cycles	Optimal	Tps d'exécution (ms)
JPEG IDCT (col)	1	35	40	17	*crossbar* complet	14	√	1091
					ROMA	16	√	7693
					aucun	21	√	297
-//-	2	57	65	27	*crossbar* complet	23	√	117
					ROMA	26	√	15117
					aucun	34	√	266
Total	1+2	92	105	44	*crossbar* complet	37	√	1208
					ROMA	42	√	22810
					aucun	55	√	563

TAB. 4.1 – Résultats obtenus pour le déploiement de l'application JPEG IDCT avec le modèle de contraintes pour l'exploration de l'espace de conception - Trois types de réseau entre opérateurs sont évalués : un *crossbar* complet, le réseau ROMA et l'absence de réseau.

4.3.2 Modèle d'architecture pour la génération de configurations

Le modèle d'architecture a été raffiné pour permettre par la suite la génération des configurations pour le processeur ROMA. La plus grande différence avec le modèle pour le DSE réside dans l'identification des ressources et la prise en compte des latences propres à l'instance du modèle choisi après l'étape de dimensionnement de l'architecture. Dans ce

cas, les latences doivent être précises pour assurer une exécution correcte des applications sur l'architecture réelle.

Un des facteurs limitant est la complexité et le degré de précision du modèle de l'architecture. Cette limite est difficile à définir et pourrait faire l'objet de travaux de recherche complets. La complexité du modèle doit être maitrisée et donc le raffinement réalisé en fonction des besoins réels. Si certains aspects ne sont pas nécessaires à la génération de configurations fonctionnelles et optimisées, soit ils sont traités par un autre outil ou dans un second plan, soit ils sont gérés au niveau du *back-end* (comme la génération d'adresses par exemple).

Le modèle est défini alors comme suit :

- le nombre de mémoires locales ainsi que leur taille sont modélisés,
  - chaque mémoire est identifiée par un nombre unique,
  - chaque mémoire comporte un unique port,
  - les latences d'écriture et de lecture sont modélisées (constantes),

- le nombre d'opérateurs,
  - chaque opérateur est identifié par un nombre unique,
  - chaque opérateur supporte un ensemble d'opérations identifiées par un nombre unique,
  - chaque opérateur dispose de deux ports d'entrée et d'un port de sortie pour le transfert des données,

- le type et la topologie du ou des réseaux d'interconnexion sont modélisés,
- les latences de traversée des réseaux sont modélisées précisément (mais si possible constantes).

4.4 Déploiement d'une application avec optimisation du temps d'exécution

Le déploiement d'une application sur CGRA en minimisant le temps d'exécution est une approche où l'objectif est la performance. Cet objectif est ambitieux car il implique d'optimiser plusieurs aspects : l'utilisation des unités fonctionnelles mais aussi une utilisation maximale du réseau de communication fonctionnel plus rapide que le réseau de communication passant par la mémoire. Le modèle doit prendre en compte ces difficultés pour permettre une résolution optimale du problème d'ordonnancement.

4.4.1 Modèle de contraintes

Ce paragraphe décrit le modèle de contraintes proposé pour déployer un graphe d'application en minimisant le temps d'exécution sous contraintes de ressources architecturales dans le cadre du projet ROMA. Ce modèle a été défini pour l'architecture représentée sur la figure 4.5 qui est une instance du modèle de CGRA ROMA présenté précédemment.

Les contraintes définies dans le modèle permettent de modéliser les caractéristiques de l'architecture lors de l'exécution de l'application, c.-à-d. l'utilisation des différents réseaux

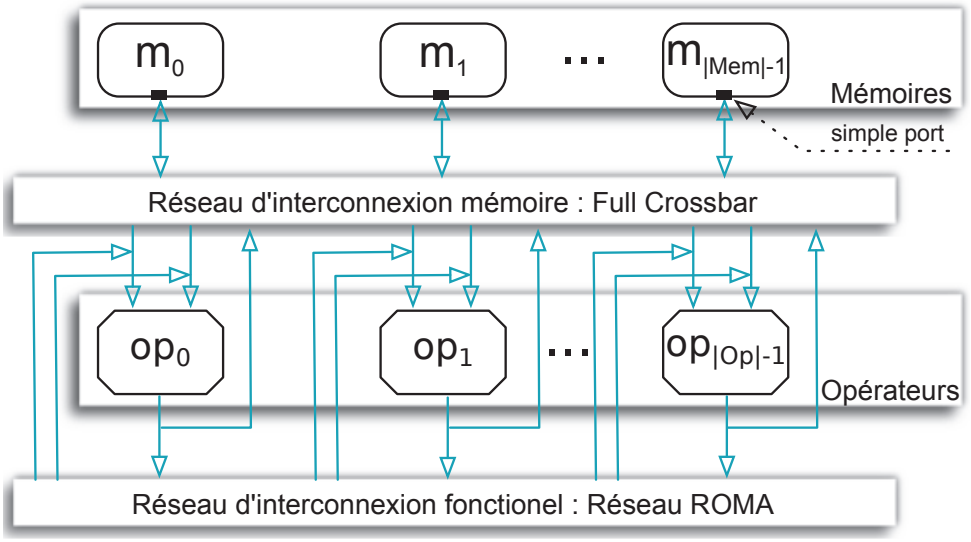


FIG. 4.5 – Modèle de la CGRA ROMA.

de communication (le choix et le respect de la topologie), le partage des ressources (opérateurs et mémoires), mais aussi les contraintes temporelles assurant l'ordre d'exécution des opérations et le respect des différents délais (délai d'exécution d'une opération, des accès mémoires, etc.). Enfin, une fonction de coût représentant la minimisation du temps d'exécution de l'application est définie.

4.4.1.1 Graphe d'application

Le graphe d'application, noté AG, est modélisé par un graphe acyclique orienté représentant le flot de données (DFG pour *Data Flow Graph*) à placer sur l'architecture cible. Sa définition donnée ci-après diffère peu de celle définie en 3.1 pour le problème de fusion de chemin de données.

Définition 4.1 *Un graphe d'application est un graphe de flot de données* $AG = (V, E)$, V *désigne l'ensemble des nœuds et* E *l'ensemble des arcs.*

- *Un nœud* $v \in V$ *représente soit*
  - *un motif de calcul, soit*
  - *une variable d'entrée/sortie.*
- *Un arc* $e_{ij} = (v_i, v_j) \in E$ *représente un transfert de données entre un nœud* v_i *et un nœud* v_j.

Concernant les nœuds d'un AG, nous définissons deux ensembles, notés OP et ES, pour différencier les nœuds représentant un motif de calcul et ceux représentant une variable d'entrée/sortie.

La figure 4.6 représente un graphe d'application dont les motifs de calcul sont des opérations élémentaires. Les variables d'entrée sont préfixées par un point d'interrogation et les variables de sortie par un point d'exclamation.

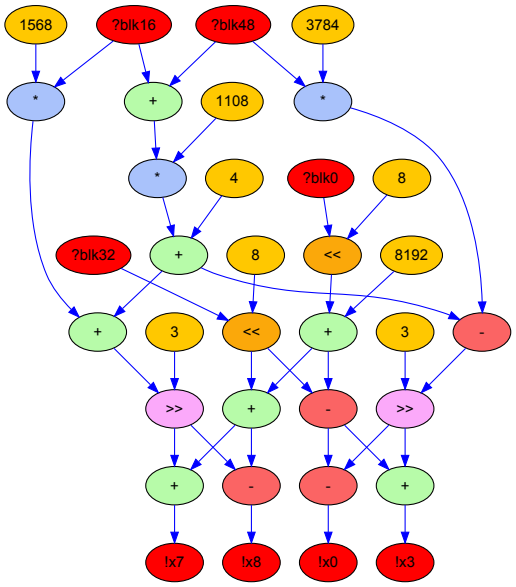


FIG. 4.6 – Exemple de graphe d'application.

Pour modéliser le problème de déploiement d'un AG sur le modèle d'architecture ROMA par la CP il est nécessaire de définir, pour chaque nœud et chaque arc de l'AG, un ensemble de variables, chacune étant initialisée avec un domaine particulier.

4.4.1.2 Modélisation d'un nœud

Un nœud $v \in V$ dans l'AG représente une variable d'entrée/sortie ou un motif de calcul. Pour rappel, le motif de calcul peut être simple, car composé d'une seule opération de base (addition, multiplication, etc.), ou complexe, car composé de plusieurs opérations de base.

Lorsqu'un nœud représente une variable d'entrée/sortie, une opération d'écriture ou de lecture en mémoire est modélisée.

Dans le cas où un nœud représente un motif, nous modélisons les opérations de transfert de la donnée en sortie ainsi que l'exécution du motif de calcul. Pour rappel, les opérateurs exécutant ces motifs ne comportent qu'un port de sortie, en conséquence une seule donnée est produite et transférée.

Le tableau 4.2 donne les principales variables associées à un nœud, leur signification ainsi que leur domaine d'initialisation. Ces variables et leur utilisation par des contraintes sont détaillées par la suite. Si le domaine d'une variable est initialisé en fonction des paramètres de l'architecture, il est noté par une étoile $*$.

4.4.1.3 Modélisation d'un arc

Un arc $e_{ij} = (v_i, v_j) \in E$ dans le graphe d'application représente un transfert de données entre deux nœuds v_i et $v_j \in V$.

Le transfert d'une donnée d'un opérateur à une autre dans le modèle d'architecture proposé dans le cadre du projet ROMA peut être réalisé de deux façons différentes : soit à

v_{start}	Date de début d'exécution du motif représenté par v	$\{0..\infty\}$
v_{delay}	Temps d'exécution du motif représenté par v	$*$
v_{end}	Date de fin de l'exécution du motif représenté par v	$\{0..\infty\}$
v_{op}	Identifiant de l'opérateur utilisé pour exécuter le motif représenté par $v \in OP$ ($\forall v \in ES, v_{op} = \emptyset$)	$\{i \mid op_i \in \{0..\lvert operateur\rvert - 1\} \wedge op_i$ peut exécuter $v\}$
$v_{op_{activity}}$	Temps d'occupation de l'opérateur utilisé pour exécuter le motif représenté par $v \in OP$ (ce temps inclut le temps d'exécution et le temps nécessaire pour le transfert de la donnée produite par v)	$\{0..\infty\}$
v_{mem}	Identifiant de la mémoire pouvant être utilisée pour le transfert de la donnée produite par v	$\{0..\lvert memoire\rvert - 1\}$
v_{mem_acces}	Représente l'utilisation du réseau mémoire pour au moins un transfert de la donnée produite par v	$\{0, 1\}$
$v_{start_{WR}}$	Date de début de l'écriture en mémoire de la donnée produite par v	$\{0..\infty\}$
v_{life_time}	Durée de vie de la variable en mémoire produite par v	$\{0..\infty\}$
v_{net_acces}	Représente l'utilisation du réseau fonctionnel pour au moins un transfert de la donnée produite par v	$\{0, 1\}$

TAB. 4.2 – Variables associées à un nœud, leur signification ainsi que leur domaine d'initialisation (Si le domaine d'une variable est initialisé en fonction des paramètres de l'architecture il est noté par une étoile $*$).

travers la mémoire ou soit à travers un réseau de communication entre opérateurs. Deux réseaux de communication distincts sont donc modélisés : un réseau mémoire de données - opérateur et un réseau opérateur - opérateur. Nous les nommerons dorénavant « réseau mémoire » et « réseau fonctionnel ».

Les principales variables associées à un arc e_{ij}, leur signification ainsi que leur domaine d'initialisation sont données dans le tableau 4.3.

e_{mem_ope}	Représente l'utilisation du réseau mémoire (valeur 1) pour le transfert de la donnée représenté par e	$\{0, 1\}$
e_{ope_ope}	Représente l'utilisation du réseau fonctionnel (valeur 1) pour le transfert de la donnée représenté par e	$\{0, 1\}$
$e_{ij_{start_{RD}}}$	Date de début de la lecture en mémoire de la donnée produite par v_i et utilisée par v_j	$\{0..\infty\}$
e_{life_time}	Durée de vie de la variable en mémoire lors du transfert de la donnée représenté par e via la mémoire	$\{0, 1\}$

TAB. 4.3 – Variables associées à un arc, leur signification ainsi que leur domaine d'initialisation.

4.4.1.4 Contraintes de communication

Choix du réseau de communication Un arc dans l'AG représente un transfert de données sur l'un des réseaux de communication, soit sur le réseau mémoire, soit sur le réseau fonctionnel. Ainsi, pour tout arc $e \in E$, la contrainte 4.4.1 impose une exclusion mutuelle sur les variables booléennes e_{mem_ope} et e_{ope_ope} représentant respectivement l'utilisation du réseau mémoire et l'utilisation du réseau fonctionnel lors d'un transfert de données. Cette contrainte est illustrée par la figure 4.7.

Contrainte 4.4.1

$$\forall e \in E,\ e_{mem\_ope} + e_{ope\_ope} = 1$$

Dans le cas particulier où le nœud source (ou le nœud destination) représente une variable d'entrée (sortie) alors l'arc issu de (ou vers) ce nœud représente forcément un transfert sur le réseau mémoire. Dans ce cas, la contrainte 4.4.2 est appliquée.

Contrainte 4.4.2

$$\forall e_{ij} = (v_i, v_j) \in E \mid (v_i \in ES \wedge v_j \in OP) \vee (v_i \in OP \wedge v_j \in ES) : e_{ij_{mem\_ope}} = 1$$

Modélisation de la topologie d'un réseau Pour modéliser toute les topologies possibles d'un réseau fonctionnel point-à-point, nous définissons une matrice de communication, $ComMat$, qui spécifie toutes les connexions entre les éléments du réseau deux à deux. Cette matrice définit une relation entre les ressources assignées aux nœuds $v_{i_{op}}$ et $v_{j_{op}}$, elle contient toutes les connexions point-à-point possibles entre ces deux ressources. La contrainte 4.4.3 est utilisée pour contraindre un transfert par la topologie du réseau emprunté. En pratique cette contrainte est implémentée avec la contrainte globale « *ExtensionalSupport* » du solveur JaCoP.

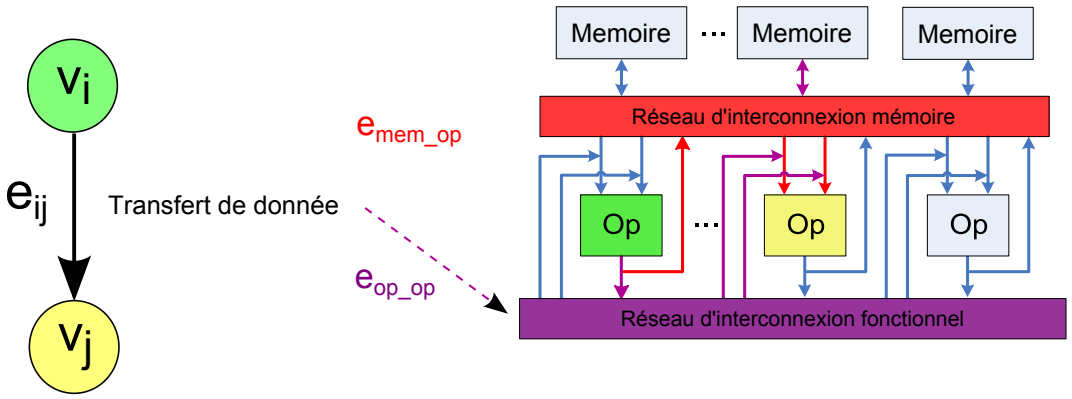


FIG. 4.7 – Modélisation du choix du réseau.

Contrainte 4.4.3

$$\forall e_{ij} = (v_i, v_j) \in E | v_i \in OP \wedge v_j \in OP :$$
$$If\ e_{ij_{ope\_ope}} = 1\ then\ ComMat[v_{i_{op}}][v_{j_{op}}] = 1$$

Dans le modèle proposé pour la CGRA ROMA, le réseau d'interconnexion fonctionnel est modélisé de manière plus précise et plus simple. Cela permet, comparé à une approche générique, de réduire l'espace des solutions et donc d'espérer obtenir une résolution plus rapide du problème de déploiement. Pour rappel, ce réseau est capable de réaliser tous les motifs de communication acyclique entre les N opérateurs du processeur. Chaque sortie est connectée aux $\frac{N}{2}$ entrées des opérateurs suivants. Le schéma de communication supporté par ROMA est illustré par la figure 4.8. Dans l'exemple proposé, l'architecture est composée de quatre opérateurs, l'opérateur 0 est connecté aux opérateurs 1 et 2, l'opérateur 1 aux opérateurs 2 et 3 et l'opérateur 2 est connecté au 3.

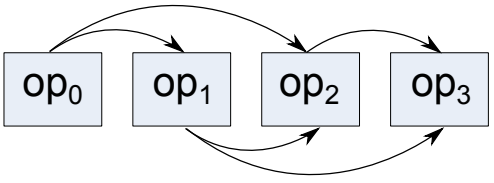


FIG. 4.8 – Réseau ROMA à quatre opérateurs.

Cette topologie particulière est modélisée par la contrainte 4.4.4 impliquant un nœud source et un nœud destination représentant tous deux un motif (et non une variable d'entrée/sortie). La variable $e_{ij_{op_r}} \in \{1..\frac{|operators|}{2}\}$ modélise les identifiants des opérateurs atteignables à partir de $v_{i_{op}}$. Le domaine de cette variable peut être initialisé à $\{0..\frac{|operators|}{2}\}$ si un rebouclage sur l'opérateur est possible.

Contrainte 4.4.4

$$\forall e_{ij} = (v_i, v_j) \in E | v_i \in OP \wedge v_j \in OP :$$
$$If\ e_{ij_{ope\_ope}} = 1\ then\ v_{j_{op}} = v_{i_{op}} + e_{ij_{op_r}}$$

Quant au réseau d'interconnexion mémoire de l'architecture ROMA il est assimilé à un réseau totalement connecté et implémenté comme un *crossbar* complet. Dans le modèle CSP, aucune contrainte n'est imposée sur ce réseau car il ne comporte aucune limitation au niveau de sa topologie.

4.4.1.5 Contraintes temporelles

La structure de l'AG impose un ordre partiel d'exécution des nœuds. Pour assurer cette propriété, un ensemble de contraintes temporelles sont définies. D'une part, nous avons modélisé le temps d'exécution d'un motif en fonction de l'opérateur utilisé et, d'autre part, le temps de transfert d'une donnée en fonction du délai de traversée du réseau emprunté. De plus, l'ordre des étapes d'écriture et de lecture en mémoire dans le cas d'un transfert via le réseau mémoire est aussi considéré.

Modélisation du temps d'exécution d'un motif représenté par un nœud La contrainte 4.4.5 garantit que pour tous les nœuds $v \in V$, la date de fin d'exécution du motif associé respecte bien le délai nécessaire à son exécution.

Contrainte 4.4.5

$$\forall v \in V : v_{end} = v_{start} + v_{delay}$$

Le temps d'exécution du motif représenté par le nœud $v \in V$, noté v_{delay}, est déterminé en fonction de la ressource sur laquelle le nœud est placé. Dans le cas où ce nœud représente une variable, son délai est nul ; lorsqu'il représente un motif de calcul alors il est égal au temps nécessaire à la ressource sur laquelle ce nœud est placé pour exécuter ce motif. Cette notion est exprimée par la contrainte 4.4.6 (« *ExtensionalSupport* » en pratique) où $v_{pattern}$ est l'identifiant du motif de calcul représenté par v et $DelayMat$ est une matrice modélisant le temps d'exécution de chaque motif de calcul en fonction de la ressource sur laquelle il est exécuté.

Contrainte 4.4.6

$$\forall v \in OP : v_{delay} = DelayMat[v_{op}][v_{pattern}]$$
$$\forall v \in ES : v_{delay} = 0$$

La figure 4.9 illustre la différence de temps d'exécution selon l'opérateur sur lequel le motif est placé. Sur cet exemple, l'opérateur $v_{op}{}'$ à un temps d'exécution plus grand que v_{op} pour exécuter le motif représenté par v.

Remarque : Il est important de noter que cette dernière contrainte complexifie le problème de déploiement d'un AG. Cela impacte le temps de résolution, en particulier dans le contexte d'une architecture comportant de nombreux opérateurs ayant une grande hétérogénéité. Notre approche est, dans ce contexte, plus adaptée à la modélisation d'architectures homogènes. La variable v_{delay} est dans ce cas définie comme une constante indépendamment de l'opérateur sur lequel est placé v.

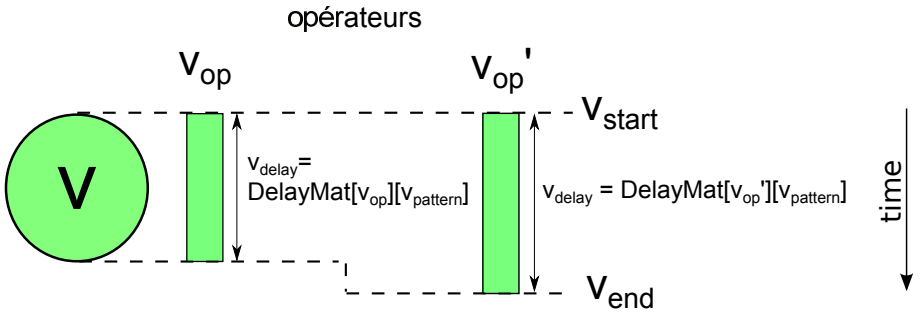


FIG. 4.9 – Temps d'exécution selon l'opérateur sur lequel le motif est placé.

4.4.1.6 Contraintes temporelles liées au choix du réseau

Cas du réseau mémoire

A l'occasion d'un transfert de données entre deux nœuds v_i et $v_j \in V$, représenté par l'arc $e_{ij} = (v_i, v_j) \in E$, les opérations de lecture et d'écriture se succèdent dans le temps. Leurs durées dépendent de la latence de traversée du réseau et des opérations mémoires proprement dites. L'ensemble de contraintes 4.4.7 modélise ces notions d'ordre en fonction des latences concernées. La figure 4.10 illustre ces contraintes.

Nous avons défini de nouvelles variables représentant les durées d'écriture et de lecture pour chaque arc, notées respectivement $\Delta e_{ij_{WR}}$ et $\Delta e_{ij_{RD}}$. Ces variables doivent être nulles si le réseau mémoire n'est pas utilisé comme le montrent les contraintes (c) et (e). La contrainte (a) assure que l'écriture d'une donnée produite par le nœud source v_i n'est effectuée qu'après la fin de son exécution ($v_{i_{end}}$). De la même manière la contrainte (b) assure que la lecture de la donnée n'est effectuée qu'à l'issue de l'écriture dont la latence est représentée par WR_{lat}. Enfin la contrainte (d) impose que l'exécution du motif représenté par le nœud destination v_j ne démarre qu'après que la donnnée dont il a besoin soit bien disponible, la latence de la lecture étant notée RD_{lat}.

On peut noter que la contrainte (a) impose une inégalité entre $v_{i_{start_{WR}}}$ et $v_{i_{end}}$ alors que la contrainte (d) impose un égalité entre la fin d'une lecture en mémoire ($e_{ij_{start_{RD}}} + \Delta e_{ij_{RD}}$) et la date de début d'exécution du motif représenté par v_j. Cette différence est issue des caractéristiques des opérateurs instanciés dans l'architecture ROMA. En effet, les opérateurs ont des registres en entrée qui leur permettent de conserver la donnée produite jusqu'à la date de début de l'écriture en mémoire (et cela, tant que les données en entrée ne sont pas modifiées). De plus, la synchronisation des données en entrée est imposée par le modèle d'architecture.

Contrainte 4.4.7 $\forall e_{ij} = (v_i, v_j) \in E$:

$$v_{i_{start_{WR}}} \geq v_{i_{end}} \quad \text{(a)}$$

$$e_{ij_{start_{RD}}} \geq v_{i_{start_{WR}}} + \Delta e_{ij_{WR}} \quad \text{(b)}$$

$$\Delta e_{ij_{WR}} = WR_{lat} * e_{ij_{mem\_ope}} \quad \text{(c)}$$

$$e_{ij_{start_{RD}}} + \Delta e_{ij_{RD}} = v_{j_{start}} \quad \text{(d)}$$

$$\Delta e_{ij_{RD}} = RD_{lat} * e_{ij_{mem\_ope}} \quad \text{(e)}$$

Si le nœud source est un nœud d'entrée ou si le nœud destination est un nœud de sortie, les contraintes à imposer sont différentes. Dans le cas d'un nœud d'entrée seules

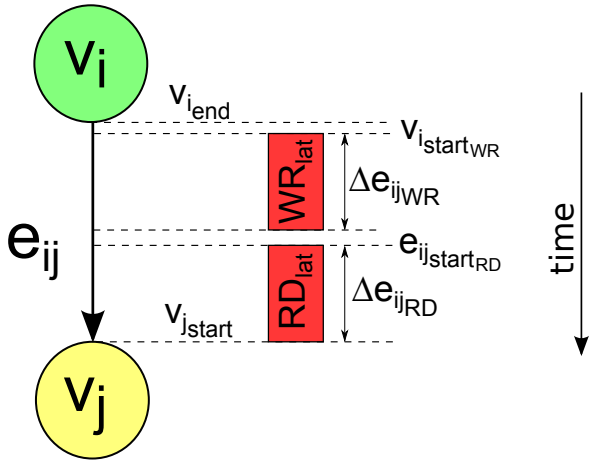


FIG. 4.10 – Opérations d'écriture et de lecture en mémoire lors d'un transfert de données via la mémoire $v_{i_{mem}}$.

des opérations de lecture ont lieu (4.4.8). Dans le cas d'un tranfert vers un nœud de sortie seule une opération d'écriture est modélisée (4.4.9).

Contrainte 4.4.8 $\forall e_{ij} = (v_i, v_j) \in E$ *tel que* $v_i \in ES$ *(entrée) :*

$$e_{ij_{start_{RD}}} = v_{i_{end}} \quad \text{(a)}$$

$$e_{ij_{start_{RD}}} + \Delta e_{ij_{RD}} = v_{j_{start}} \quad \text{(b)}$$

$$\Delta e_{ij_{RD}} = RD_{lat} \quad \text{(c)}$$

Contrainte 4.4.9 $\forall e_{ij} = (v_i, v_j) \in E$ *tel que* $v_j \in ES$ *(sortie) :*

$$v_{i_{start_{WR}}} \geq v_{i_{end}} \quad \text{(a)}$$

$$v_{j_{start}} = v_{i_{start_{WR}}} + \Delta e_{ij_{WR}} \quad \text{(b)}$$

$$\Delta e_{ij_{WR}} = WR_{lat} \quad \text{(c)}$$

Cas du réseau fonctionnel

Pour modéliser le temps d'occupation du réseau fonctionnel lors d'un transfert de données, nous définissons pour chaque arc entre deux nœuds représentant l'exécution d'un motif de calcul une nouvelle variable $\Delta e_{ij_{ope\_ope}}$. Cette variable est non nulle lors d'un transfert via le réseau fonctionnel. Les contraintes sur cette variable sont données en 4.4.10 où la variable ope_ope_{lat} représente la latence de traversée du réseau fonctionnel. La figure 4.11 montre que le temps de traversée du réseau fonctionnel a une valeur comprise sur l'intervalle fin du nœud source $v_{i_{end}}$ et début du nœud destination $v_{j_{start}}$.

Contrainte 4.4.10 $\forall e_{ij} = (v_i, v_j) \in E \mid v_i \in OP \wedge v_j \in OP$ *:*

$$\Delta e_{ij_{ope\_ope}} = v_{j_{start}} - v_{i_{end}}$$

$$\Delta e_{ij_{ope\_ope}} \geq ope\_ope_{lat} * e_{ij_{ope\_ope}}$$

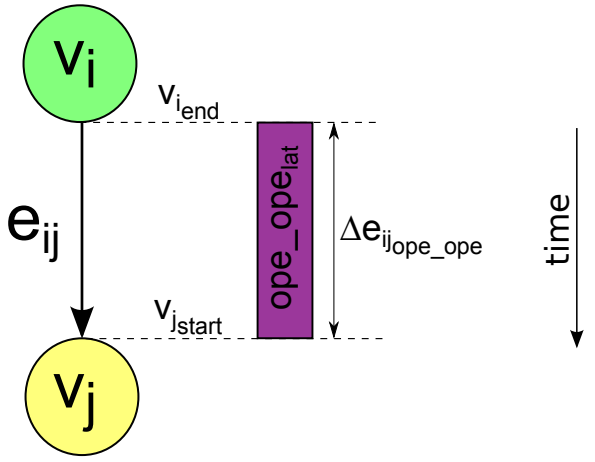


FIG. 4.11 – Transfert de données via le réseau fonctionnel.

4.4.1.7 Contraintes de partage de ressources

La modélisation du problème de partage de ressources est effectuée à l'aide de la contrainte globale différentielle « *Diff2* » présentée précédemment (2.1.2.3). Cette contrainte assure pour un ensemble d'opérations, chacune représentée dans un espace bidimensionnel temps/ressource par un rectangle noté $Rec = [x, y, \Delta x, \Delta y]$, qu'il n'y a aucun chevauchement entre elles. La variable x représente le début de l'opération, y l'identifiant de la ressource associée et Δx son temps d'occupation. La variable Δy est en général une variable booléenne qui modélise l'utilisation de la ressource ($\Delta y = 1$ si la ressource est utilisée, 0 sinon).

Ainsi, il est facile de modéliser le problème de partage de ressources. Par exemple, le fait que deux nœuds de l'AG, représentant tous deux un motif de calcul, ne puissent pas être placés sur le même opérateur au même moment, ou que deux accès à une même mémoire simple port, ne se fassent pas en même temps.

Modélisation des accès aux mémoires Chaque accès potentiel à la mémoire est modélisé par un rectangle. Etant donné qu'un seul port est présent sur la sortie des opérateurs, il ne peut y avoir qu'un seul accès en écriture par nœud $v_i \in V$ pour stocker la donnée produite par l'opérateur v_{op_i}. C'est pourquoi, seul un rectangle noté $Rec(v_{i_{WR}})$ est défini en 4.4.11(a) pour modéliser cette écriture. Rappelons que la variable $v_{i_{mem\_acces}}$, définie en 4.4.11(b), modélise l'utilisation de la mémoire pour au moins un transfert. Par contre, on définit en 4.4.12 pour chaque arc e_{ij} sortant du nœud v_i, un rectangle $Rec(e_{ij_{RD}})$ représentant un possible accès en lecture de la donnée nécessaire à l'opération représentée par v_j.

Contrainte 4.4.11 $\forall v_i \in OP$:

$$Rec(v_{i_{WR}}) = [v_{i_{start_{WR}}}, v_{i_{mem}}, WR_{lat}, v_{i_{mem\_access}}] \quad \text{(a)}$$

$$v_{i_{mem\_acces}} \in \{0,1\} \Leftrightarrow \sum_{\forall e_{ij} \in v_{i_{outputs}}} e_{ij_{mem\_ope}} > 0 \quad \text{(b)}$$

Contrainte 4.4.12 $\forall e_{ij} \in v_{i_{outputs}}$:

$$Rec(e_{ij_{RD}}) = [e_{ij_{start_{RD}}}, v_{i_{mem}}, RD_{lat}, e_{ij_{mem\_ope}}]$$

Dans notre modèle, nous avons pris en compte le cas particulier où les nœuds source et destination représentent une opération d'entrée/sortie pour laquelle il n'y a respectivement aucune écriture ou aucune lecture.

La contrainte 4.4.13 assure que tout accès à une mémoire est exclusif dans le temps grâce à la contrainte différentielle « *Diff2* ». Cette contrainte est en fait trop restrictive. En effet, certaines exceptions doivent être prises en compte lorsque plusieurs lectures de la même donnée sont effectuées simultanément. Cela correspond en pratique au cas où une accès en lecture est effectué sur une mémoire et la donnée lue est utilisée par plusieurs opérateurs à des moments différents (mais avec un recouvrement entre les deux opérations de lecture modélisées). Alors la donnée lue doit être présente en sortie de la mémoire jusqu'à la fin de la dernière lecture.

Pour gérer ces cas, il est possible de spécifier pour la contrainte 4.4.13 les paires de rectangles pouvant se chevaucher, ce qui est fait pour toutes les lectures potentielles d'une même donnée (représentées par tous les arcs sortant d'un même nœud) dans 4.4.14.

Contrainte 4.4.13 $\forall v_i \in V,\ e_{ij} \in v_{i_{outputs}}$ *:*

$$\texttt{Diff2}([...Rec(v_{i_{WR}}), Rec(e_{ij_{RD}})...])$$

Contrainte 4.4.14 $\forall v_i \in E \wedge e_{ij}, e_{ik} \in v_{i_{outputs}} \wedge j \neq k$ *:*

$$[Rec_j, Rec_k] = [Rec(e_{ij_{RD}}), Rec(e_{ik_{RD}})]$$

La figure 4.12 illustre les accès à la mémoire $v_{i_{mem}}$ pour le transfert de la donnée produite par le nœud source v_i vers les deux nœuds destinations v_j et v_k. Ces accès à la mémoire sont représentés par les trois rectangles rouges. Dans cet exemple, l'exception permet d'accepter le chevauchement des deux lectures de la donnée.

Modélisation de l'occupation des cellules des mémoires La modélisation du temps d'occupation d'une cellule mémoire en vue du stockage de la donnée en sortie d'un nœud v_i est effectuée par la variable v_{life_time} définie pour chaque nœud de l'AG. Cette variable correspond au temps passé par une donnée en mémoire, c.-à-d. entre son écriture et sa dernière lecture. Pour cela il est nécessaire de définir, pour chaque arc sortant de v_i, une variable $e_{ij_{life\_time}}$ correspondant à la durée de vie de la variable en mémoire (contrainte 4.4.15(a)). La valeur maximale de ces variables, définie par la contrainte 4.4.15(b), représente la durée de vie d'une donnée en mémoire nécessaire pour assurer tous ses transferts.

Contrainte 4.4.15 $\forall v_i \in V,\ e_{ij} \in v_{i_{outputs}}$ *:*

$$e_{ij_{life\_time}} = v_{j_{start}} - v_{i_{start_{WR}}} \quad \text{(a)}$$

$$v_{i_{life\_time}} = max(..., e_{ij_{life\_time}} * e_{ij_{mem\_ope}}, ...) \quad \text{(b)}$$

Dans le cas où v_i représente une variable d'entrée, il n'y a pas d'écriture dans la mémoire mais seulement des lectures, car on considère que la donnée est présente dans la mémoire au début de l'exécution de l'AG. Le temps passé en mémoire par une variable d'entrée correspond donc à la date de sa dernière lecture, soit $v_{i_{life\_time}} = max(..., v_{j_{start}} * e_{ij_{mem\_ope}}, ...)$.

Dans le cas où v_i représente un opérateur connecté à une variable de sortie v_j, la donnée produite reste en mémoire jusqu'à la fin de l'exécution de l'AG. Donc $e_{ij_{life\_time}} =$

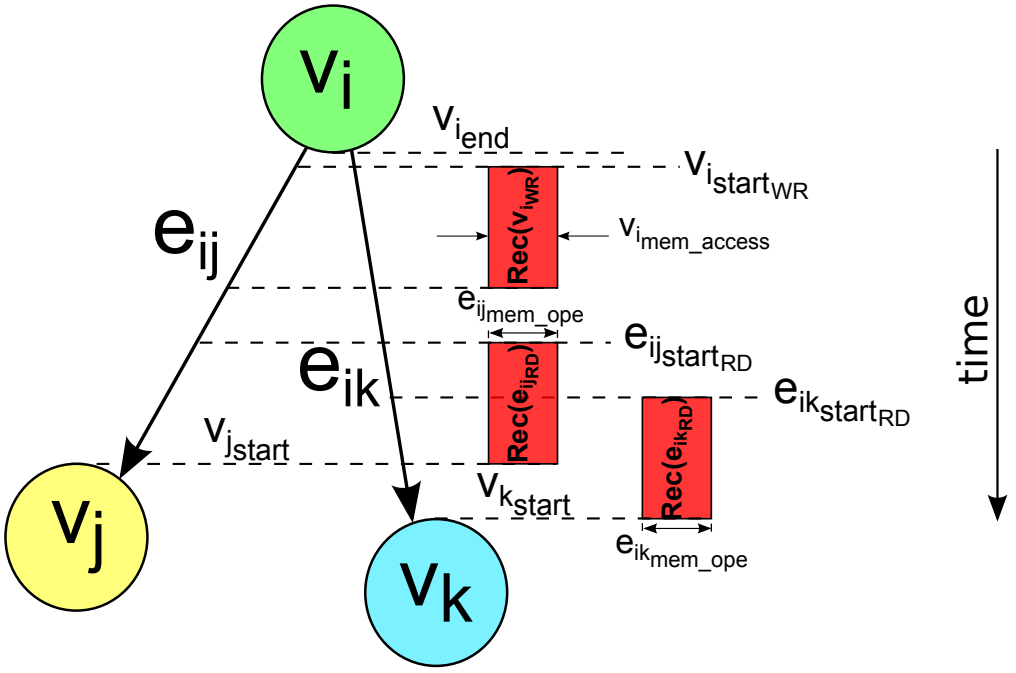


FIG. 4.12 – Opérations d'écriture et de lecture lors de deux transferts de données via la mémoire $v_{i_{mem}}$ avec exception sur les deux lectures.

$CostFunc - v_{i_{start_{WR}}}$. La variable $CostFunc$ représente la date de la fin de l'exécution de l'AG.

Le nombre de cellules par mémoire étant limité, nous utilisons la contrainte globale cumulative (définie dans le chapitre 2 en 2.1.2.3) pour assurer qu'à aucun moment cette limite n'est dépassée. Cette contrainte est définie pour chaque mémoire par une limite, ici notée m_size, et un ensemble de rectangles $Rec = [t, \Delta t, m\_used]$ représentant l'occupation d'une cellule mémoire pour une donnée à partir de la date t, pendant Δt. Un rectangle de ce type est défini pour chaque nœud de l'AG pour toutes les mémoires comme le montre la contrainte 4.4.16(a). La variable m_used vaut 1 si c'est bien la mémoire m_i qui est utilisée pour stocker la donnée en sortie v_i, m_used vaut alors 0 pour toutes les autres mémoires (contrainte 4.4.16(b)). La contrainte cumulative est définie en 4.4.16(c).

Contrainte 4.4.16 $\forall m_i \in \{0..|Mem|-1\},\ \forall v_i \in V \wedge v_i \notin ES$:

$$Rec(m_i, v_i) = [v_{i_{start_{WR}}}, v_{i_{life\_time}}, m\_used_i] \quad \text{(a)}$$

$$m\_used_i \in \{0,1\} \Leftrightarrow v_{i_{mem}} = m_i \quad \text{(b)}$$

$$\texttt{Cumulative}(Rec(m_i, v_i), m\_size) \quad \text{(c)}$$

A noter que dans le cas où v_i représente une variable d'entrée, la cellule mémoire est occupée dès le début de l'exécution de l'AG donc $Rec(m_i, v_i) = [0, v_{i_{life\_time}}, m\_used_i]$.

La figure suivante 4.13 montre un exemple d'occupation d'une cellule mémoire. Dans cet exemple, v_k est le dernier nœud à utiliser la donnée produite par v_i donc $v_{i_{life\_time}} = e_{ik_{life\_time}}$.

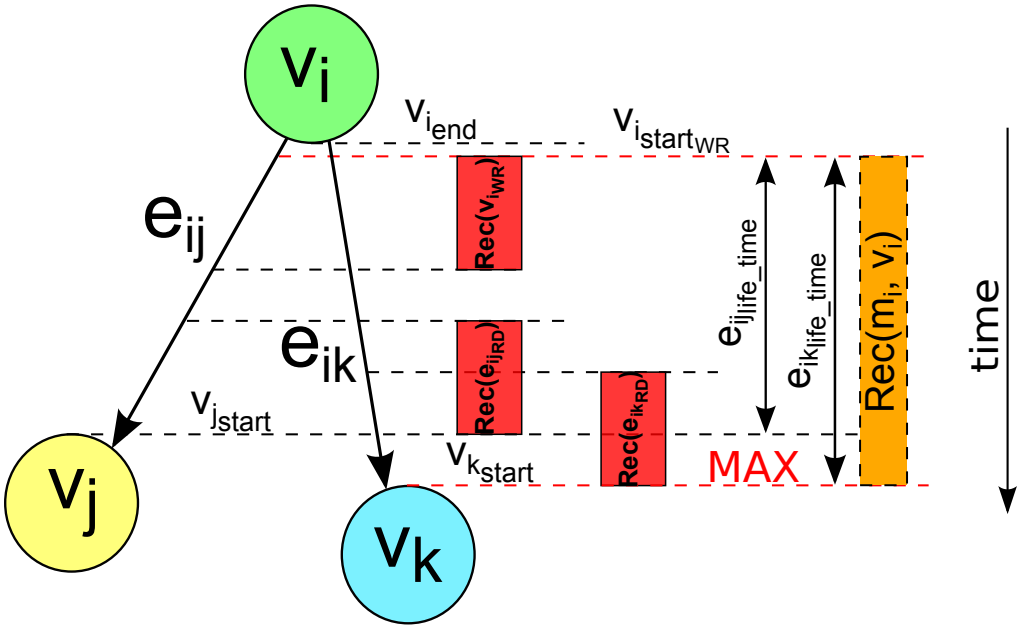


FIG. 4.13 – Occupation d'une cellule de la mémoire m_i représentée par le rectangle $Rec(m_i, v_i)$.

Modélisation de l'activité des opérateurs Lorsqu'un nœud représente l'exécution d'un motif de calcul, il faut assurer qu'il est placé sur un opérateur disponible. La modélisation des contraintes de partage des opérateurs est décrite par la contrainte « *Diff2* ». L'activité des opérateurs est modélisée par les rectangles $Rec(v_{i_{op}})$ représentant l'occupation d'un opérateur pour un nœud $v_i \in OP$, comme le montre la contrainte 4.4.17.

Contrainte 4.4.17 $\forall v_i \in OP$:

$$Rec(v_{i_{op}}) = [v_{i_{start}}, v_{i_{op}}, v_{i_{op_{activity}}}, 1]$$
$$\texttt{Diff2}([..., Rec(v_{i_{op}}), ...])$$

Pour modéliser l'occupation d'un opérateur, nous utilisons la variable $v_{i_{op_{activity}}}$ dont la valeur est définie en fonction du réseau d'interconnexion utilisé pour transférer la variable produite par v_i. En effet, un opérateur est actif depuis le début de l'exécution d'un motif jusqu'à la dernière utilisation de la donnée qu'il produit en sortie. Or, dans le cas du transfert de cette donnée via le réseau mémoire, on considère que la donnée est consommée à la fin de son écriture en mémoire alors que, dans l'autre cas, elle n'est considérée comme consommée qu'après la fin de son transfert sur le réseau fonctionnel. Deux variables $v_{i_{activity\_1}}$ et $v_{i_{activity\_2}}$ modélisent l'activité d'un opérateur en fonction du réseau, elles sont définies par les contraintes 4.4.18(a) pour le réseau mémoire et 4.4.18(b) pour le réseau fonctionnel. Ces variables permettent de définir l'occupation de l'opérateur $v_{i_{op_{activity}}}$ par la contrainte 4.4.18(c). Le choix de réseau est modélisé par les variables booléennes $v_{i_{mem\_acces}}$ (définie en 4.4.11(b)) et $v_{i_{net\_acces}}$ (définie en 4.4.18(d)).

Contrainte 4.4.18 $\forall v_i \in OP,\ e_{ij} \in v_{i_{outputs}}$:

$$v_{i_{activity\_1}} = v_{i_{start_{WR}}} + WR_{lat} - v_{i_{start}} \quad \text{(a)}$$

$$v_{i_{activity\_2}} = max(..., \Delta e_{ij_{ope\_ope}}, ...) + v_{i_{delay}} \quad \text{(b)}$$

$$v_{i_{op_{activity}}} = max(v_{i_{activity\_1}} * v_{i_{mem\_acces}}, v_{i_{activity\_2}} * v_{i_{net\_acces}}) \quad \text{(c)}$$

$$v_{i_{net\_acces}} \in \{0,1\} \Leftrightarrow \sum_{\forall e_{ij} \in v_{i_{outputs}}} e_{ij_{ope\_ope}} > 0 \quad \text{(d)}$$

La figure 4.14 montre un exemple dans lequel le nœud source v_i transfère sa donnée en sortie vers un nœud v_j via le réseau mémoire, et vers deux nœuds v_k et v_l via le réseau fonctionnel. L'activité de l'opérateur $v_{i_{op_{activity}}}$ débute à $v_{i_{start}}$ et dure jusqu'à la dernière utilisation de la donnée, qui dans cet exemple, correspond à la fin du dernier transfert de cette donnée via le réseau fonctionnel soit $v_{l_{start}}$.

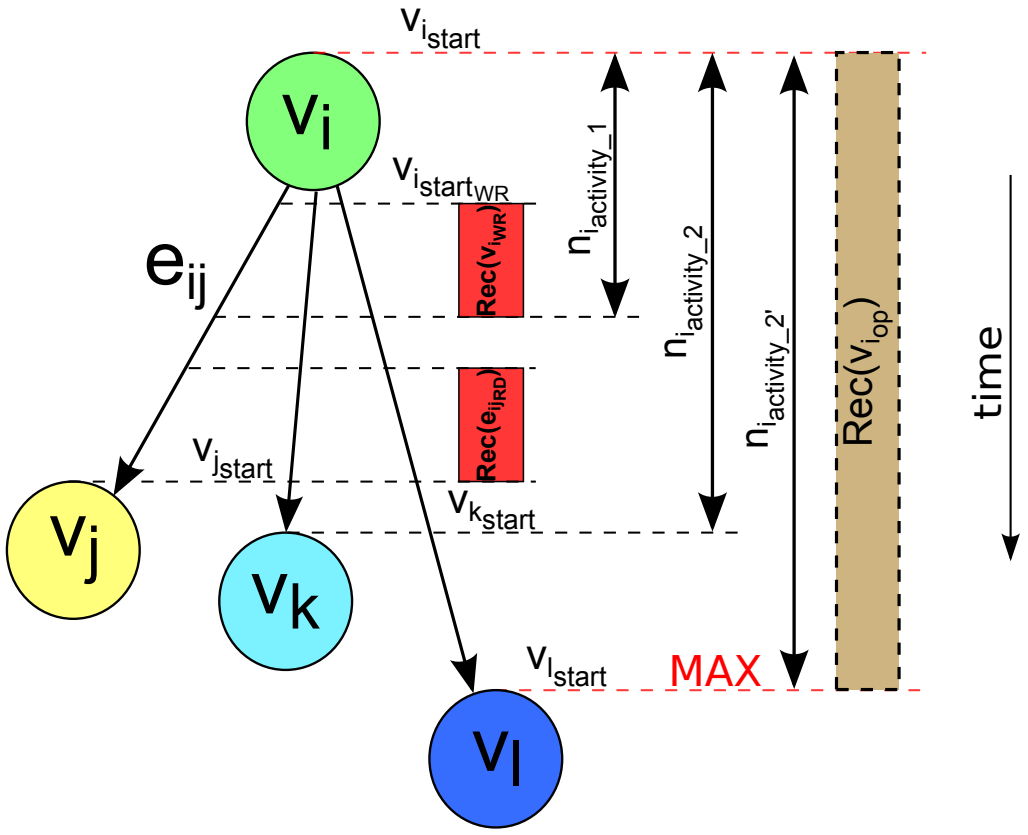


FIG. 4.14 – Exemple illustrant l'activité d'un opérateur lorsque la donnée qu'il produit est transférée via les deux réseaux de communication.

4.4.1.8 Fonction de coût

L'objectif de ce modèle est d'optimiser le temps d'exécution d'une application lors de son déploiement. Nous avons pour cela introduit la variable $CostFunc$ représentant la date de fin d'exécution du dernier nœud de sortie (c.-à-d. la dernière écriture en mémoire). Cette variable doit être minimisée comme le montre la fonction de coût définie par la contrainte 4.4.19.

Contrainte 4.4.19 $\forall v_i \in ES \mid e_{ij} \in v_{i_{outputs}} = \emptyset$:

$$CostFunc = max(..., v_{i_{end}}, ...)$$
$$minimize(CostFunc)$$

4.4.1.9 Exemple détaillé

Nous proposons dans cette section de présenter un exemple simple issu de [91] dans le but de montrer une vision plus globale du modèle de contraintes. Cet exemple, représenté par la figure 4.15, illustre le modèle de contraintes avec quatre communications, deux sur chaque réseau.

L'activité mémoire est représentée dans un espace bi-dimensionnel avec pour abscisse les identifiants des mémoires et pour ordonnée le temps. Toutes les principales variables impliquées dans l'activité mémoire sont représentées. L'opération d'écriture de la donnée produite par le nœud source v_i dans la mémoire $v_{i_{mem}}$ qui dure WR_{lat} ainsi que les deux opérations de lecture de cette donnée par les nœuds v_j et v_k. L'occupation de la cellule mémoire pour y stocker la donnée commence au début de son écriture jusqu'à sa dernière lecture par v_k.

Le temps d'occupation du réseau fonctionnel est aussi représenté sur la figure. Il correspond au temps maximum entre la fin d'exécution de v_i et le début d'exécution des nœuds v_l et v_m.

L'activité des opérateurs est aussi représentée dans un espace bi-dimensionnel avec pour abscisse les identifiants des opérateurs. L'opérateur $v_{i_{op}}$ est actif du début d'exécution du nœud v_i jusqu'à l'utilisation de la donnée produite, c.-à-d. jusqu'à la fin du dernier transfert, que ce soit en mémoire (fin d'une écriture en mémoire) ou vers un opérateur (date de début d'exécution du nœud destinataire).

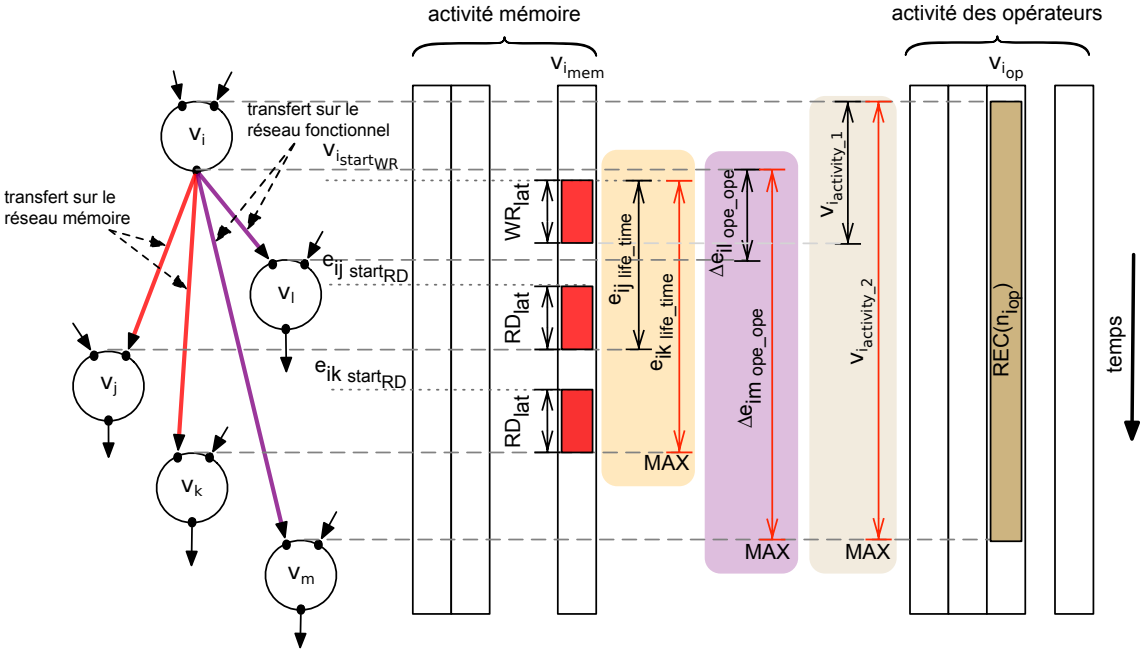


FIG. 4.15 – Illustration du modèle de contraintes avec communications sur les deux réseaux.

4.4.2 Résultats

4.4.2.1 Conditions d'expérimentation

Dans le but d'évaluer notre modèle de contraintes pour le déploiement d'un graphe d'application sur l'architecture ROMA, nous avons compilé certains noyaux de calcul intensif très répandus dans les applications multimédia. Ces résultats ont été présentés à la conférence internationnale DASIP (*Design and Architectures for Signal and Image Processing*) en 2010 [91]. Nous avons reçu à ce titre le prix du meilleur papier.

Les applications sélectionnées sont listées ci-après.

- DCT inverse (IDCT) de l'application de compression d'image JPEG,
- Écriture des en-têtes des fichiers image de type BMP,
- Algorithme de Sobel utilisé en traitement d'image pour la détection de contours,
- Multiplication de matrices issue de l'application MESA qui est une implémentation open source de la spécification openGL,
- Filtre à réponse impulsionnelle infinie (IIR pour *Infinite Impulse Response*),
- Pré-filtre de l'application PRiME[9] (FIR avec décimation) de Technicolor (noté *Roma H filter*) choisi pour valider le prototype du processeur ROMA.

L'instance de l'architecture ROMA utilisée pour ces tests comporte quatres opérateurs homogènes et huit mémoires locales. Les réseaux de communication de l'architecture sont ceux décrits précédemment, à savoir, un réseau tout connecté entre mémoires et opérateurs et le réseau ROMA entre opérateurs. Concernant les latences utilisées pour nos expérimentations, nous avons considéré que $WR_{lat} = RD_{lat} = ope\_ope_{lat} = 1$ et que toutes les opérations de base ont une latence d'un cycle à l'exception de la multiplication qui a une latence de deux cycles.

Le modèle de contraintes présenté prend en compte les problèmes liés au choix du réseau, les contraintes temporelles et d'ordres mais aussi de partage des ressources, de limitation du nombre de cellules mémoire et enfin d'optimisation du temps d'exécution global. Si nous imposons toutes ces contraintes simultanément, cela implique une grande complexité du problème et donc un temps de résolution qui peut être conséquent. Or, dans le cadre du projet ROMA, nous avons focalisé nos expérimentations sur le déploiement pour la génération de code. Nous avons observé que la contrainte cumulative modélisant la limitation du nombre de cellules contenues dans chaque mémoire n'apportait pas d'intérêt car la quantité de données traitée n'était pas un point dur. De plus, la génération d'adresse et la vérification du non dépassement du nombre de cellules mémoire ont été reportées au niveau de la génération des configurations.

4.4.2.2 Analyse des résultats

Le résultat du déploiement pour la première composante de l'application JPEG IDCT (col) est illustré sur la figure 4.16. L'identifiant de l'opérateur est donné pour chaque nœud représentant une opération. Au niveau des arcs, le label « F » signifie que le transfert de données est effectué à travers le réseau fonctionnel et « Mx » à travers le réseau mémoire (x représente l'identifiant de la mémoire).

[9] PRiME signifie *Pixel Recursif Motion Estimation*, estimation de mouvement récursif au niveau pixel.

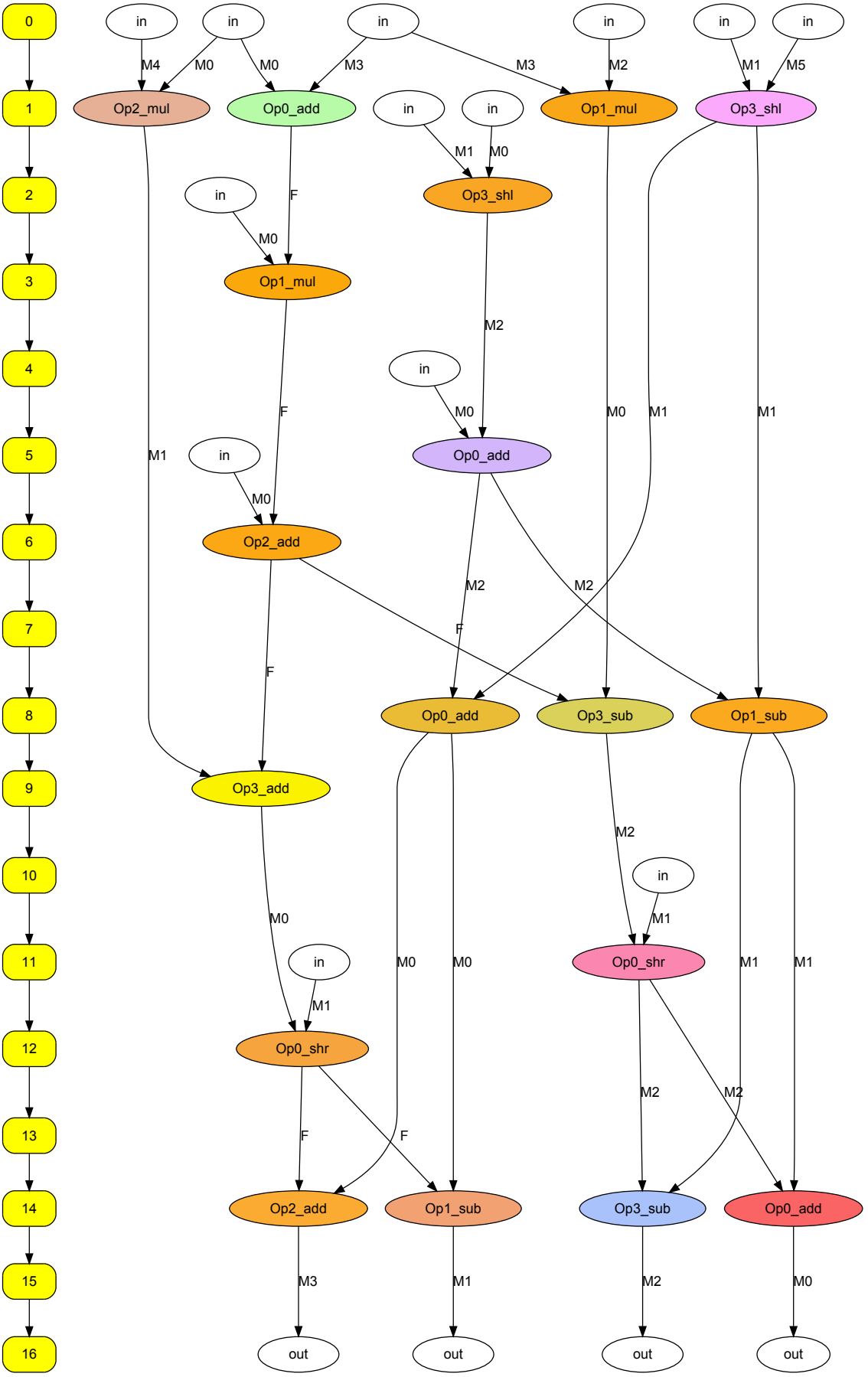


FIG. 4.16 – Exemple de déploiement pour l'application JPEG IDCT (col, composante 1), « F » signifie que le transfert de données est effectué à travers le réseau fonctionnel et « Mx » lorsqu'il est à travers le réseau mémoire (x est l'identifiant de la mémoire).

Le tableau 4.4 présente les résultats obtenus pour le déploiement des algorithmes choisis sur l'instance de ROMA considérée. Ce tableau donne les principales caractéristiques des AG représentant le cœur de calcul de chaque algorithme en termes de nombre de nœuds, d'arcs, et de variables d'entrée/sortie. Les résultats sont donnés en nombre de cycles (valeur de la variable « *CostFunc* »). Nous précisons aussi si le résultat est prouvé comme optimal par le solveur JaCoP. Enfin, le temps de résolution du problème de déploiement est donné en milliseconde.

Les résultats montrent que dans la majorité des cas, 78% exactement, les résultats du déploiement sont prouvés optimaux. Il y a trois cas où l'optimalité n'a pas pu être prouvée.

Dans le premier cas (JPEG row), mis à part la grande taille de l'AG, cela vient du fait qu'il existe dans l'AG des arcs qui relient des nœuds dont les dates d'exécution sont très éloignées dans le temps. De plus, ce graphe contient des groupes de nœuds ayant des connexions entre eux à différents niveaux. Dans les deux autres cas (MESA et IIR) les caractéristiques des AG sont particulières. Pour MESA, l'AG a une largeur maximum de 13 et surtout une largeur moyenne de 8,67. Pour IIR, l'AG est long avec un chemin critique de 23 et comporte comme JPEG row de longues connexions.

Ces résultats montrent bien les limites de notre approche si l'on ne tient pas compte des propriétés du graphe qui potentiellement amènent une grande complexité du problème. En effet, l'analyse de ces propriétés nous amène à penser que la division de l'AG en composantes dont on maitrise les propriétés permettrait de résoudre le problème de déploiement sur des graphes comme JPEG row, MESA ou IIR.

Dans nos expérimentations, les tailles des AG utilisés ne dépassent pas la centaine de nœuds. Cela vient du fait que notre approche, sans *clustering*, ne permet pas de traiter de grands problèmes. Il réside donc, dans certains cas, une problématique de passage à l'échelle que l'on rencontre aussi dans les appoches de type solveur (ILP, MIP, etc.).

Application	AG	Nœuds	Arcs	Entrées	Sorties	Cycles	Optimal	Tps d'exécution (ms)	Time Out (s)
JPEG IDCT (col)	1	35	40	13	4	16	√	7693	30
-//-	2	57	65	22	5	26	√	15117	30
Total pour JPEG IDCT (col)	1+2	92	105	35	9	42	√	22810	30
JPEG IDCT (row)	3	106	127	34	17	29	×	TO	30
Write BMP Header	4	73	72	29	16	13	√	875	10
-//-	5	19	18	8	4	5	√	15	10
-//-	6	27	26	12	4	9	√	47	10
-//-	7	27	26	12	4	9	√	46	10
-//-	8	9	8	4	2	5	√	0	10
Total pour Write BMP Header	4+..+8	155	150	65	30	41	√	983	10
sobel 7x7 (unrolled 2x2)	9	52	54	24	2	24	√	360	10
MESA Matrix Mul	10	52	60	20	4	16	×	TO	30
IIR biquad N sections (unrolled x4)	11	66	73	29	1	55	×	TO	30
Roma H filter	12	43	42	21	2	28	√	297	10

TAB. 4.4 – Résultats obtenus pour le déploiement sous contraintes de ressources d'une sélection d'applications multimédia.

4.5 Déploiement d'une application sur un modèle d'architecture pipelinée

Nous nous intéressons dans cette partie à la modélisation par des contraintes du déploiement d'un AG sur un modèle d'architecture pipelinée avec pour objectif l'optimisation de la latence totale du pipeline ou encore l'optimisation de la puissance consommée lors de l'exécution de l'AG.

4.5.1 Modèle d'architecture pipelinée

Le modèle d'architecture considéré dans cette partie comporte globalement les mêmes caractéristiques que dans la partie précédente, un ensemble d'opérateurs reconfigurables à gros grain au niveau fonctionnel, un ensemble de mémoires, deux réseaux de communication et un contrôleur global. Les principaux changements sont au niveau des opérateurs et des mémoires. En effet, les opérateurs sont pipelinés et les mémoires, à double port, peuvent se comporter comme des buffers circulaires. Ce modèle d'architecture est représenté sur la figure 4.17

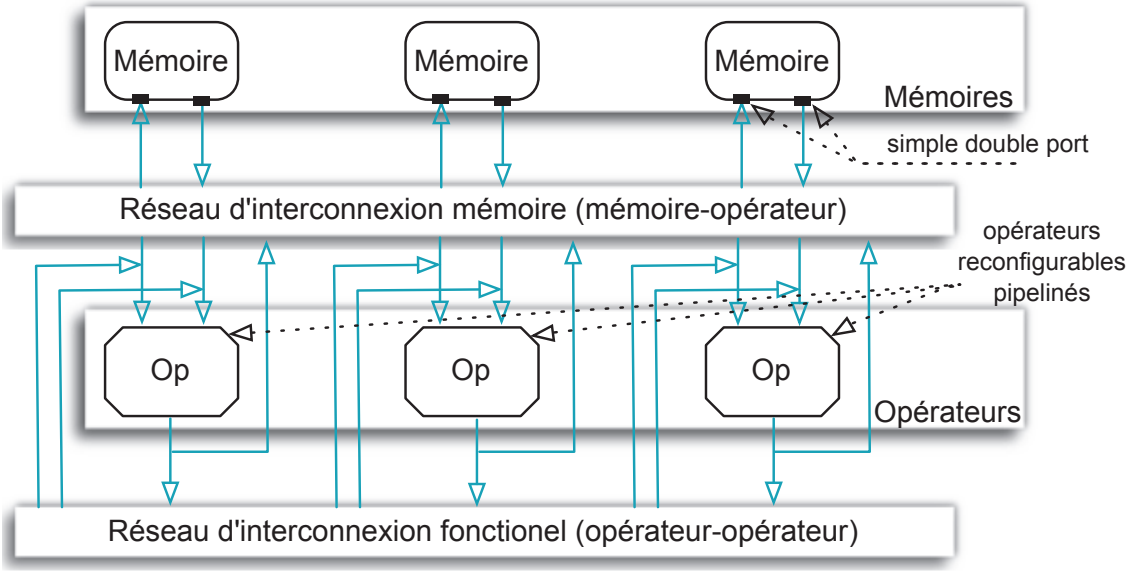


FIG. 4.17 – Modèle générique d'architecture pipelinée.

4.5.1.1 Opérateur pipeliné

Un opérateur pipeliné décompose l'exécution d'une opération ou d'un motif (un ensemble d'opérations successives) en plusieurs étapes. Cette caractéristique lui permet de traiter plusieurs données en parallèle plutôt que d'attendre la fin d'un traitement pour en

démarrer un autre. D'un point de vue matériel, un opérateur pipeliné est composé de plusieurs étages, chaque étage est délimité par un ensemble de « registres d'étage » assurant ainsi un point de synchronisation.

La figure 4.18 illustre, par un exemple simple, la différence entre un opérateur pipeliné et un opérateur non pipeliné en termes d'exécution et de matériel. Dans le cas d'un opérateur multi-cycle non pipeliné (combinatoire), ayant une latence de 3 cycles, une opération peut être lancée tous les 3 cycles. Dans le cas d'un opérateur pipeliné (comme celui représenté sur la figure 4.18), une opération peut être lancée à tous les cycles.

Cet exemple démontre qu'un opérateur pipeliné est plus efficace en termes de performance temporelle (avec un ordonnancement en 5 cycles) qu'un opérateur qui ne l'est pas (avec 9 cycles). De manière générale, si l'on compare deux architectures, l'une pipelinée et l'autre non[10], un noyau de calcul sous forme de boucle(s) peut être exécuté en un nombre de cycle égal à $2 * (L_A - 2) + I$ (L_A est la latence totale de l'architecture A et I est le nombre d'itérations de la boucle) au lieu de $L_A * I$ cycles pour l'architecture non pipelinée. Mais cet avantage n'est pas sans certains inconvénients.

Tout d'abord, un opérateur pipeliné a une surface plus grande comparée à un opérateur combinatoire du fait de la présence de registres d'étage. Ensuite, pour pouvoir exploiter au maximum un opérateur pipeliné il faut l'alimenter en données à chaque cycle.

L'utilisation d'opérateur de ce type dans le modèle d'architecture permet une exécution pipelinée d'un AG ce qui augmente les performances de l'architecture. Mais la problématique de déploiement d'un AG dans ce contexte est différente.

4.5.1.2 Mémoire à buffer circulaire

Le modèle d'architecture considéré comporte des mémoires à double port. Chaque mémoire dispose un port d'écriture et un port de lecture ce qui permet d'effectuer une écriture et une lecture simultanées à deux adresses différentes. Dans le contexte d'une exécution pipelinée, on réalise à l'intérieur de chaque mémoire un buffer circulaire dont la taille correspond à la latence nécessaire entre l'écriture et la lecture d'une donnée.

La figure 4.19 représente deux buffers circulaires effectuant respectivement une ligne à retard d'un cycle (le buffer contient une donnée) et de trois cycles (il contient alors trois données) entre l'écriture et la lecture d'une même donnée. Un générateur d'adresse capable de générer des adresses du type « *modulo C* » (C est une constante) est alors nécessaire (*mod* 1 et *mod* 3 sur la figure).

4.5.2 Problématique

Lors d'une exécution pipelinée, les données traversent l'architecture à la chaîne à travers un certain nombre d'étages (selon la profondeur de la chaîne) et cela de manière synchronisée pour assurer la cohérence des résultats. La chaîne est cadencée par l'horloge de l'architecture. Idéalement, les données en entrée sont transmises à chaque cycle et, après une certaine latence correspondant au temps nécessaire pour traverser tous les étages, les données produites sont alors fournies en sortie à chaque cycle. Ce modèle d'exécution nécessite de prendre en considération les aspects suivants.

[10] Les opérateurs fonctionnent dans les deux cas à la même fréquence.

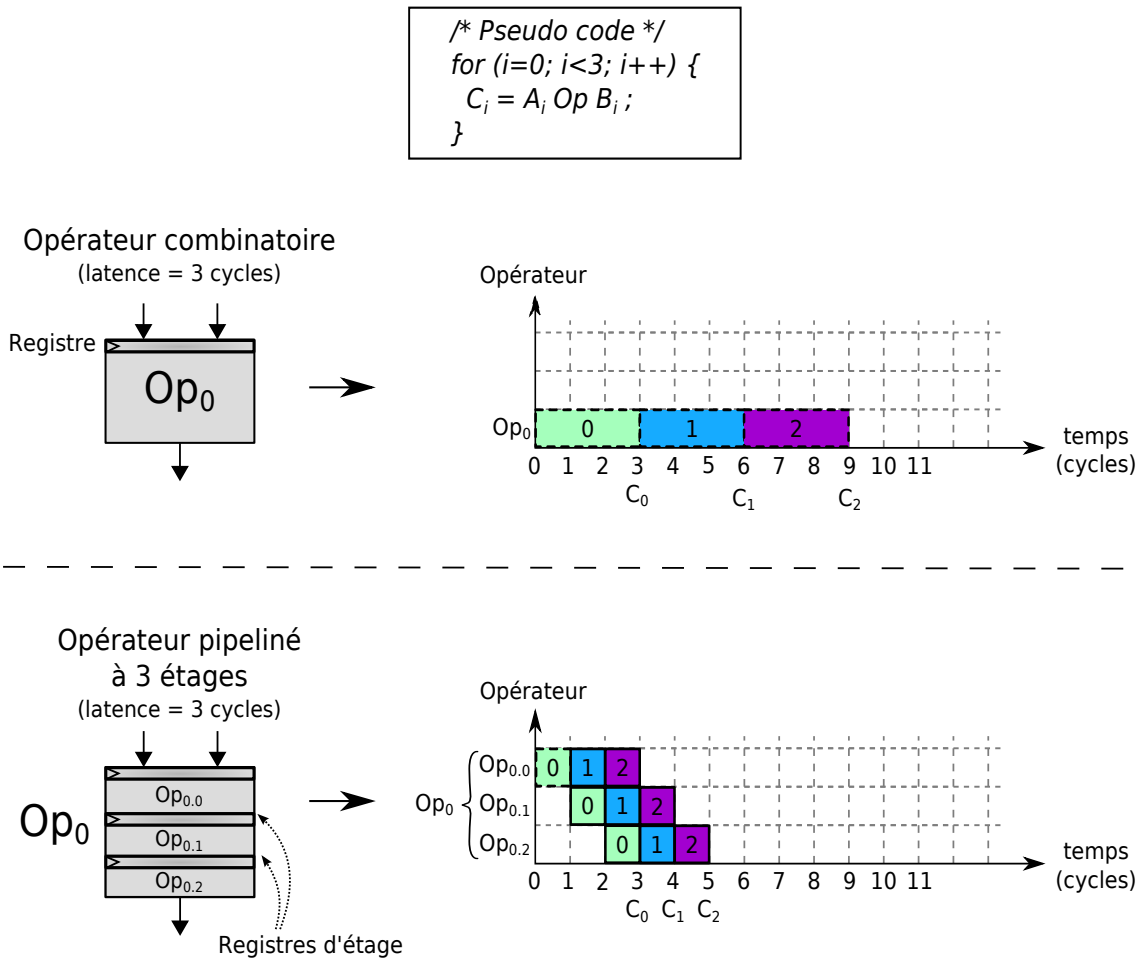


FIG. 4.18 – Différence entre opérateur pipeliné et non pipeliné en termes d'exécution sur un exemple d'ordonnancement simple.

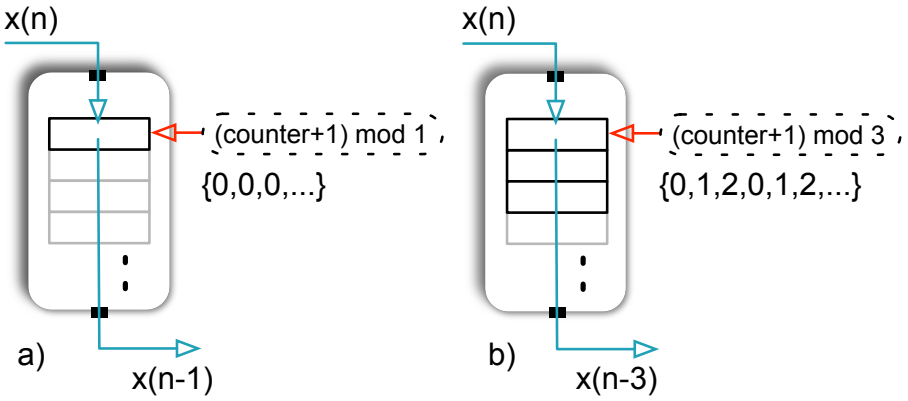


FIG. 4.19 – Modélisation d'un buffer circulaire. Ce type de mémoire permet d'imposer une latence entre l'écriture et la lecture d'une donnée. Dans les deux exemples illustrés sur cette figure, les latences sont de 1 et 3 cycles.

– Il n'est plus possible de reconfigurer partiellement l'architecture pendant l'exécution d'un AG car cela implique une rupture dans la chaîne d'exécution. En effet, il est nécessaire d'arrêter l'approvisionnement des données en entrée et de récupérer les dernières données produites après leur traversée des étages du pipeline. La reconfiguration partielle requiert dans ce contexte un contrôle supplémentaire qui n'est pas considéré dans notre modèle d'architecture. Une exécution pipelinée implique donc un déploiement de l'AG en une seule configuration.

– La synchronisation des données en entrée de chaque opérateur à chaque cycle doit être assurée pour permettre une exécution efficace et pour assurer que les résultats produits soient corrects. Or, dans notre modèle d'architecture le transfert de données peut être effectué à travers deux réseaux de communication : entre les opérateurs (réseau fonctionnel) et entre les opérateurs et les mémoires (réseau mémoire). Concernant le réseau mémoire, une donnée peut être lue ou écrite en mémoire à chaque cycle, de plus, la durée entre l'écriture et la lecture d'une même donnée peut varier ce qui permet une certaine flexibilité lors de l'ordonnancement. Mais concernant le réseau fonctionnel, il ne comporte pas de registre et son délai de traversée est négligeable par rapport au délai nécessaire pour effectuer un calcul à l'intérieur dun cycle. On peut alors considérer que sa traversée à une latence nulle ce qui impose plus de contraintes lors de l'ordonnancement. La présence de ces deux réseaux dont les propriétés diffèrent nécessite de modéliser la synchronisation des données en fonction du réseau emprunté.

– La présence d'opérateurs à accumulation interne dans l'architecture pose certains problèmes dans le contexte d'une exécution pipelinée. En effet, ce type d'opérateurs retient la donnée à accumuler durant un certains temps ce qui se traduit par la consommation de données en entrée à chaque cycle sans production de donnée en sortie à la même cadence. Dans notre modèle pour une exécution pipelinée, cela ne pose pas de problème dans les cas où les opérateurs à accumulation ne transmettent pas les données produites à un autre opérateur. En d'autres termes, le nœud dans l'AG représentant une accumulation doit être terminal (c.-à-d. dont l'arc de sortie est uniquement relié à un nœud représentant une variable).

4.5.3 Exemple illustratif

Un exemple illustrant la problématique de déploiement d'un AG dans le contexte d'une exécution pipelinée est donné figure 4.20. Cet exemple montre un ordonnancement de l'AG représenté sur la figure. On considère que la latence d'un opérateur pour effectuer une addition est de un cycle, de deux cycles pour une multiplication. On considère la traversée du réseau fonctionnel comme immédiate et la latence d'une lecture/écriture en mémoire est de un cycle.

Dans un contexte mono-configuration il n'y a pas de partage de ressources donc chaque nœud de l'AG représentant une opération est placé sur un opérateur durant toute la configuration.

Dans le but d'assurer qu'un opérateur reçoive les bonnes données au bon moment, il est parfois nécessaire de retarder les données produites par les opérateurs source en les faisant passer par les bancs de mémoire. C'est ce qu'il se passe pour les opérateurs Op_0 et Op_4 dont les données produites doivent passer par les bancs de mémoire avant d'être transmises respectivement aux opérateurs Op_1 et Op_3. Notons que l'addition représentée

par le nœud v_1 sur l'opérateur Op_4 est ordonnancée au cycle 1 plutôt qu'au cycle 0. En effet, il est possible de définir l'instant de lecture des données à additionner au début de l'exécution et d'éviter ainsi de retarder les données produites par l'opérateur Op_4 pour être synchronisées en entrée de l'opérateur Op_2.

Lorsque les données doivent transiter par les bancs de mémoire lors d'un transfert entre deux opérateurs, il est indispensable d'emprunter le réseau mémoire et donc d'interdire l'utilisation du réseau fonctionnel.

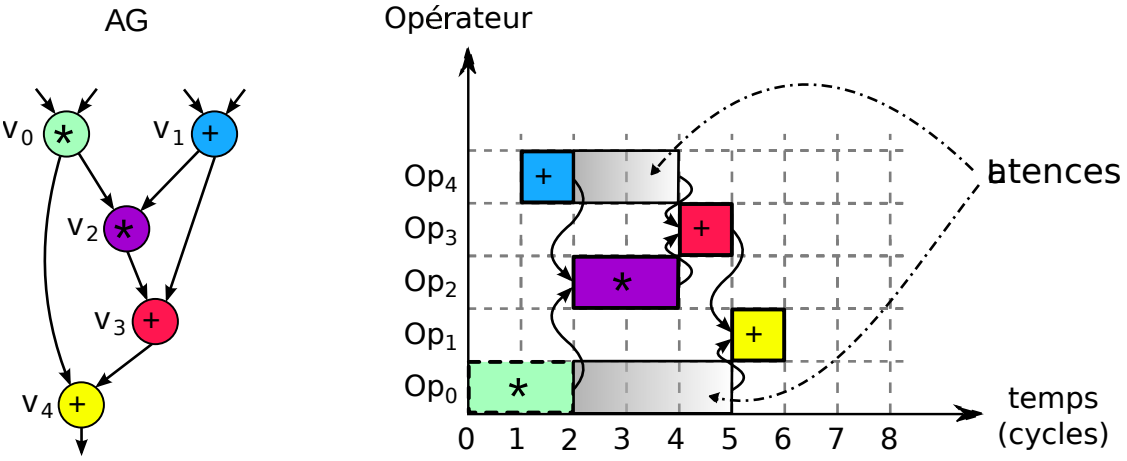


FIG. 4.20 – Exemple illustrant la problématique de déploiement d'un AG dans le contexte d'une exécution pipelinée : pas de partage de ressources et synchronisation des données en entrée de chaque opérateur par l'ajout de latence.

4.5.4 Couverture de l'AG avec des motifs de calcul

Ce modèle vise à permettre la compilation d'un AG pour une exécution pipelinée en minimisant la latence totale du pipeline ou la consommation d'énergie. Pour atteindre cet objectif, nous proposons de couvrir l'AG à placer par des motifs de calcul représentant des opérateurs optimisés. Ces derniers sont modélisés sous forme de graphes et sont disponibles dans un bibliothèque.

La couverture de l'AG par des motifs de calcul a deux principaux intérêts. Le premier a trait à l'absence de partage de ressources. En effet, le nombre d'opérateurs dans l'architecture est fixe et un opérateur est alloué au même traitement durant toute l'exécution de l'unique configuration. De plus, le nombre de liens sur les réseaux de communication est limité. Les motifs présents dans la bibliothèque ont des tailles différentes, variant du simple opérateur arithmétique ou logique au motif comportant plusieurs opérateurs reliés en interne. La couverture de l'AG par ces motifs va permettre dans certains cas de répondre à la contrainte de déploiement en une configuration grâce à l'utilisation de motifs comportant plusieurs opérateurs reliés en interne, ce qui n'est pas possible si la couverture est effectuée à une étape antérieure dans le flot de déploiement. C'est d'ailleurs le choix qui a été fait dans le modèle précédent pour assurer une complexité du problème de déploiement acceptable. La conséquence directe est un meilleur passage à l'échelle.

Le second intérêt est lié à l'optimisation de la latence du pipeline. Les motifs utilisés pour couvrir l'AG diffèrent dans leur taille mais aussi dans leur composition et surtout

dans le nombre d'étages qu'ils comportent. Or, notre objectif étant d'obtenir le déploiement ayant la latence la plus faible, le fait d'effectuer la couverture en même temps que le déploiement permet une sélection des motifs qui optimise la latence totale de l'unique configuration.

Remarque : A noter que dans ce nouveau modèle d'architecture, le nombre d'entrées d'un opérateur peut être supérieur à deux (le nombre de sorties reste égal à un comme imposé par le modèle d'architecture ROMA). Cela permet aux opérateurs de supporter une variété de motifs plus grande.

4.5.5 Modèle de contraintes

Les contraintes définies dans ce modèle sont présentées selon les aspects suivants.

- Couverture de l'AG par des motifs de calcul,
- déploiement de l'AG en une configuration :
  - allocation des opérateurs,
  - transfert de données sur les deux réseaux de communication,
- aspects temporels :
  - dépendances de données,
  - synchronisation des données dans le pipeline,
- optimisation de la latence du pipeline ou de la consommation d'énergie.

Le modèle de contrainte pour la couverture de l'AG s'appuie sur les travaux d'Antoine Floc'h. Ceux-ci ne sont pas détaillés dans ce manuscrit. Pour plus de détails sur ce modèle, le lecteur peut se référer à [42].

Un ensemble de variables sont définies pour modéliser les nœuds et les arcs dans l'AG ainsi que l'appariement entre un motif de calcul de la bibliothèque et un sous-graphe dans l'AG.

4.5.5.1 Modélisation d'un appariement

Dans notre modèle de contraintes, l'appariement entre un motif de calcul issu de la bibliothèque de motifs et le sous-graphe dans l'AG isomorphe à ce motif est modélisé. Il est noté « a » et est défini par les variables du tableau 4.5.

4.5.5.2 Modélisation d'un nœud

Un nœud $v \in V$ a la même sémantique que dans le modèle précédent. On modélise en plus l'appariement avec le motif de calcul dans lequel ce nœud est impliqué par la variable notée v_a. Le domaine de cette variable est composé des identifiants des appariements pouvant couvrir v, soit $\{id | a_{id} \in appariements_v\}$. Une procédure est utilisée pour initialiser l'ensemble $appariements_v$, elle est présentée un peu plus loin dans le manuscrit (partie 4.5.5.4).

a_{id}	Identifiant de l'appariement	$\{0..\|appariement\|-1\}$
a_{sel}	Représente la sélection de l'appariement	$\{0,1\}$
a_{op}	Identifiant de l'opérateur utilisé pour exécuter le motif représenté par a	$\{i \mid op_i \in \{0..\|operateur\|-1\} \wedge op_i$ peut exécuter $a\}$
a_{lat_in}	Date à laquelle débute l'exécution de l'appariement a dans le pipeline	$\{0..\infty\}$
a_{lat}	Latence de l'opération associée à l'appariement	$\{0..\infty\}$
a_{pwr}	Puissance consommée par l'opération associée à l'appariement	$\{0..\infty\}$

TAB. 4.5 – Variables associées à un appariement, leur signification ainsi que leur domaine d'initialisation.

Le tableau 4.6 donne les principales variables associées à un nœud, leur signification ainsi que leur domaine d'initialisation.

v_a	Identifiant de l'occurrence de l'appariement sélectionnée pour couvrir v	$\{id \mid a_{id} \in appariements_v\}$
v_{lat_in}	Date à laquelle débute l'exécution du nœud v	$\{0..\infty\}$
v_{lat}	Latence de l'opération associée au nœud v	$\{0..\infty\}$
v_{op}	Identifiant de l'opérateur utilisé pour exécuter l'appariement couvrant le nœud $v \in OP$	$\{i \mid op_i \in \{0..\|operateur\|-1\} \wedge op_i$ peut exécuter $v_a\}$

TAB. 4.6 – Variables associées à un nœud, leur signification ainsi que leur domaine d'initialisation.

4.5.5.3 Modélisation d'un arc

De la même manière que précédemment, un arc $e_{ij} \in E$ dans le graphe d'application représente un transfert de données entre deux nœuds v_i et v_j. Les variables associées à un arc sont données dans le tableau 4.7. Une nouvelle variable notée e_{int} modélise un transfert interne à un appariement (c.-à-d. à un opérateur), dans ce cas aucun des deux réseaux de communication n'est emprunté.

4.5.5.4 Contraintes de couverture de l'AG par les motifs de calcul issus de la bibliothèque

La sélection des motifs pour couvrir l'AG est effectuée à partir des motifs de la bibliothèque représentant des opérateurs pipelinés optimisés. Une étape clé dans la couverture est l'identification des motifs pouvant couvrir chacun des nœuds de l'AG. Le résultat de cette étape est, pour chaque nœud $v \in OP$, l'ensemble des occurrences d'appariement pouvant le couvrir noté $appariements_v$. L'algorithme utilisé dans cette étape est issu de [72]

e_{mem_ope}	Désigne l'utilisation du réseau mémoire pour le transfert de données représenté par e	$\{0, 1\}$
e_{ope_ope}	Désigne l'utilisation du réseau fonctionnel pour le transfert de données représenté par e	$\{0, 1\}$
e_{int}	Désigne un transfert de données interne à un opérateur, c.-à-d. à un appariement	$\{0, 1\}$
e_{lat}	Désigne la latence d'un transfert sur un des réseaux de communication	$\{0..\infty\}$
e_{mem}	Identifiant de la mémoire pouvant être utilisée pour le transfert des données représenté par e	$\{0..\|memoire\| - 1\}$

TAB. 4.7 – Variables associées à un arc, leur signification ainsi que leur domaine d'initialisation.

et est décrit ci-après (algorithme 2). Dans cet algorithme, le nombre d'occurrences d'un motif p dans l'AG est défini grâce à la méthode $TrouverToutesLesOccurrences(AG, p)$ implémentée en utilisant la contrainte $GraphMatch$ issue de [115]. Cette contrainte effectue l'isomorphisme de sous-graphe entre les motifs de la bibliothèque et l'AG.

Algorithme 2 Générer tous les appariements

// $AG = (V, E)$: Graphe d'application
// EMB : Ensemble de Motifs de la Bibliothèque
// A_p : Ensemble des occurences pour un motif p
// A : Ensemble de toutes les occurences
// $appariements_v$: Ensemble des occurences pouvant couvrir un nœud v
$A_p \Leftarrow \emptyset$
pour tout $p \in EMB$ **faire**
  $A_p \Leftarrow TrouverToutesLesOccurrences(AG, p)$
  $A \Leftarrow A \cup A_p$
fin pour
pour tout $a \in A$ **faire**
  pour tout $v \in V$ **faire**
    $appariements_v \leftarrow appariements_v \cup a$
  fin pour
fin pour

Afin d'assurer que tous les nœuds soient impliqués dans un seul appariement, les contraintes 4.5.1 et 4.5.2 sont imposées. Dans 4.5.1, la contrainte *Count* est utilisée pour calculer le nombre d'occurrences $a_{count} \in \{0, a_{taille}\}$ parmi l'union de tous les v_a représentant l'ensemble des appariements a pouvant couvrir v, noté ici $a_{set} = \cup_{v \in V} v_a$ ($taille$ représente le nombre de nœuds dans l'appariement a). La contrainte 4.5.2 garantit que lorsqu'un appariement a est sélectionné ($a_{sel} = 1$), cela implique que les nœuds qui composent cet appariement sont tous couverts par celui-ci ($a_{count} = taille$ et $taille > 0$ donc $a_{count} > 0$) et réciproquement.

Contrainte 4.5.1

$$\forall a_i \in A,\ a_{set} = \bigcup_{v \in a} v_a,\ a_{count} \in \{0, a_{taille}\}\ :\ Count(i, a_{set}, a_{count})$$

Contrainte 4.5.2

$$a_{count} > 0 \Leftrightarrow a_{sel}$$

4.5.5.5 Contraintes de déploiement

Les contraintes de déploiement permettent de prendre en considération les limitations de l'architecture, le nombre et type d'opérateurs, l'utilisation et la topologie des réseaux de communication et le nombre de mémoires.

Contraintes d'allocation des opérateurs La modélisation de l'allocation des opérateurs dans le contexte mono-configuration est faite avec la contrainte globale $Diff2$. La contrainte 4.5.3 modélise, pour chaque appariement a, l'allocation de l'opérateur utilisé pour exécuter le motif de l'appariement. Elle définit pour cela un rectangle $Rec(a_{op})$ dans un espace bidimensionnel où l'axe des abscisses représente le temps et l'axe des ordonnées représente l'identifiant des opérateurs. La problématique de déploiement en une configuration est modélisée par la définition de tous les rectangles sur une seule unité de temps ($x = 0$ et $\Delta x = 1$). L'allocation de l'opérateur à proprement parlé est modélisée par la variable a_{op} qui correspond à l'identifiant de l'opérateur utilisé pour exécuter le motif de l'appariement a, la hauteur du rectangle dépend de m_{sel}. En effet, si l'appariement n'est pas sélectionné alors le rectangle a une hauteur nulle, sinon, sa hauteur vaut 1. La contrainte $Diff2$ assure qu'aucun rectangle ne se chevauche.

Contrainte 4.5.3

$$\begin{aligned} \forall a \in A: \quad Rec(a_{op}) &= \{x, y, \Delta x, \Delta y, \} \\ &= \{0, a_{op}, 1, a_{sel}\} \end{aligned}$$

$$Diff2(..., Rec(a_{op}), ...)$$

Comme le montre la contrainte précédente, ce sont les appariements sélectionnés qui sont placés sur les opérateurs de l'architecture. Il est donc nécessaire d'associer, pour chacun des nœuds de l'AG, la variable v_{op} représentant l'opérateur sur lequel le nœud v est placé avec la variable a_{op} représentant l'opérateur exécutant l'appariement a pouvant couvrir v. Cette association est assurée par la contrainte 4.5.4.

Contrainte 4.5.4

$$\forall v \in OP : v_{op} = Liste_{op}[v_a]\ \ avec\ Liste_{op} = [a_{op} | a \in A \wedge v \in a]$$

Contraintes sur les transferts de données Un arc dans l'AG peut représenter trois types de transfert, via la mémoire, le réseau fonctionnel ou encore un transfert interne à un opérateur. En imposant la contrainte 4.5.5, on assure que ces trois types de transfert sont mutuellement exclusifs. Un transfert interne a lieu si deux nœuds reliés par un arc sont couverts par le même appariement (c.-à-d. placés sur le même opérateur), ceci est modélisé par la contrainte 4.5.6.

Contrainte 4.5.5

$$\forall e \in E : e_{mem\_ope} + e_{ope\_ope} + e_{int} = 1$$

Contrainte 4.5.6

$$\forall e_{ij} = (v_i, v_j) \in E,\ v_i \in OP \wedge v_j \in OP : e_{ij_{int}} \Leftrightarrow v_{i_a} = v_{j_a}$$

Dans le cas particulier où le nœud source (ou le nœud destination) représente une variable d'entrée (sortie) alors l'arc issu de (ou vers) ce nœud représente forcément un transfert sur le réseau mémoire. Dans ce cas, la contrainte 4.5.7 est appliquée.

Contrainte 4.5.7

$$\forall e_{ij} = (v_i, v_j) \in E,\ (v_i \in ES \wedge v_j \in OP) \vee (v_i \in OP \wedge v_j \in ES) : e_{ij_{mem\_ope}} = 1$$

Contraintes pour le transfert de données à travers les mémoires Dans un contexte mono-configuration, les contraintes de partage des accès aux mémoires sont largement simplifiées. En effet, non seulement comme pour les opérateurs la contrainte est ramenée à une configuration (sur une unité de temps), mais en plus il n'est pas nécessaire de modéliser les écritures et les lectures séparément. Ainsi, la contrainte 4.5.8 définit, pour chaque arc e_{ij} entre les nœuds v_i et v_j, le rectangle associé modélisant l'utilisation de la mémoire $e_{ij_{mem}}$. Comme pour l'allocation des opérateurs, l'axe des abscisses représente le temps et l'axe des ordonnées l'identifiant des mémoires.

La contrainte 4.5.8 impose, par l'utilisation de la contrainte globale *DisjointConditional* notée DC, qu'aucun rectangle ne se chevauche ce qui signifie qu'une mémoire ne peut être utilisée que pour un transfert de données entre deux opérateurs et cela tout au long de l'unique configuration. La contrainte DC, à la différence de la contrainte $Diff2$, permet de spécifier pour chaque paire de rectangles une exception en fonction d'une variable booléenne. Lorsque cette variable vaut 0 alors les deux rectangles peuvent se chevaucher, si elle vaut 1 alors ces rectangles ne peuvent pas se chevaucher.

Contrainte 4.5.8

$$\begin{aligned} \forall e_{ij} = (v_i, v_j) \in E : \quad Rec(e_{ij_{mem}}) &= \{x, y, \Delta x, \Delta y, \} \\ &= \{0, e_{ij_{mem}}, 1, e_{ij_{mem\_op}}\} \end{aligned}$$

$$DC(..., Rec(e_{ij_{mem}}), ...)$$

Il est important de noter que dans ce modèle, la contrainte 4.5.8 impose l'utilisation d'une mémoire différente pour les transferts d'une même donnée vers différents opérateurs. Cela est dû au fait que les latences à imposer lors de ces transferts peuvent être différentes. Une exception est donc imposée dans le cas où ces latences sont identiques comme le montre la contrainte 4.5.9. Dans cette contrainte, la variable booléenne nsl_{jk} (pour *not same latency*) vaut 0 si les latences sont égales ce qui valide l'exception. Si les latences sont différentes alors elle vaut 1 ce qui empêche l'utilisation d'une même mémoire pour les deux transferts.

Contrainte 4.5.9

$$\forall v_i \in E \wedge e_{ij}, e_{ik} \in v_{i_{outputs}} \wedge j \neq k :$$
$$[Rec_j, Rec_k] = [Rec(e_{ij_{mem}}), Rec(e_{ik_{mem}}), nsl_{jk}]$$
$$nsl_{jk} \Leftrightarrow e_{ij_{lat}} \neq e_{ik_{lat}}$$

Dans le cas où plusieurs arcs sont issus d'un nœud d'entrée, il est nécessaire de ne considérer l'utilisation que d'une seule mémoire. Ainsi, un seul rectangle $Rec(e_{i?_{mem}})$ est modélisé dans le cas où le nœud d'entrée v_i possède plusieurs successeurs.

Contraintes pour le transfert de données sur le réseau fonctionnel : Les contraintes de communication sur le réseau fonctionnel sont liées à la topologie du réseau. Elles sont très proches de celles présentées pour le modèle précédent.

Concernant la modélisation de la topologie ROMA, la contrainte 4.5.10 est imposée sur tous les arcs reliant deux nœuds qui représentent tous deux une opération. Pour rappel, la variable $e_{ij_{op_r}} \in \{1..\frac{|operateur|}{2}\}$ modélise les identifiants des opérateurs atteignables à partir de $v_{i_{op}}$.

Contrainte 4.5.10

$$\forall e_{ij} = (v_i, v_j) \in E,\ v_i \in OP \wedge v_j \in OP :$$
$$If\ e_{ij_{ope\_ope}} = 1\ then\ v_{j_{op}} = v_{i_{op}} + e_{ij_{op_r}}$$

Concernant la modélisation de n'importe quelle topologie d'un réseau point-à-point, la contrainte 4.5.11 est utilisée. Pour rappel, la matrice de communication, $ComMat$, spécifie toutes les connexions entre les éléments du réseau deux à deux.

Contrainte 4.5.11

$$\forall e_{ij} = (v_i, v_j) \in E,\ v_i \in OP \wedge v_j \in OP :$$
$$If\ e_{ij_{ope\_ope}} = 1\ then\ v_{j_{op}} = ComMat[v_{i_{op}}]$$

Contraintes temporelles Chaque nœud est couvert par un appariement sélectionné parmi tous ceux pouvant le couvrir. Cet appariement est exécuté par un opérateur de l'architecture. Concernant la latence à laquelle débute l'exécution de l'appariement et la latence nécessaire à son exécution, elles sont identiques pour tous les nœuds couverts par l'appariement. Cette relation est définie par les contraintes 4.5.12 et 4.5.13 dans lesquelles la liste $Liste_{lat\_in}$ contient les latences auxquelles les appariements couvrant v débutent. La liste $Liste_{lat}$ contient les latences d'exécution de ces mêmes appariements.

Contrainte 4.5.12

$$\forall v \in OP : v_{lat\_in} = Liste_{lat\_in}[v_a]\ \ avec\ Liste_{lat\_in} = [a_{lat\_in} | a \in A \wedge v \in a]$$

Contrainte 4.5.13

$$\forall v \in OP : v_{lat} = Liste_{lat}[v_a]\ \ avec\ Liste_{lat} = [a_{lat} | a \in A \wedge v \in a]$$

Dépendances de données entre nœuds et synchronisation dans le pipeline Pour assurer la synchronisation des données en entrée des opérateurs, la contrainte 4.5.14 est définie. Cette contrainte impose l'utilisation du réseau mémoire pour le transfert de données entre deux nœuds représentant des opérations fonctionnelles lorsque l'ajout d'une latence est nécessaire pour la synchronisation des données. Enfin, la contrainte 4.5.15 assure que les dépendances de données entre les nœuds de l'AG ainsi que la synchronisation des données en entrée d'un opérateur sont bien respectées, et cela, en fonction des latences des opérations et des transferts de données.

Contrainte 4.5.14

$$\forall e_{ij} = (v_i, v_j) \in E,\ v_i \in OP \wedge v_j \in OP :$$
$$e_{ij_{mem\_ope}} \Leftrightarrow e_{ij_{lat}} > 0$$

Contrainte 4.5.15

$$\forall e_{ij} = (v_i, v_j) \in E,\ v_i \in OP \wedge v_j \in OP :$$
$$If\ e_{ij_{int}} = 0\ then\ v_{j_{lat\_in}} = v_{i_{lat\_in}} + v_{i_{lat}} + e_{ij_{lat}}$$

4.5.5.6 Fonctions de coût

Dans ce modèle de contraintes, deux fonctions de coût sont proposées. Celle pour l'optimisation de latence totale du pipeline est donnée par la contrainte 4.5.16. Celle pour la minimisation de la consommation d'énergie est donnée par la contrainte 4.5.17. Cette dernière contrainte permet la sélection des motifs qui consomment le moins d'énergie et qui minimisent les transferts de données sur le réseau mémoire (la constante $memory_{pwr}$ représente la consommation d'énergie lors d'un transfert via la mémoire). Elle peut être utilisée conjointement avec une contrainte de latence de la forme $CostFunc_{lat} \leq Latency_{MAX}$ (à la place de $minimize(CostFunc_{lat})$) où $Latency_{MAX}$ représente la latence maximale acceptée.

Contrainte 4.5.16

$$\forall v \in OP :\ CostFunc_{lat} = max(..., v_{lat\_in} + v_{lat}, ...)$$
$$minimize(CostFunc_{lat})$$

Contrainte 4.5.17

$$CostFunc_{pwr} = \sum_{a \in A} a_{pwr} * a_{sel} + \sum_{e \in E} e_{mem\_ope} * memory_{pwr}$$
$$minimize(CostFunc_{pwr})$$

4.5.6 Exemple détaillé

Un exemple simple est illustré sur la figure 4.21. Cette figure représente l'ordonnancement d'un AG couvert par les motifs de la bibliothèque ainsi que le graphe réduit où chaque nœud est un motif et dans lequel apparait la ligne de retard (représentée par un hexagone).

Les appariements possibles avec les motifs de la bibliothèque sont donnés par la figure 4.22. La solution illustrée sur la figure 4.21 est représentée ici par les points noirs, les gris étant les appariements possibles.

Enfin, la figure 4.23 montre le résultat du déploiement de l'AG sur une instance du modèle d'architecture pipelinée. Les liens en pointillés ne sont pas utilisés dans la solution proposée. La mémoire utilisée pour réaliser la ligne de retard est identifiable par l'hexagone.

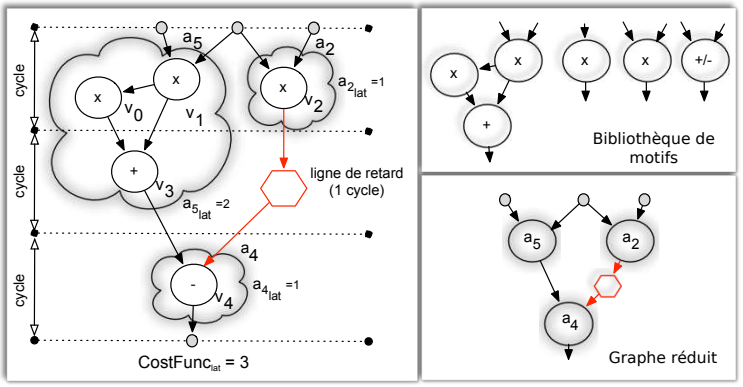


FIG. 4.21 – Exemple d'ordonnancement d'un AG après couverture par des motifs issus de la bibliothèque.

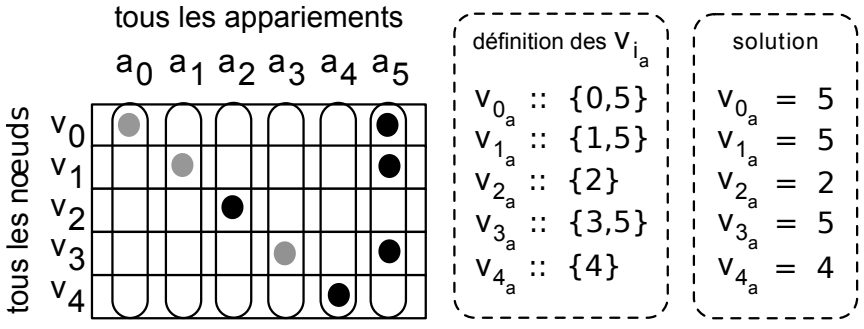


FIG. 4.22 – Appariements possibles (gris) et sélectionnés (noirs) des motifs de la bibliothèque de l'exemple.

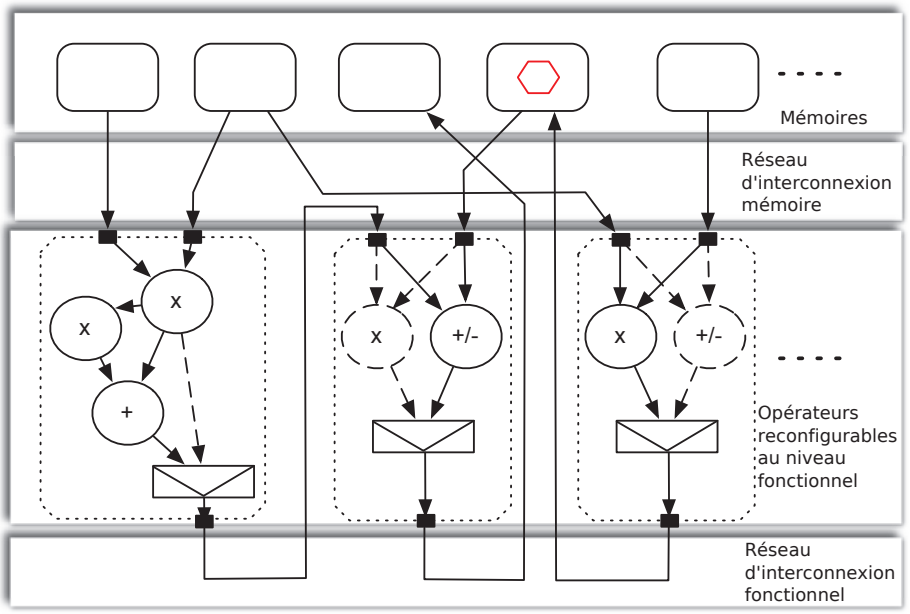


FIG. 4.23 – Déploiement de l'AG sur une instance de l'architecture pipelinée, les liens en pointillés ne sont pas utilisés.

4.5.7 Résultats

4.5.7.1 Conditions d'expérimentation

Dans le but de valider et d'évaluer le modèle de contraintes proposé pour le déploiement d'un AG sur un modèle d'architecture pipelinée, nous avons procédé à la compilation de plusieurs noyaux de calcul intensifs issus d'applications multimédia dont les caractéristiques sont données dans le tableau 4.8. Les applications choisies sont sensiblement les mêmes que celles utilisées pour évaluer le modèle de contraintes précédent (pour une architecture non pipelinée)[11]. La sélection des applications est un peu différente car nous avons voulu montrer que ce second modèle de contraintes, plus simple car la modélisation est effectuée au niveau d'une configuration, est capable de placer de manière optimale des AG de plus grande taille. Nous avons fait l'hypothèse que l'on dispose d'une architecture ayant un nombre de ressources suffisant pour un déploiement en une configuration. Ainsi, aucune limitation architecturale au niveau du nombre d'opérateurs ou de mémoires n'a été imposée pour ces expérimentations.

L'instance de l'architecture ROMA utilisée pour ces tests comporte un nombre illimité d'opérateurs homogènes et de mémoires locales. Les réseaux de communication de l'architecture sont ceux décrits précédemment, un réseau tout connecté entre les mémoires et les opérateurs et le réseau ROMA entre les opérateurs. Concernant les latences utilisées dans nos expérimentations nous avons considéré que la traversée du réseau fonctionnel a une latence nulle, que celle du réseau mémoire a une latence supérieure à zéro.

Concernant les opérateurs, nous avons utilisé deux bibliothèques pour la couverture des AG. La première bibliothèque est issue de l'opérateur ROMA, la seconde est issue

[11] A noter que les caractéristiques des AG communs aux deux évaluations peuvent varier du fait d'un filtrage plus agressif qui supprime les variables temporaires par exemple.

Application	AG	Nœuds	Arcs	Entrées/Sorties
Auto Regression Filter	1	56	58	28
gost	2	21	21	11
Write BMP Header	3	71	70	43
-//-	4	19	18	12
-//-	5	27	26	16
-//-	6	27	26	16
-//-	7	9	8	6
total	3+..7	153	148	93
sobel 7x7	8	14	13	8
(unrolled 2x2)	9	49	51	24
MESA Matrix Mul	10	52	60	24
(unrolled x2)	11	88	120	32
(unrolled x3)	12	124	180	40
(unrolled x4)	13	160	240	48
IIR biquad N sections	14	18	19	9
(unrolled x4)	15	65	72	30
Roma H filter (unrolled)	16	42	41	22

TAB. 4.8 – Caractéristiques des graphes d'application sélectionnés.

des motifs extraits avec le sytème UPaK (une brève présentation du système est donnée en 2.3.4.1, les motifs sont décrits dans l'article [117]). Ces deux bibliothèques ont été complétées par une base commune de motifs simples (opérations de base) dont la latence est de 1 cycle et la consommation de puissance est de 3 mW (cette puissance est une sur-approximation car elle se situe entre le calcul d'une valeur absolue d'une différence et une multiplication de l'opérateur ROMA en technologie ASIC 130 nm [62]). Toutes les latences et les consommations d'énergie pour la bibliothèque UPaK ont été approximées à partir de la synthèse de l'opérateur ROMA. A noter tout de même que pour la bibliothèque UPaK, le choix a été fait de modéliser l'addition et la soustraction comme des opérations de base (latence de 1 cycle et 3 mW de consommation de puissance) alors que dans la bibliothèque ROMA ces opérations sont implémentées dans l'opérateur (avec une latence de 3 cycles et une consommation propre de seulement 1 mW).

Ces données sont à considérer lors de la lecture des résultats mais n'ont pas d'impact direct sur la complexité de résolution du problème étant donné que nous considérons l'architecture comme homogène.

4.5.7.2 Analyse des résultats

Les tableaux 4.9 et 4.10 donnent les résultats obtenus en termes de latence et de puissance consommée pour la bibliothèque basée sur les motifs de l'opérateur ROMA et celle basée sur les motifs extraits par le système UPaK respectivement. Le nombre d'appariements, incluant ceux ne contenant qu'un seul nœud, est détaillé ainsi que le nombre d'opérateurs nécessaires au déploiement de l'AG en une seule configuration.

Lorsque la bibliothèque issue des motifs de l'opérateur ROMA est utilisée pour la couverture, les appariements ne contiennent qu'un nœud (sauf pour l'AG de l'application sobel (non déroulée) qui comporte une accumulation). Cela vient du fait que les applications utilisées dans nos expérimentations ne sont pas dans le champ d'application de l'opérateur ROMA qui est spécialisé pour les boucles non déroulées contenant une accumulation et/ou

une valeur absolue. Cet opérateur a déjà prouvé son efficacité dans le cadre d'algorithmes comme la DCT, SAD et DWT (*Discrete Wavelet Transform* ou transformée en ondelettes discrète) [63], l'objectif de nos expérimentations n'est pas de vérifier cela.

Étant donné que les appariements ne comportent qu'un nœud, la complexité du problème est faible. Nous obtenons donc facilement la preuve de l'optimalité des solutions dans 100% des cas, même pour l'AG 13 (MESA Matrix Mul déroulé avec un facteur 4) comportant 160 nœuds et 240 arcs avec un temps d'exécution de moins de 6 secondes. Nous observons par ailleurs que pour ce noyau de calcul totalement parallélisable, nous obtenons la même latence quelle que soit le facteur de déroulage. Le nombre d'opérateurs et la puissance consommée sont alors directement proportionnels au facteur de déroulage. Il est important de noter que le temps d'exécution nécessaire pour déterminer l'optimalité de la solution croit exponentiellement en fonction de la taille du problème.

Dans le cas où la bibliothèque basée sur les motifs extraits par UPaK est utilisée, le nombre d'appariements est plus élevé étant donné que les motifs de cette bibliothèque sont spécialement dédiés à l'accélération de ces applications. L'impact de la couverture des AG au niveau de la complexité du problème à résoudre est donc plus visible. Cela se traduit par l'obtention de la preuve de l'optimalité des solutions dans $93,75\%$ des cas lorsque le latence du pipeline est optimisée et dans $62,5\%$ des cas lorsque c'est la puissance consommée qui est optimisée. La différence entre les deux optimisations s'explique par le fait que la fonction de coût utilisée pour minimiser la puissance est plus complexe car elle fait intervenir la sélection du réseau de communication en plus de la sélection des appariements en fonction des opérateurs (voir la constrainte 4.5.17). Enfin, il reste deux cas où aucune solution n'est déterminée (cas particulier mettant à défaut notre implémentation ou phénomène de divergence ?).

Comme pour le modèle d'architecture non pipelinée, on observe que l'approche basée sur la CP ne permet pas d'obtenir des résultats dans tous les cas. En effet, si le problème à résoudre est trop complexe, même sur des graphes de petite taille, alors la solution ne peut être prouvée optimale ou même aucune solution n'est déterminée. Cela est aussi vrai lorsque le problème est simple mais l'AG est grand et totalement parallélisable. Nous pouvons en conclure qu'il est nécessaire de proposer, pour chaque modèle, des heuristiques au niveau des mécanismes de recherche de solutions et au niveau de l'ordre d'évaluation des variables. Cette approche a déjà été utilisée dans notre équipe de recherche et a permis d'améliorer le passage à l'échelle de certains modèles d'ordonnancement.

AG	Appariements (1 nœud)	Latence	Nbre Op	Tps d'exécution (ms)	Optimal	Puissance (mW)	Nbre Op	Tps d'exécution (ms)	Optimal
1	28 (28)	24	28	374	√	132	28	374	√
2	10 (10)	6	10	78	√	50	10	78	√
3	28 (28)	11	28	561	√	180	28	468	√
4	7 (7)	2	7	16	√	51	7	31	√
5	11 (11)	5	11	15	√	75	11	16	√
6	12 (11)	5	11	15	√	75	11	62	√
7	3 (3)	2	3	0	√	23	3	0	√
3+..+7		25	60	607	√	404	60	577	√
8	7 (6)	15	5	16	√	25	5	47	√
9	25 (25)	27	25	280	√	94	25	249	√
10	28 (28)	12	28	375	√	148	28	452	√
11	56 (56)	12	56	1094	√	296	56	1139	√
12	84 (84)	12	84	2497	√	444	84	2638	√
13	112 (112)	12	112	5445	√	592	112	5820	√
14	9 (9)	18	9	62	√	46	9	63	√
15	36 (36)	60	36	515	√	174	36	390	√
16	20 (20)	34	20	390	√	84	11	468	√

TAB. 4.9 – Résultats de latence et de consommation de puissance obtenus dans le cas d'une architecture pipelinée comportant des opérateurs de type ROMA.

AG	Appariements (1 nœud)	Latence	Nbre Op	Tps d'exécution (ms)	Optimal	Puissance (mW)	Nbre Op	Tps d'exécution (ms)	Optimal
1	60 (28)	14	12	624	√	x	x	TO	×
2	10 (10)	6	10	62	√	52	10	63	√
3	30 (28)	7	27	828	√	180	26	546	√
4	7 (7)	2	7	16	√	51	7	16	√
5	11 (11)	5	11	16	√	75	11	16	√
6	12 (11)	5	11	16	√	75	11	32	√
7	3 (3)	2	3	0	√	23	3	0	√
3+..+7		21	59	876	√	404	58	610	√
8	7 (6)	8	5	15	√	33	5	31	√
9	31 (25)	x	x	TO	×	123	19	608	√
10	76 (28)	6	20	1219	√	164	12	1156	×
11	152 (56)	6	40	2888	√	332	24	2404	×
12	228 (84)	6	60	7007	√	500	36	5884	×
13	304 (112)	6	80	17475	√	668	48	13200	×
14	17 (9)	9	6	109	√	50	6	187	√
15	68 (36)	27	24	1686	√	202	24	984	×
16	52 (20)	13	17	1223	√	113	20	281	√

TAB. 4.10 – Résultats de latence et de consommation de puissance obtenus dans le cas d'une architecture pipelinée comportant des opérateurs de type UPaK.

4.6 Génération de configurations

Cette section présente brièvement la génération de configurations propre au processeur ROMA aussi appelée *back-end* ROMA. L'objectif de cette étape du flot de compilation est de transformer le graphe d'application enrichi après le déploiement en un ensemble de fichiers binaires représentant les différentes configurations de la CGRA ROMA ainsi que le fichier de contrôle associé qui décrit le contrôle des configurations au cours du temps. Un simulateur de l'architecture au niveau transferts de registre a permis la simulation des déploiements avec une précision au cycle près.

4.6.1 Fichiers de configurations

Différents fichiers de configurations pour la CGRA ROMA doivent être générés. Chaque élément de l'architecture, opérateurs, réseaux d'interconnexion, générateurs d'adresses a un fichier de configuration associé. De plus, le contrôleur global a aussi un fichier associé qui comporte l'ensemble des instructions nécessaires au déclenchement des configurations dans le temps. La figure 4.24 montre l'ensemble des fichiers générés par le *back-end*. Sur cette figure apparaît aussi les fichiers de données spécifiant l'état des mémoires locales au lancement de l'exécution du simulateur.

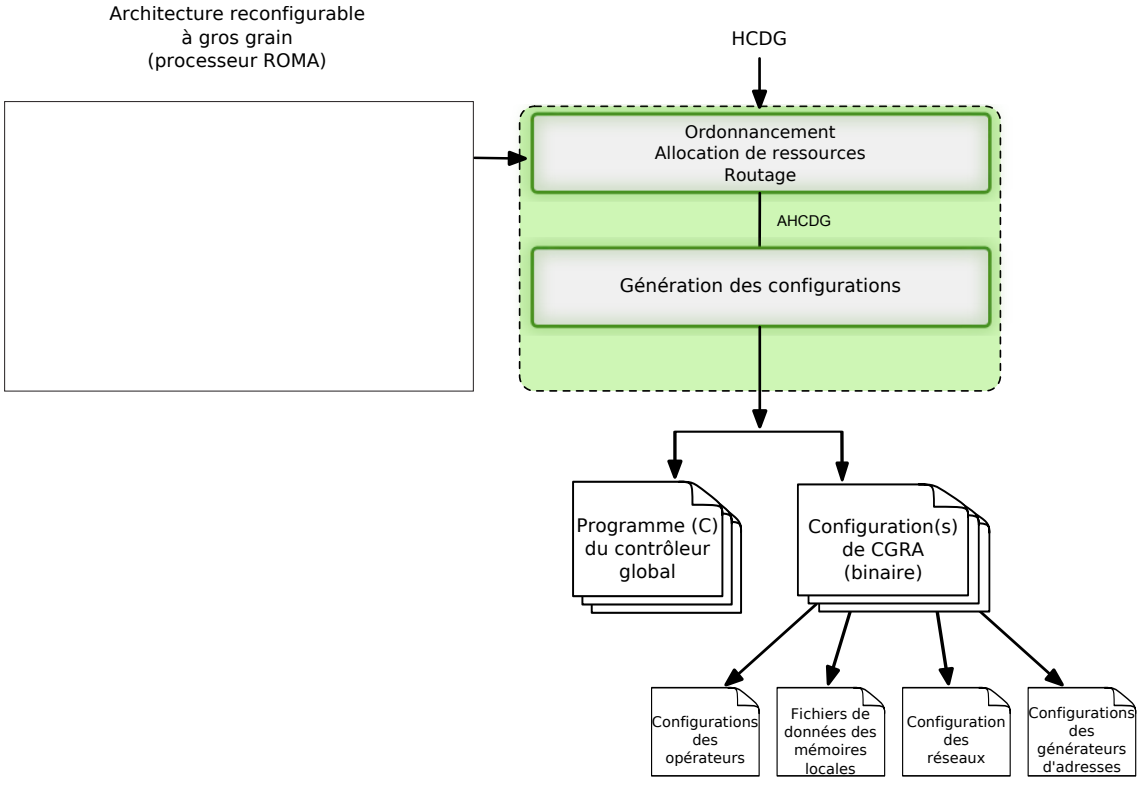


FIG. 4.24 – Ensemble des fichiers de configurations et de contrôle générés par le back-end ROMA.

4.6.2 Génération d'adresses

Comme nous l'avons précisé auparavant la génération d'adresses est gérée lors de la génération des configurations. Pour cela, les calculs des index des tableaux accédés en lec-

ture et en écriture sont extraits et analysés. De plus, une analyse des itérateurs de boucle permet d'identifier la réutilisation d'une donnée, ainsi elle n'est écrite en mémoire que pour la durée nécessaire au lieu de l'écrire à chaque itération. Les générateurs d'adresses de ROMA sont programmables et permettent de définir des motifs d'accès aux données. Une des grandes difficultés dans l'implémentation du *back-end* a été d'identifier ces motifs lors de l'analyse des calculs d'index et de programmer les générateurs d'adresses en conséquence.

4.6.3 Gestion des déclenchements des configurations

Le déploiement d'un AG par le modèle de contraintes prend en compte la partie flot de données mais ne gère pas le contrôle des configurations (elles sont considérées comme immédiates). Or, dans l'architecture réelle, il y a une latence de configuration, c.-à-d. un délai entre le déclenchement d'une configuration par le contrôleur global et le moment où cette configuration est active. Cette latence est différente en fonction du type d'élément reconfiguré. Le tableau 4.11 donne les latences de configuration en fonction de l'élément reconfiguré ainsi que les latences d'une lecture et d'une écriture en mémoire, du réseau mémoire et des opérateurs pour effectuer des opérations élémentaires.

Latence de configuration des opérateurs	3
Latence de configuration des réseaux de communication	3
Latence d'un lecture en mémoire	6
Latence d'un écriture en mémoire	5
Latence des opérateurs pour effectuer une opération élémentaire	3 (MUL = 4)
Latence du réseau mémoire	1

TAB. 4.11 – Latences réelles de ROMA.

Ainsi, mis à part le paramétrage de l'architecture en particulier des latences, certains ajouts ont été apportés au modèle de contraintes pour permettre de générer un code correct.

La principale modification du modèle concerne les accès à la mémoire car la latence d'une écriture (5 cycles) est différente de la latence de la lecture (6 cycles). Nous avons donc imposé un delai minimum entre deux accès consécutifs, le premier en écriture et le second en lecture. De plus, certains ajustements ont été apportés pour coller au mieux aux caractéristiques réelles précises de l'architecture. A noter que ces caractéristiques ont évolué durant l'implémentation du *back-end*, ce qui explique les différences entre l'exemple proposé ici (qui utilise les dernières caractéristiques de l'architecture) et le modèle plus théorique présenté précédemment. Mais cela montre la grande généricité de notre approche et la rapidité et la facilité de paramétrer le modèle d'architecture et de modifier sensiblement les contraintes du modèle.

4.6.4 Validation du flot complet

Dans le but de valider le flot de compilation basé sur la CP dans le cadre du projet ROMA, nous avons généré les configurations du pré-filtre de l'application PRiME et exécuté celles-ci sur le simulateur. Les données en entrée sont issues d'une séquence d'images et une vérification au bit près a permis de valider l'aspect fonctionnel du flot. Cette expérimentation a demandé beaucoup d'effort et n'a été possible que grâce à une

forte collaboration avec les différents partenaires du projet ROMA. Pour plus d'informations sur les résultats préliminaires de cette expérimentation, le lecteur peut se référer au délivrable du projet associé [50].

Conclusion et perspectives

Conclusion

Les applications modernes telles que celles rencontrées dans le domaine du multimédia et leurs implémentations dans des systèmes électroniques embarqués posent de nombreux problèmes auxquels s'attaquent les équipes de recherche universitaires, comme CAIRN à l'IRISA, et les industriels de ce secteur comme Technicolor.

Un des grands axes de recherche et développement de ces dernières années correspond à l'accélération des applications trop consommatrices de puissance de calcul et de transferts de données pour être exécutées sur des systèmes généralistes conventionnels. Parmi les solutions proposées, l'utilisation de processeurs spécialisés ou d'accélérateurs reconfigurables associés à des outils de conception et de compilation spécifiques et performants permet un compromis performance, flexibilité et consommation énergétique intéressant. Les problématiques de cette solution peuvent être résumées comme un problème d'adéquation entre l'application, l'architecture et l'outil de conception et de compilation.

Les problèmes d'optimisation liés en particulier à l'adéquation application $\leftrightarrows$ architecture sont nombreux et complexes. Jusqu'à maintenant, ils étaient traités séparément par des approches dédiées ou simultanément dans un contexte idéalisé. Cette thèse adresse deux problèmes d'optimisation propres à la conception et à la compilation pour architectures reconfigurables gros grain, type d'architecture particulièrement adapté à l'accélération d'applications multimédia.

- La fusion de motifs de calculs spécifiques à une application pour la synthèse d'unités reconfigurables dans le cadre de l'extension du jeu d'instructions d'un ASIP représente la première contribution de cette thèse.

- Le placement optimisé d'une application sur un modèle d'architecture reconfigurable à gros grain, dans le cadre du projet coopératif ROMA, représente la seconde contribution. Nous avons étendu cette contribution avec la modélisation par des contraintes d'une d'architecture pipelinée.

L'originalité des travaux présentés réside dans l'utilisation de la programmation par contraintes dans les deux contributions. Cette approche permet d'exprimer de manière simple des problèmes d'optimisations complexes et interdépendants, de les résoudre simultanément et de prouver dans la majorité des cas l'optimalité de la solution. De plus, il est possible de répondre à des problématiques réelles et concrètes, ce qui a été démontré par le développement d'un démonstrateur dans le cadre du projet ROMA permettant l'accélération d'une application multimédia issue de Technicolor. Nous avons aussi montré la grande

généricité de notre approche à travers son utilisation à différents niveaux d'abstraction, de l'exploration de l'espace de conception jusqu'à la génération de code pour un processeur réel.

L'approche basée sur l'utilisation de la CP que nous proposons comporte cependant ses limitations. La plus importante est la résolution de problèmes réels à grande échelle et cela même en mettant en place des contraintes spécialisées aux problèmes adressés. Nous sommes convaincus que cette limitation peut être repoussée, ce qui représente de vraies perspectives de recherches à court terme. En effet, même si nous avons essayé d'identifier la nature des limitations de la programmation par contraintes, l'étude des phénomènes de divergence lors du passage à l'échelle de nos modèles de contraintes ainsi que les méthodes permettant de les supprimer représentent un travail de recherche à part entière.

Perspectives

Concernant la fusion de chemins de données, la simplification du graphe de compatibilité devrait permettre de fusionner des chemins de données plus grand et/ou avec de plus fortes contraintes technologiques (longueur du chemin critique, consommation énergétique, etc.). L'appariement des nœuds d'entrée/sortie nous paraît être aussi pertinent dans le but d'augmenter l'impact de l'utilisation du concept de contournement (les opérateurs en entrée/sortie pourront alors être contournés).

A plus long terme, la fusion d'un ensemble de chemins de données en plusieurs cellules reconfigurables au niveau système, est une perspective importante. En effet, l'optimisation de la surface sous contrainte de performance (ou le contraire), ou encore, une approche multi-objectifs permettrait par exemple de générer les unités d'une architecture de type VLIW dans un contexte d'applications embarquées temps-réel.
L'ajout de contraintes sur la largeur des données, ou encore la synthèse de cellules reconfigurables pipelinées voir de type SWP, sont aussi des perspectives à évaluer.

Concernant le déploiement d'une application, le passage à l'échelle reste une forte limitation dans l'approche « solveur » de la CP car des AG comportant plus d'une centaine de nœuds posent des problèmes en termes de temps de résolution. Dans le cadre de cette thèse, une étude préliminaire a permi d'identifier des pistes pour résoudre ce problème.

La première solution envisagée est de diviser, lors de la résolution, les AG en *clusters* (c.-à-d. en grappes) tout en conservant des résultats localement optimaux. La division d'un AG doit être faite de telle manière que l'on puisse remplir l'architecture, c.-à-d. que le parallélisme de celle-ci soit exploité au maximum. Ainsi, la longueur, la largeur et la densité du sous-AG traité doivent être les principaux facteurs de division.

La seconde solution envisagée est d'effectuer le placement en utilisant une fenêtre glissante sur l'AG. Les même facteurs de clusterisation doivent alors être exploités.

La dernière solution est d'ordonner les variables (au sens CSP) lors de leur évaluation par le solveur. Cette solution est exploitée dans d'autres travaux de notre équipe de recherche.

De plus, nous avons remarqué que certaines propriétés d'un AG conduisaient à faire diverger la recherche de solutions par le solveur. Par exemple, dans le cas où plusieurs arcs sortant du même nœud sont reliés à des nœuds destinations dont la date de début d'exécution sont éloignés dans le temps. Une solution possible est de remplacer cette dépendance de données par l'utilisation de variables temporaires.

Enfin, certaines idées apparaissent comme incontournables au vu des travaux présentés dans cette thèse. La mise en place d'une approche multi-objectifs pour l'exploration de l'espace de conception en est un exemple tout à fait pertinent. Autre aspect intéressant, la prise en compte des différents niveaux de parallèlisme des applications à travers l'optimisation des transferts de données au niveau applicatif par exemple ou au contraire à plus bas niveau lors de la génération d'adresses. Ces sujets représentent un vrai intérêt pour la communauté scientifique mais aussi pour les industriels.

Glossaire

ACSS *Application Class-Specific System*, système spécifique à une classe d'application (spectre applicatif réduit). 16, 19, 24, 91, 94

ADSS *Application Domain-Specific System*, système spécifique à un domaine d'application (spectre applicatif large). 16, 19, 21, 27, 91

AG *Application Graph*, graphe d'application. 59, 61–63, 70, 72, 77, 79, 81, 100, 101, 103, 105, 108–110, 116, 118, 119, 121–126, 128, 129, 131–133, 137, 140, 141

ASIC *Application Specific Integrated Circuit*, circuit spécialisé pour une application. 4, 16, 36, 66, 90, 132

ASIP *Application-Specific Instruction-set Processor*, processeur dont le jeu d'instruction est spécialisé pour une ou plusieurs applications. 17, 26, 52, 54, 59, 61, 84, 90, 139

B&B *Branch and Bound*, algorithme par séparation et évaluation. 50, 52

CDFG *Control Data Flow Graph*, graphe de flot de données et de contrôle. 53, 54

CGRA *Coarse Grain Reconfigurable Architecture*, architecture reconfigurable à gros grain. 4–6, 8–11, 13, 15–21, 24, 27, 37, 39, 43, 44, 87, 88, 90, 91, 95, 96, 99, 104, 136

CP *Constraint Programming*, programmation par contraintes. 10, 11, 44–46, 49, 51–54, 58, 85, 87, 88, 90, 95, 98, 101, 133, 137, 140

CSP *Constraint Satisfaction Problem*, problème de satisfaction de contraintes. 46, 50, 105, 140

DCT *Discret Cosine Transform*, transformée en cosinus discrète. 32, 36, 114, 133

DFG *Data Flow Graph*, graphe de flot de données. 53, 59, 100

DFS *Depth First Search*, parcours en profondeur. 50, 52

DPM *Data Path Merging*, fusion de chemins de données. 61, 62

DSE *Design Space Exploration*, exploration de l'espace de conception. 96–98

DSP *Digital Signal Processor*, processeur dédié au traitement du signal. 15, 33, 36, 82, 90, 94

FIR *Finite Impulse Response*, filtre à réponse impulsionnelle finie. 32

FPGA *Field-Programmable Gate Array*, circuit logique programmable. 4, 13–15, 23, 29, 39, 61, 82, 90

GPP *General Purpose Processor*, processeur généraliste. 4, 5, 8, 19

HCDG *Hierarchical Conditional Dependency Graph*, Graphe Hiérarchisé aux Dépendances Conditionnées. 51, 53, 54, 59, 88

ILP *Integer Linear Programming*, programmation linéaire en nombres entiers. 87, 116

MAC *Multiply and ACumulate*, multiplication avec accumulation. 32, 33, 40, 93

MIP *Mixed Integer Programming*, programmation linéaire mixte c.-à-d. avec variables enitères et continues. 87, 116

P&R *Place and Route*, placement et routage. 14, 39

RPU *Reconfigurable Processing Unit*, unité de traitement reconfigurable. 18–21, 24, 26, 27

SAD *Sum of Absolute Differences*, somme des différences absolues. 93, 133

SIMD *Single Instruction on Multiple Data*, modèle d'exécution dans lequel une instruction est exécuter sur un ensemble de données. 40, 52

SoC *System on Chip*, système sur puce. 2–4, 16, 21, 23, 32

SWP *Sub-Word Parallelism*, parralélisme dit de sous-mot. Se dit d'un opérateur capable d'effectuer des opérations en parallèle sur des données de largeur réduite (par exemple, deux opérations en parallèle sur des données de 16 bits au lieu d'une opération sur 32 bits). 86, 93, 140

UAL Unité Arithmétique et Logique. 29, 32, 33, 35, 39

VLIW *Very Long Instruction Word*, processeur qui adressent plusieurs unités fonctionnelles en parallèle dans la même instruction, exploitant ainsi le parallélisme d'instruction. 15, 33, 37, 39, 40, 57, 86, 87, 140

Publications

Journaux

C. Wolinski, K. Kuchcinski et E. Raffin : Automatic Design of Application-Specific Reconfigurable Processor Extensions with UPaK Synthesis Kernel. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15 :1-36, 2009.

E. Raffin, C. Wolinski, F. Charot, K. Kuchcinski, S. Guyetant, S. Chevobbe, A. Floch et E. Casseau : Scheduling, Binding and Routing System for a Run-Time Reconfigurable Operator Based Multimedia Architecture. *Submitted to International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2011.

Conférences

D. Menard, H.-N. Nguyen, F. Charot, S. Guyetant, J. Guillot, E. Raffin et E. Casseau : Exploiting reconfigurable SWP operators for multimedia applications. *In Proceedings of the 36th International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Prague, République tchèque, 2011.

E. Raffin, C. Wolinski, F. Charot, K. Kuchcinski, S. Guyetant, S. Chevobbe et E. Casseau : Scheduling, Binding and Routing System for a Run-Time Reconfigurable Operator Based Multimedia Architecture. *In Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Édimbourg, Écosse, 2010.
Best Paper Award.

C. Wolinski, K. Kuchcinski, E. Raffin et F. Charot : Architecture-Driven Synthesis of Reconfigurable Cells. *In Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, Patras, Grèce, 2010.

C. Wolinski, K. Kuchcinski, K. Martin, E. Raffin et F. Charot : How constrains programming can help you in the generation of optimized application specific reconfigurable processor extensions. *In Proceedings of the International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA)*, Las Vegas, USA, 2009 (papier invité).

Workshops

C. Wolinski, K. Kuchcinski, K. Martin, A. Floch, E. Raffin et F. Charot : Graph Constraints in Embedded System Design. *In Workshop on Combinatorial Optimization for Embedded System Design (COESD)*, Bologne, Italie, 2010.

C. Wolinski, K. Kuchcinski, K. Martin, E. Raffin et F. Charot : Graph Constraints in Embedded System Design. *In the 8th Workshop of the Network for Sweden-based researchers and practitioners of Constraint programming (SweConsNet)*, Bologne, Italie, 2009 (présentation invitée).

Bibliographie

[1] GeCoS. generic compiler suite - http ://gecos.gforge.inira.fr/. 54, 59, 88

[2] JaCoP, solveur de programmation par contraintes, http ://jacop.osolpro.com/. 45, 50, 51, 53

[3] Mescal project, http ://embedded.eecs.berkeley.edu/mescal/. 26, 61

[4] PIPS : Automatic parallelizer and code transformation framework, http ://pips4u.org/. 52

[5] Silicon hive, http ://www.siliconhive.com/. 40, 41, 42

[6] A. Abnous et J. Rabaey : Ultra-low-power domain-specific multimedia processors. *In IX Workshop on VLSI Signal Processing*, p. 461 –470, 1996. 30

[7] G. Ansaloni, P. Bonzini et L. Pozzi : EGRA : A coarse grained reconfigurable architectural template. *IEEE Transactions on VLSI Systems*, PP:1–13, 2010. 19, 20

[8] K. Apostolos : *Outils pour la Validation Temporelle et l'Optimisation de Programmes Synchrones*. Thèse de doctorat, Université de Rennes 1, Rennes, FRANCE, 1998. 51, 59, 88

[9] R. Bartak : Theory and practice of constraint propagation. *In Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control*, 2001. 50

[10] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach et M. Weinhardt : PACT XPP—a self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26:167–184, 2003. 29, 32

[11] J. Becker, T. Piontek et M. Glesner : DReAM : A dynamically reconfigurable architecture for future mobile communications applications. *In Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL)*, p. 312–321, 2000. 29

[12] R. Beckmann, U. Bieker et I. Markhof : *Constraints in Computational Logics*, vol. 845 de *Lecture Notes in Computer Science*, chap. Application of constraint logic programming for VLSI CAD tools, p. 183–200. SpringerLink, 1994. 51

[13] M. Bekooij : *Constraint Driven Operation Assignment for Retargetable VLIW Compilers*. Thèse de doctorat, Eindhoven University of Technology, 2004. 42

[14] F. Bouwens, M. Berekovic, A. Kanstein et G. Gaydadjiev : Architectural exploration of the ADRES coarse-grained reconfigurable array. *In Proceedings of the 3rd international conference on Reconfigurable computing (ARC'07)*, p. 1–13, 2007. 39

[15] F. Bouwens, M. Berekovic, B. D. Sutter et G. Gaydadjiev : Architecture enhancements for the ADRES coarse-grained reconfigurable array. *In Proceedings of HiPEAC*, 2008. 39

[16] J. Brenner, J. van der Veen, S. Fekete, J. Oliveira Filho et W. Rosenstiel : Optimal simultaneous scheduling, binding and routing for processor-like reconfigurable architectures. *In International Conference on Field Programmable Logic and Applications (FPL '06)*, 2006. 87

[17] P. Brisk, A. Kaplan et M. Sarrafzadeh : Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. *In Proceedings of the 41st Design Automation Conference (DAC)*, p. 395–400, 2004. 26

[18] C. Brunelli, F. Garzia, D. Rossi et J. Nurmi : A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calculations. *EUROMICRO Journal of Systems Architecture*, 56(1):38–47, 2010. 29

[19] G. F. Burns, M. Jacobs, M. Lindwer et B. Vandewiele : Exploiting parallelism, while managing complexity using silicon hive programming tools. White paper, 2004 (or later). 42, 43

[20] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit et D. E. Thomas : Managing pipeline-reconfigurable FPGAs. *In Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 1998. 29

[21] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo et R. Guerrieri : A VLIW processor with reconfigurable instruction set for embedded applications. *In Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, p. 250 – 491 vol.1, 2003. 29

[22] Y. Caseau, F.-X. Josset et F. Laburthe : CLAIRE : combining sets, search and rules to better express algorithms. *Theory Practice of Logic Programming*, 2:769–805, 2002. 52

[23] E. Casseau, F. Charot, A. Floch, S. Khan, D. Ménard, O. Sentieys, C. Wolinki, S. Guyetant, S. Chevobbe, A. Tisserand, H. Heijnen, J.-P. Le Glanic et E. Raffin : Roma project intermediate progress report. Rapport technique, Agence Nationale de la Recherche (ANR) - IRISA, CEA List, LIRMM, Thomson R&DF, 2008. publication sur le site web http ://roma.irisa.fr. 11, 90

[24] T. Cervero, S. Lopez et R. Sarmiento : Dynamically reconfigurable architectures for multimedia applications. *In Proceedings of the XXIV Conference on Design of Circuits and Integrated Systems (DCIS)*, 2009. 27, 29, 30

[25] S. R. Chalamalasetti, S. Purohit, M. Margala et W. Vanderbauwhede : MORA - an architecture and programming model for a resource efficient coarse grained reconfigurable processor. *In Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, vol. 0, p. 389–396, 2009. 29

[26] S.-Y. Chien, Y.-W. Huang, C.-Y. Chen, H. H. Chen et L.-G. Chen : Hardware architecture design of video compression for multimedia communication systems. *IEEE Communications Magazine*, 43:122–131, 2005. 2, 3

[27] N. T. Clark, H. Zhong et S. A. Mahlke : Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, 54:1258–1270, 2005. 26

[28] K. Compton et S. Hauck : Reconfigurable computing : a survey of systems and software. *ACM Computing Surveys*, 34:171–210, 2002. 16

[29] K. L. Compton : *Architecture Generation of Customized Reconfigurable Hardware*. Thèse de doctorat, Northwestern University, 2003. 26

[30] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak et J. M. Rabaey : Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:877–888, 1996. 66, 85

[31] E. v. Dalen, S. G. Pestana et A. v. Wel : An integrated, low-power processor for image signal processing. *In Proceedings of the Eighth IEEE International Symposium on Multimedia (ISM)*, p. 501–508, 2006. 41

[32] R. David : *Architecture reconfigurable dynamiquement pour applications mobiles*. Thèse de doctorat, Université de Rennes 1, Rennes, FRANCE, 2003. 29, 31, 87

[33] R. David, D. Chillet, S. Pillement et O. Sentieys : DART : a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. *In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, p. 156 –163, 2002. 32

[34] R. David, D. D. Lavenier et S. Pillement : Du microprocesseur au circuit FPGA : Une analyse sous l'angle de la reconfiguration. *Technique et Science Informatiques*, 4:395–422, 2005. 16

[35] S. de Givry et L. Jeannin : A unified framework for partial and hybrid search methods in constraint programming. *Computers and Operations Research*, 33:2805–2833, 2006. 52

[36] C. C. de Souza, A. M. Lima, G. Araujo et N. B. Moreano : The datapath merging problem in reconfigurable systems : Complexity, dual bounds and heuristic evaluation. *Journal of Experimental Algorithmics (JEA)*, 10:2.2, 2005. 26, 62

[37] C. Ebeling, D. C. Cronquist et P. Franklin : RaPiD - reconfigurable pipelined datapath. *In Proceedings of the 6th International Workshop on Field-Programmable Logic (FPL)*, p. 126–135, 1996. 29

[38] C. Ekelin : *An Optimization Framework for Scheduling of Embedded Real-Time Systems*. Thèse de doctorat, School of Computer Science and Engineering Chalmers University of Technology, 2004. 51

[39] G. Estrin : Reconfigurable computer origins : the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Annals of the History of Computing*, 24:3–9, 2002. 13

[40] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto et A. Tumeo : Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:911–924, June 2010. 88

[41] A. Floch : Compilation pour architecture reconfigurable à grain épais. Mémoire de master recherche, Institut de Formation Supérieure en Informatique et Communication (IFSIC), 2007. 14

[42] A. Floch, C. Wolinski et K. Kuchcinski : Combined scheduling and instruction selection for processors with reconfigurable cell fabric. *In Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, p. 167 –174, 2010. 57, 123

[43] G. L. Fol : *Architecture parallèle pour la compression vidéo : Contribution à la conception d'un module VLSI programmable et à l'étude d'outils de compilation reciblables*. Thèse de doctorat, Université de Rennes 1, Rennes, FRANCE, 1997. 44

[44] E. C. Freuder : In pursuit of the holy grail. *Constraints*, 2:57–61, 1997. 45

[45] W. Geurts : *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, 1997. 61

[46] M. Gries et K. Keutzer : *Building ASIPs : The Mescal Methodology*. Springer, 2005. 61

[47] E. Grâce : *Hiérarchie mémoire reconfigurable faible consommation pour systèmes enfouis*. Thèse de doctorat, Université de Rennes 1, Rennes, FRANCE, 2010. 95

[48] C. Guettier : *Optimisation globale et placement d'applications de traitement du signal sur architectures parallèles utilisant le programmation par contraintes*. Thèse de doctorat, École Nationale Supérieure des Mines de Paris, 1997. 52

[49] Y. Guo : *Mapping applications to a coarse-grained reconfigurable architecture*. Thèse de doctorat, University of Twente, 2006. 32, 34, 35, 87

[50] S. Guyetant, E. Raffin, E. Jolly et F. Charot : Projet roma : Rapport sur la plateforme de démonstration et ses résultats - point de vue industriel -. Rapport technique, Agence Nationale de la Recherche (ANR) - CEA-LIST, Thomson R&D France, IRISA, 2010. Délivrable ANR. 138

[51] T. R. Halfhill : Silicon hive breaks out, 2003. 41, 42

[52] R. Hartenstein : The microprocessor is no longer general purpose : why future reconfigurable platforms will win. *In Proceedings of the Second Annual IEEE International Conference on Innovative Systems in Silicon*, p. 2 –12, 1997. 14

[53] R. Hartenstein : A decade of reconfigurable computing : a visionary retrospective. *In Proceedings of the conference on Design, automation and test in Europe (DATE '01)*, p. 642–649, 2001. 4, 16

[54] J. Hauser et J. Wawrzynek : Garp : a MIPS processor with a reconfigurable coprocessor. *In Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, p. 12 –21, 1997. 30

[55] S. D. Haynes, H. G. Epsom, R. J. Cooper et P. L. McAlpine : UltraSONIC : A reconfigurable architecture for video image processing. *In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, p. 482–491, 2002. 29

[56] Z. Huang et S. Malik : Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. *In Proceedings of the Design, Automation and Test in Europe (DATE'01)*, p. 735–740, 2001. 61

[57] Z. Huang, S. Malik, N. Moreano et G. Araujo : The design of dynamically reconfigurable datapath coprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 3:361–384, 2004. 26, 62

[58] I. Hurbain, C. Ancourt, F. Irigoin, M. Barreteau, N. Museux et F. Pasquier : A case study of design space exploration for embedded multimedia applications on SoCs. *In Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP)*, vol. 0, p. 133–139, 2006. 52

[59] J. Jonsson et K. G. Shin : A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system. *Parallel Processing, International Conference on*, 0:158, 1997. 87

[60] L. Jówiak, N. Nedjah et M. Figueroa : Modern development methods and tools for embedded reconfigurable systems : A survey. *Integration, the VLSI Journal*, 43:1–33, 2010. 10, 16

[61] S. Kelem, B. Box, S. Wasson, R. Plunkett, J. Hassoun et C. Phillips : An elemental computing architecture for SD radio. *In Proceeding of the SDR 07 Technical Conference and Product Exposition*, 2007. 29

[62] S. Khan, E. Casseau et D. Menard : Reconfigurable SWP operator for multimedia processing. *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 0:199–202, 2009. 132

[63] S. Khan, E. Casseau et D. Menard : High speed reconfigurable SWP operator for multimedia processing using redundant data representation. *International Journal of Information Sciences and Computer Engineering*, 1:45–52, 2010. 93, 133

[64] C. Kim, A. Rassau, S. Lachowicz, S. Nooshabadi et K. Eshraghian : 3D-SoftChip : A novel 3D vertically integrated adaptive computing system. *In VLSI-SoC : From Systems To Silicon*, vol. 240, p. 71–86. Springer Boston, 2007. 29

[65] K. Kuchcinski : An approach to high-level synthesis using constraint logic programming. *In Proceedings of the 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Sweden*, 1998. 51

[66] K. Kuchcinski : Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8:355–383, 2003. 10

[67] K. Kuchcinski et C. Wolinski : Global approach to assignment and scheduling of complex behaviors based on HCDG and constraint programming. *Journal of Systems Architecture - Special issue : Synthesis and verification*, 49:489–503, December 2003. 51

[68] C. Lee, M. Potkonjak et W. H. Mangione-smith : MediaBench : A tool for evaluating and synthesizing multimedia and communications systems. *In Proceedings of the International Symposium on Microarchitecture*, p. 330–335, 1997. 82

[69] C. Liang et X. Huang : SmartCell : An energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, p. 15 pages, 2009. 5, 29

[70] A. d. C. Lucas, S. Heithecker et R. Ernst : FlexWAFE - a high-end real-time stream processing library for FPGAs. *In Proceedings of the 44th annual Design Automation Conference (DAC)*, p. 916–921, 2007. 29

[71] D. Marpe, T. Wiegand et G. Sullivan : The H.264/MPEG4 advanced video coding standard and its applications. *IEEE Communications Magazine*, 44:134 – 143, 2006. 1

[72] K. Martin : *Génération automatique d'extensions de jeux d'instructions de processeurs*. Thèse de doctorat, Université de Rennes 1, 2010. 8, 54, 56, 88, 124

[73] K. Martin, C. Wolinki, K. Kuchcinski, A. Floch et F. Charot : DURASE : Generic environment for design and utilization of reconfigurable application-specific processors extensions. *In University booth at DATE'09*, 2009. 8, 26, 59

[74] B. Mei, F.-J. Veredas et B. Masschelein : Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. *In Proceedings of the International Conference on Field Programmable Logic and Applications*, p. 622 – 625, 2005. 39

[75] B. Mei, S. Vernalde, D. Verkest et R. Lauwereins : Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture : A case study. *In Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, 2004. 37, 38

[76] B. Mei, S. Vernalde, D. Verkest, H. D. Man et R. Lauwereins : ADRES : An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *Lecture Notes in Computer Science*, 2778/2003:61–70, 2003. 29, 37, 98

[77] D. Menard, E. Casseau, S. Khan, O. Sentieys, S. Chevobbe, S. Guyetant et R. David : Reconfigurable Operator Based Multimedia Embedded Processor. *In Reconfigurable Computing : Architectures, Tools and Applications*, vol. 5453, p. 39–49, 2009. 91

[78] E. Mirsky et A. DeHon : MATRIX : A reconfigurable computing architecture with configurable instruction distribution and deployable resources. *In IEEE Symposium on FPGAs for Custom Computing Machines*, p. 157–166, 1996. 29

[79] N. Moreano, G. Araujo et C. C. de Souza : CDFG merging for reconfigurable architectures. Rapport technique IC-03-18, Institute of Computing, University of Campinas, October 2003. 61

[80] N. Moreano, G. Araujo, Z. Huang et S. Malik : Datapath merging and interconnection sharing for reconfigurable architectures. *In Proceedings of the 15th International Symposium on System Synthesis*, p. 38–43, 2002. 62

[81] N. Moreano, E. Borin, C. de Souza et G. Araujo : Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):969–980, 2005. 26, 27, 61, 62, 66, 79, 81, 85

[82] N. Museau : *Aide au Placement d'Applications de Traitement du Signal sur Machines Parallèles Multi-SPMD*. Thèse de doctorat, École Nationale Supérieure des Mines de Paris, 2001. 52

[83] S. Niskanen et P. Östergärd : Cliquers user's guide, version 1.0. 65

[84] P. R. J. Östergård : A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8:424–436, 2001. 65

[85] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer et T. Wedi : Video coding with H.264/AVC : tools, performance, and complexity. *IEEE Circuits and Systems Magazine*, 4(1):7–28, 2004. 1

[86] K. V. Palem, L. N. Chakrapani, Z. M. Kedem, A. Lingamneni et K. K. Muntimadugu : Sustaining moore's law in embedded computing through probabilistic and approximate design : retrospects and prospects. *In Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (CASES)*, p. 1–10, 2009. 2

[87] S. Pillement, O. Sentieys et R. David : DART : A Functional-Level Reconfigurable Architecture for High Energy Efficiency. *EURASIP Journal on Embedded Systems (JES)*, 1:1–13, 2008. 29, 31, 32, 33

[88] C. A. Pinto, A. Beric, H. Peters, E. van Dalen et R. Sethuraman : HiveFlex VSP2000 series : A scalable & flexible video signal processor for coding/decoding and pre-/post-processing applications. White paper, July 2007. 41

[89] Y. Qu, J.-P. Soininen et J. Nurmi : Static scheduling techniques for dependent tasks on dynamically reconfigurable devices. *Journal of Systems Architecture*, 53:861 – 876, 2007. 88

[90] B. Radunovic et V. M. Milutinovic : A survey of reconfigurable computing architectures. *In Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm (FPL '98)*, p. 376–385, 1998. 15

[91] E. Raffin, C. Wolinki, F. Charot, K. Kuchcinski, S. Guyetant, S. Chevobbe et E. Casseau : Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture. *In Proceedings of IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)*, October 2010. 88, 91, 113, 114

[92] K. Rath, S. Tangirala, P. Friel, P. Balsara, J. Flores et J. Wadley : Reconfigurable array media processor (RAMP). *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 287, 2000. 29

[93] B. R. Rau : Iterative modulo scheduling : an algorithm for software pipelining loops. *In Proceedings of the 27th annual international symposium on Microarchitecture (MICRO 27)*, p. 63–74, 1994. 37

[94] J.-C. Régin : Solving the maximum clique problem with constraint programming. *In Proceedings of International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, 2003. 71

[95] J.-C. Régin : Modélisation de contraintes globales en programmation par contraintes, novembre 2004. Habilitation à Diriger des Recherches, Université de Nice-Sophia Antipolis. 45, 49, 50

[96] M. Sadiq et S. Khan : Optimal mapping of DSP algorithms on commercially available off-the-shelf (COTS) VLIW DSPs. *Consumer Electronics, IEEE Transactions on*, 53:1061 –1067, 2007. 87

[97] T. Sato, H. Watanabe et K. Shiba : Implementation of dynamically reconfigurable processor DAPDNA-2. *In Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-TSA-DAT)*, p. 323 – 324, 2005. 29

[98] M. Schwehm et T. Walter : Mapping and scheduling by genetic algorithms. *In* B. Buchberger et J. Volkert, éds : *Parallel Processing : CONPAR 94 Ů VAPP VI*, vol. 854 de *Lecture Notes in Computer Science*, p. 832–841. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-58430-7.72. 88

[99] Y. Shengfa, C. Zhenping et Z. Zhaowen : Instruction-level optimization of H.264 encoder using SIMD instructions. *In International Conference on Communications, Circuits and Systems Proceedings*, vol. 1, p. 126–129, 2006. 2

[100] N. Shirazi, W. Luk et P. Y. K. Cheung : Automating production of run-time reconfigurable designs. *In FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, p. 147–156, 1998. 61, 62

[101] S. Shukla, N. W. Bergmann et J. Becker : QUKU : A FPGA based flexible coarse grain architecture design paradigm using process networks. *In 21st International Parallel and Distributed Processing SymposiumIPDPS)*, p. 1–7, 2007. 29

[102] H. Singh, M. Lee, G. Lu, F. Kurdahi et N. Bagherzadeh : MorphoSys : A reconfigurable architecture for multimedia applications. *In Proceedings of the XI Brazilian Symposium on Integrated Circuit Design*, vol. 0, p. 134, 1998. 29

[103] G. J. Smit, M. A. Rosien, Y. Guo et P. M. Heysters : Overview of the tool-flow for the montium processor tile. *In International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, p. 45–51, 2004. 33, 35

[104] G. J. M. Smit, A. B. J. Kokkeler, P. T. Wolkotte, P. K. F. Hölzenspies, M. D. van de Burgwal et P. M. Heysters : The chameleon architecture for streaming DSP applications. *EURASIP Journal on Embedded Systems*, 2007(1):11–11, 2007. 29

[105] L. Smit, G. Rauwerda, A. Molclerink, P. Wolkotte et G. Smit : Implementation of a 2-D 8x8 IDCT on the reconfigurable montium core. *In International Conference on Field Programmable Logic and Applications (FPL).*, p. 562 –566, 2007. 36

[106] P. Sunna : AVC/H.264 - Un système de codage vidéo évolué pour la HD et SD. *UER - REVUE TECHNIQUE*, 2005. 1

[107] J. Teich, T. Blickle et L. Thiele : An evolutionary approach to system-level synthesis. *In CODES '97 : Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, p. 167, 1997. 88

[108] G. Theodoridis, D. Soudris et S. Vassiliadis : *Fine- and Coarse-Grain Reconfigurable Computing*, chap. A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools, p. 89–149. Springer-Verlag New York, 2007. 16, 24, 25, 27, 28

[109] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer et W. Luk : Reconfigurable computing : architectures and design methods. *In IEE Proceedings Computers and Digital Techniques*, p. 193–207, 2005. 16, 17

[110] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi et L. Jing : High-level synthesis challenges and solutions for a dynamically reconfigurable processor. *In Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06)*, p. 702–708, 2006. 29

[111] Z. ul Abdin et B. Svensson : Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocessors and Microsystems*, 33:161–178, 2009. 4, 16

[112] E. van Dalen et S. G. Pestana : HiveFlex ISP2100 : An integrated, low-power processor for image signal processing. White paper, February 2007. 41

[113] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov et E. M. Panainte : The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53:1363–1375, 2004. 29

[114] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe et A. Agarwal : Baring it all to software : Raw machines. *IEEE Computer*, 30:86–93, 1997. 29

[115] C. Wolinski et K. Kuchcinski : Identification of application specific instructions based on sub-graph isomorphism constraints. *In IEEE 18th Intl. Conference on Application-specific Systems, Architectures and Processors, Montréal, Canada*, 2007. 125

[116] C. Wolinski, K. Kuchcinski, K. Martin, A. Floch, E. Raffin et F. Charot : Graph constraints in embedded system design. *In Workshop on Combinatorial Optimization for Embedded System Design (COESD)*, 2010. 54

[117] C. Wolinski, K. Kuchcinski et E. Raffin : Automatic design of application-specific reconfigurable processor extensions with UPaK synthesis kernel. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15:1–36, 2009. 54, 59, 132

[118] C. Wolinski, K. Kuchcinski, E. Raffin et F. Charot : Architecture-driven synthesis of reconfigurable cells. *In Proceedings of the 12th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD '09)*, 2009. 80

[119] H. Zhang, M. Wan, V. George et J. Rabaey : Interconnect architecture exploration for low-energy reconfigurable single-chip DSPs. *In Proceedings of the IEEE Computer Society Workshop on VLSI'99 (WVLSI '99)*, p. 2, 1999. 21, 22

Résumé

Les systèmes embarqués sont des dispositifs électroniques et informatiques autonomes, dédiés à une tâche bien précise. Leur utilisation s'est désormais démocratisée à de nombreux domaines d'applications et en particulier au multimédia. Ce type d'application est caractérisé par un besoin important en puissance de calcul et en échange de données. Les architectures matérielles au cœur de ces systèmes sont généralement dotées d'accélérateurs chargés de l'exécution des noyaux de calcul intensif.

Les architectures reconfigurables à gros grain (CGRA) sont particulièrement adaptées à l'accélération d'applications multimédia car elles répondent au mieux aux contraintes de performance, d'efficacité énergétique, de flexibilité et de coût de conception. En effet, ce type d'architecture est un compromis entre les processeurs à usage général, les architectures dédiées et celles reconfigurables à grain fin.

Cette thèse traite de certains aspects liés aux problématiques de conception et de compilation d'applications pour CGRA. Nos travaux s'inscrivent dans une démarche d'adéquation applications multimédia / CGRA / conception et compilation basées sur la programmation par contraintes (CP). Notre méthodologie nous a permis, grâce à la CP, de modéliser et de résoudre un ensemble de problèmes combinatoires complexes. Le premier modèle présenté a trait à la fusion d'unités fonctionnelles reconfigurables sous contraintes architecturales et technologiques. Les deux autres modèles abordent les problèmes de : placement, ordonnancement et routage des données pour le déploiement d'une application sur CGRA. Notre approche permet, dans la majorité des cas, de prouver l'optimalité de la solution obtenue.

Mots-clés : systèmes embarqués, architectures reconfigurables à gros grain, applications multimédia, méthodologie de conception, compilation, programmation par contraintes

Abstract

Embedded systems are stand alone electronic and computer devices dedicated to handle a particular task. They cover a wide range of application areas, particularly in the multimedia. This application area is characterized by tremendous processing power and data exchange requirements. Hardware architectures within these systems are generally fitted with accelerators dedicated to the execution of intensive computation kernels.

Coarse grain reconfigurable architectures (CGRA) are particularly well suited to speed up multimedia applications because they fit to design constraints : performance, power efficiency, flexibility and design cost constraints. Indeed, this type of architecture is a good trade-off between general purpose processors, application specific integrated circuit and fine grain reconfigurable architectures.

This thesis deals with certain design and compilation aspects for CGRA. Our work falls within a framework of adequation between multimedia applications / CGRA / constraint programming (CP)-based design and compilation. Thanks to CP, our methodology has allowed us to model and solve a set of complex combinatorial problems. The first model presented here is related to data path merging under architectural and technological constraints. The two other models address the application scheduling, binding and routing problems on CGRA. In most cases, our approach provides optimal results.

Mots-clés : embedded systems, coarse grain reconfigurable architectures, multimédia applications, design methodology, compilation, constraint programming