Contribution à la prise en compte des contraintes des applications TDSI dans la synthèse de haut niveau

Bertrand Le Gal

Thèse

pour obtenir le titre de

Docteur de l'Université de Bretagne Sud

Mention : Sciences et Sciences de l'Ingénieur

présentée et soutenue publiquement par

Bertrand Le Gal

le 8 décembre 2005

Contribution à la Prise en Compte des Contraintes des Applications TDSI dans la Synthèse de Haut Niveau

---

Directeur de thèse : Eric MARTIN

Jury composé de :

E. CASSEAU	Maitre de Conférences HDR à l'Université de Bretagne Sud	Examinateur
A.M. FOUILLART	Ingénieur de Recherche et Développement, Thales	Examinateur
A.A. JERRAYA	Directeur de Recherche CNRS, TIMA, INPG Grenoble	Rapporteur
E. MARTIN	Professeur des Universités à l'Université de Bretagne Sud	Examinateur
P. QUINTON	Professeur des Universités à l'Université de Rennes I	Rapporteur
O. SENTIEYS	Professeur des Universités à l'Université de Rennes I	Examinateur

Laboratoire d'Electronique des Systèmes Temps Réel (LESTER)
CNRS FRE2734
Université de Bretagne Sud, Lorient

Remerciements

Je remercie tout particulièrement Monsieur *Eric Martin*, Professeur des Universités à l'Université de Bretagne Sud et Directeur du LESTER, pour m'avoir accueilli dans son équipe de recherche et pour avoir accepté d'être mon directeur de thèse. Je le remercie également pour les conseils qu'il m'a apportés tout au long de ces trois années.

Je tiens à remercier Monsieur *Emmanuel Casseau*, Maîtres de Conférences à l'Université de Bretagne Sud pour sa disponibilité son encadrement, son enthousiasme et sa gentillesse tout au long de cette aventure.

Je remercie Monsieur *Olivier Sentieys*, pour m'avoir fait l'honneur d'être président du jury.

Je remercie Monsieur *Amhed Amine Jerraya* et Monsieur *Patrice Quinton*, pour avoir accepté d'être rapporteur de ce manuscrit.

Je remercie Madame *Anne Marie Fouillart* pour avoir accepté d'examiner ces travaux de recherche.

Je tiens à exprimer ma gratitude à *Sylvain Huet* pour sa gentillesse, son soutien, son écoute et ses conseils. Il en va de même pour l'ensemble de mes *nombreux voisins* de bureau durant ces 3 années.

Merci à tous les membres du laboratoire *LESTER*,

Enfin, je tiens à exprimer ma profonde reconnaissance à *Rachel* pour avoir su me supporter durant ces longs mois.

Table des matières

Table des figures

Introduction

Les travaux relatifs à cette thèse sont menés dans le cadre de la conception des systèmes sur puce (SoC) en considérant conjointement 2 axes de progrès :

1. la réutilisation de blocs préconçus, concept bien connu en développement logiciel, qui s'applique dorénavant également aux éléments tant matériels que logiciels des systèmes sur silicium dont la complexité et l'hétérogénéité sont grandissantes,
2. la synthèse de haut niveau, qui permet de passer de manière automatisée d'une spécification algorithmique d'une application à la spécification de son architecture matérielle.

Le concept de *composant virtuel de niveau comportemental*, proposé par le LESTER, autorise une grande flexibilité et une bonne adéquation entre algorithme et architecture. Ce type de composant est spécifié sous forme algorithmique et est destiné à être synthétisé par des outils de synthèse de haut niveau. Dans ce contexte, nos travaux adressent plus spécifiquement la prise en considération des contraintes imposées par les applications de Traitement du Signal et de l'Image (TDSI) dans le processus de synthèse de haut niveau.

Comme dans tout processus devant s'exécuter en *"temps réel"*, ou temps contraint, les indéterminismes contenus dans la spécification algorithmique (exécutions dépendantes du contexte ou des données) posent des problèmes théoriques de modélisation mais également d'exécution. En effet le couple (modèle de représentation, contraintes supportées) utilisé par l'outil de synthèse contraint les domaines d'utilisation possibles. Le modèle de représentation utilisé pour modéliser l'ensemble des traitements à effectuer peut restreindre les primitives algorithmiques acceptées dans la description comportementale. De son coté, l'outil de synthèse employé doit permettre la prise en compte de l'ensemble des contraintes d'intégration du concepteur et y apporter une réponse adaptée.

Nous nous proposons dans ce mémoire d'adresser cette problématique en considérant plus particulièrement le modèle de spécification, le modèle architectural et les transformations qui permettent d'automatiser la synthèse de haut niveau.

Plan du mémoire

Le premier chapitre de ce mémoire est consacré à la problématique générale relative à la conception des *systèmes-sur-puce (SoC)*. Nous détaillons les différentes étapes du flot de conception et présentons les

travaux associés. Nous passons ensuite en revue les différentes techniques utilisées pour l'adéquation algorithme architecture. Nous mettons alors en avant les problèmes spécifiques de la réutilisation de composants virtuels dans des applications TDSI temps réel et proposons l'utilisation de la synthèse de haut niveau pour concevoir les systèmes correspondants.

Le chapitre II présente un état de l'art des techniques permettant l'implémentation matérielle des primitives algorithmiques (boucles, conditions, adressages dynamiques de données) présentes dans une grande majorité des applications de TDSI actuelles. Nous nous intéressons plus particulièrement à la gestion des structures itératives, des structures conditionnelles ainsi qu'aux accès aux données structurées.

Le chapitre III définit le modèle de représentation utilisé qui est basé sur les graphes. Ce modèle, décorrélé de l'architecture d'implémentation, permet de modéliser les éléments sémantiques des applications de TDSI complexes. Plusieurs primitives sont détaillées telles les structures itératives, les structures conditionnelles, les accès aux données structurées, etc. Ce modèle est ensuite exploité dans les différentes étapes du processus de synthèse.

Le chapitre IV présente le modèle d'implémentation utilisé pour la synthèse architecturale du modèle de représentation. Le flot de synthèse de haut niveau associé est présenté dans un second temps. Une architecture cible originale, basée sur l'utilisation de séquenceurs d'accès à la mémoire, permet de considérer efficacement des séquences d'accès déterministes mais également des séquences d'accès indéterministes. Cette dernière est optimisée grâce à l'implémentation d'un chemin de données placé dans le séquenceur mémoire.

Le chapitre V confronte nos travaux à des applications TDSI et met en avant l'intérêt du modèle de représentation et des méthodes de synthèse proposées. Nous illustrons la méthodologie dans un premier temps à l'aide d'exemples pédagogiques, puis nous montrons la capacité de notre approche et de notre outil à traiter des problèmes de complexité importante (travaux réalisés dans le cadre du projet RNRT *"ALIPTA"*).

Enfin, le dernier chapitre conclut ce manuscrit en résumant les apports de notre approche et les perspectives qui s'en dégagent.

Chapitre 1

Conception de Systèmes Électroniques Embarqués

Si la complexité des applications de TDSI en général et Télécom en particulier [WWRF] [ITRS03] croît exponentiellement en accord avec la complexité d'intégration des circuits [ITRS03], tous les acteurs s'accordent à souligner le point d'achoppement que forment les outils de CAO. La complexité des applications oblige à fractionner leurs conceptions pour des équipes de spécialistes. A ce stade, la réutilisation est un facteur d'économie de temps fondamental. La discontinuité du flot de conception induit la nécessaire vérification des résultats de synthèse et ajoute un délai et un coût majeur à la conception. Dans ce chapitre, nous introduisons d'abord le contexte de la conception de systèmes intégrés. Puis nous présentons quelques pistes de progrès envisageables. Nous nous intéressons, entre autre, au concept de composants virtuels de niveau algorithmique et présentons l'intérêt que peut avoir la synthèse d'architecture dans le développement de systèmes de TDSI.

1.1 Introduction

Le marché actuel des composants électroniques est fortement orienté vers les produits grand public qui ciblent de plus en plus des applications couplant les télécommunications et le multimédia : téléphones mobiles, *PDA (Personal Digital Assistant)*, systèmes *GPS (Global Positioning System)*, jeux et lecteurs vidéo... L'*"International Technology Roadmap for Semiconductors"* [ITRS03] prévoit que les processeurs contiendront 97 millions de transistors en 2007 et 1,5 milliards en 2013. L'évolution permet déjà de combiner les fonctions d'un téléphone, d'un navigateur Internet, d'un appareil photo numérique, d'un caméscope, d'un lecteur multimédia et d'un PDA au sein d'un unique objet portable. Les systèmes électroniques mobiles de demain, dit de troisième ou quatrième génération, ont un marché potentiel dont le revenu mondial est estimé à 320 milliards de dollars US à l'horizon 2010.

Dans ce contexte, les systèmes complets sont depuis quelques années intégrés sur une même puce nommée *Système-sur-Silicium (SoC)*. Cette puce se compose de parties logicielles et matérielles afin de respecter les contraintes drastiques en terme de consommation électrique, de performances temporelles et

YEAR OF PRODUCTION	2003	2006	2009	2012	2015	2018
Process Technology (nm)	101	90	65	45	32	22
Supply Voltage (V)	1.2	1	0.8	0.6	0.5	0.4
Clock Frequency (MHz)	300	450	600	900	1200	1500
Application (maximum required performance)	Still Image Processing	Real Time Video Codec (MPEG4/CIF)		Real Time Interpretation		
Application (other)	Web Browser Electric Mailer Scheduler	TV Telephone (1:1) Voice Recognition (Input) Authentication (Crypto Engine)		TV Telephone (>3:1) Voice Recognition (Operation)		
Processing Performance (GOPS)	0.3	2	14	77	461	2458
Required Average Power (W)	0.1	0.1	0.1	0.1	0.1	0.1
Required Standby Power (mW)	2	2	2	2	2	2
Battery Capacity (Wh/Kg)	120	200	200	400	400	400

FIG. 1.1 – Projection des besoins pour les processeurs embarqués dans les PDA [ITRS03].

de coût. Les parties contenant des traitements intensifs sont effectuées par des accélérateurs matériels tandis que les parties de contrôle sont plutôt dévolues aux microprocesseurs. Ces systèmes complexes sont aussi contraints typiquement par le besoin de développement rapide qu'exigent les produits grand public et l'hétérogénéité des circuits à concevoir.

Nous allons aborder dans ce chapitre différentes méthodologies qui répondent à cette complexité croissante tout en exploitant au mieux les progrès technologiques.

1.2 Flot de développement de niveau système

1.2.1 Évolutions technologiques prévues

Le marché de l'électronique s'oriente, pour une grande part, vers des produits grand public visant principalement des applications de télécommunications et multimédias. La croissance des besoins est soutenue par l'augmentation des capacités d'intégration : l'*International Technology Roadmap for Semiconductors* [ITRS03] prévoit que la puissance de calcul nécessaire dans les applications embarquées va être multipliée par 1200 entre 2003 et 2015 (figure 1.1). Cette augmentation de la puissance de calcul autorisera l'intégration d'une multitude de fonctions des domaines du multimédia et des télécommunications dans un seul appareil portable.

L'augmentation de la puissance calculatoire sera assurée en partie grâce à l'augmentation de la fréquence de fonctionnement (x3). Afin de conserver des niveaux de consommation électrique équivalents permettant la mobilité des systèmes nomades, les niveaux de tension interne au coeur du circuit seront descendus. Cette diminution des tensions d'alimentation sera autorisée grâce aux progrès technologiques réalisés sur la finesse de gravure des circuits.

L'augmentation des performances des applications multimédias embarquées se fait actuellement ressentir, en particulier dans les systèmes nomades où la complexité des applications augmente constamment.

Nous allons aborder, à titre d'exemple, cette thématique dans la prochaine partie.

1.2.2 Augmentation de la complexité des applications

L'augmentation de la complexité des applications s'est accompagnée d'une augmentation des capacités des canaux de communication. En effet, ces dernières années les besoins en terme de transfert d'informations ont augmenté de manière exponentielle conjointement aux besoins des applications. L'augmentation des débits de communication s'explique par la croissance des applications multimédias embarquées dans les applications nomades :

1. Dans la 2G, on ne pouvait transmettre que la voix et des messages textuels (*SMS : Short Message Service*), la vitesse de transmission d'informations numériques atteignait 9,6kbits/s.
2. La 2.5G a vu arriver les messages multimédias (*MMS : Multimedia Message Service*) qui ont permis de transmettre des photos et des fichiers audios. De plus cette génération a permis, pour la première fois, de se connecter à Internet au travers des services WAP. La vitesse de connexion a augmenté de façon importante (jusqu'à 384kbits/s).
3. L'actuelle 3G (basée sur la norme de communication *(UMTS)* offre des débits supérieurs (2Mbits) qui permettent de transmettre en temps réel des contenus multimédias (vidéo).

Atteindre plus de 50 Mbits/s pour répondre aux besoins futurs de la 4G, tel est typiquement le challenge de demain. Les recherches actuelles misent sur les modulations de type *MIMO (Multiple Input Multiple Output)* pour permettre une augmentation de la capacité de transfert. Différents standards utilisant ces recherches se profilent suivant les zones géographiques d'utilisation. De plus, ces mêmes standards établissent un besoin de flexibilité dans les chaînes de réception afin de s'adapter aux conditions de transmission et de réception. Ces techniques doivent s'adapter à la qualité de la transmission pour assurer des taux d'erreurs faibles en toute circonstance. Pour réaliser ces opérations de vérification / correction des trames reçues, différentes techniques sont actuellement exploitées [Wick99] [Blah02] [More02] : les codeurs et décodeurs de *Viterbi*, de *Reed Solomon*, les *Turbocodes*, les *LDPC (Low Density Parity-Check)*, etc. S'adapter aux normes et aux applications implémentées sur les systèmes nomades impose une flexibilité des circuits numériques.

La figure 1.2 nous montre que les couches applicatives nécessitent, elles aussi, de la flexibilité afin de s'adapter aux contenus numériques à traiter car le nombre et le type d'implémentations possibles des différents étages d'une chaîne de transmission compatible 3G sont importants.

La complexité des algorithmiques de codage source (et de décodage dans le récepteur) a augmenté drastiquement depuis la parution des premières normes de codage des informations multimédias. Il va de même pour les techniques de décodage canal (dans les chaînes de réception).

Afin de palier à ce besoin toujours plus important de performances (complexité calculatoire, flexibilité), différentes pistes de progrès sont explorées afin de répondre aux besoins exprimés par les concepteurs de tels circuits.

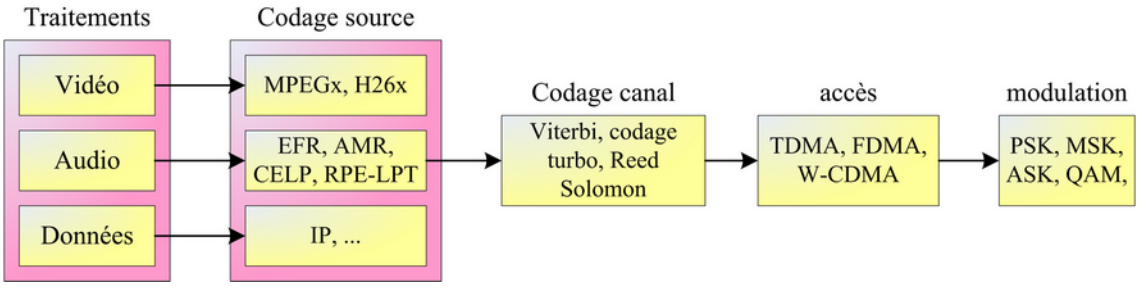

FIG. 1.2 – Exemple de chaîne de transmission adaptée à la 3G.

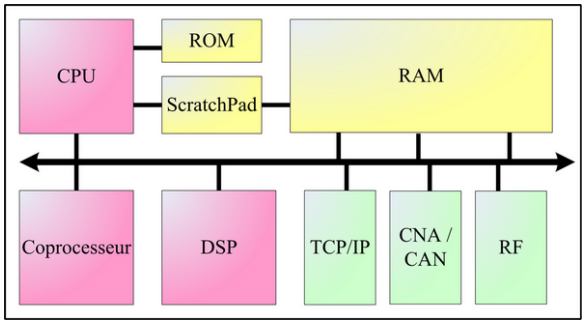

FIG. 1.3 – Exemple d'une architecture de SoC.

1.2.3 Pistes de progrès

Afin de répondre aux différentes problématiques liées à l'intégration de systèmes flexibles toujours plus complexes, les intégrateurs ont développé différentes approches permettant de réduire les temps de développement tout en augmentant la puissance de traitement des systèmes conçus. Les applications et les fonctions sont intégrées dans une seule et unique puce *(System on Chip)* qui contient des parties logicielles et matérielles. Compte tenu des contraintes de consommation électrique et de performance temporelle des systèmes embarqués actuels (et futurs), la conception de circuits numériques doit en général intégrer des accélérateurs matériels pour le traitement intensif. Par ailleurs, comme les fonctions multimédias et de télécommunication vont devoir s'adapter au contexte courant, des solutions logicielles vont être mises en oeuvre afin de permettre une adaptation de l'architecture à ses nouveaux paramètres de fonctionnement. Pour développer de tels systèmes, les équipes de recherche réfléchissent donc à des méthodologies de conception qui intègrent une gestion des différentes parties composant l'application (calcul, contrôle, mémoire, communication). A cette difficulté de conception s'ajoute la contrainte de développement rapide des produits grand public et l'hétérogénéité des circuits (analogique/numérique, multiples domaines d'applications, contrôle/données, matériel/logiciel (fig. 1.3)) qui demande un ensemble de compétences difficiles à réunir au sein d'une même équipe de concepteurs.

Actuellement la majorité des ASIC développés comporte donc au moins un CPU embarqué. La tendance actuelle consiste à intégrer des réseaux formés de plusieurs processeurs dans le cas d'applications complexes (exemple : les SoCs de consoles possédant des processeurs dédiés aux graphismes, à l'audio, au réseau, ...). Les mécanismes de communication inter-processeurs à l'intérieur d'un SoC sont générale-

ment mis en oeuvre autour de bus partagés, mais au vu de l'accroissement de la complexité pour les années à venir, la tendance est à la mise en oeuvre de réseaux sur puce nommés *NoC (Network on Chip)* [Beni02] [Jant03]. On peut voir un SoC comme un assemblage de composants matériels standardisés pour en faire un système comme on le fait pour les cartes. Ainsi, le problème n'est plus tant de concevoir des composants matériels efficaces mais surtout d'obtenir une architecture système qui respecte les contraintes (performance, coût) imposées. En ce qui concerne les composants matériels, le problème devient plutôt de s'assurer de leur validité fonctionnelle dans le contexte applicatif avant de les utiliser.

Nous présentons plus en détail dans la suite de ce paragraphe quelques axes de progrès permettant de répondre aux besoins de conception des SoC.

La réutilisation de blocs préconçus

La réutilisation de composants préconçus est un concept bien connu dans le domaine de la conception de logiciels. Cette technique s'applique depuis quelques années à la conception des systèmes sur silicium : plutôt que de concevoir intégralement toutes les fonctions d'un système, il peut s'avérer intéressant d'utiliser (ou plutôt de réutiliser) des fonctions déjà développées en interne ou par une tierce partie dont la complexité et l'hétérogénéité sont grandissantes. Les composants ainsi réutilisés sont nommés *Composant Virtuel* (ou *Intellectual Property (IP)*). Actuellement, les composants virtuels réutilisés au sein de différents *designs* sont majoritairement les processeurs généraux *(CPU)* et spécialisés *(DSP)*, les interfaces de communication normalisées *(PCI, PCI Express, USB)* ainsi que des blocs dédiés au traitement intensif de données.

La réutilisation implique généralement deux acteurs différents : le fournisseur (celui qui conçoit le bloc) et l'utilisateur (celui qui le réutilise) [Keat98] [Seep00] [Cava03]. Passer de l'un à l'autre suppose que l'ensemble des informations est échangé sans ambiguïté (performances, interfaces, vérifications, etc ...). Les règles de standardisation classent les composants virtuels matériels en trois familles suivant leur niveau d'abstraction [VSIA97] :

1. *HARD IP (ou IP matériel) :* le composant est délivré au niveau physique : *Netlist* entière, routage et optimisations pour une librairie technologique spécifique, *Layout* personnalisé ou une combinaison des deux. Cela permet d'optimiser la puissance, la taille ou les performances pour une technologie spécifique. L'inconvénient de ce type de composant est qu'il n'est pas flexible ni portable car le processus est dépendant de la technologie. Par contre, il a l'avantage d'être entièrement prédictif.
2. *SOFT IP (ou IP logiciel) :* le composant est livré sous sa forme HDL synthétisable. Le principal avantage de ce type de composant est qu'il est flexible (certains paramètres génériques de l'architecture peuvent être modifiés par le concepteur) et donc potentiellement réutilisable dans de nombreuses applications. L'inconvénient majeur est qu'il ne peut être très prédictif en terme de superficie, puissance et temps.
3. *FIRM IP (ou IP flexible) :* avec ce type d'IP délivré sous forme de *Netlist*, il est possible d'optimiser la structure et la topologie pour les performances et la superficie à travers le placement des composants sur le plan et dans la topologie. Le composant virtuel FIRM offre un compromis entre les composants *SOFT* et *HARD*.

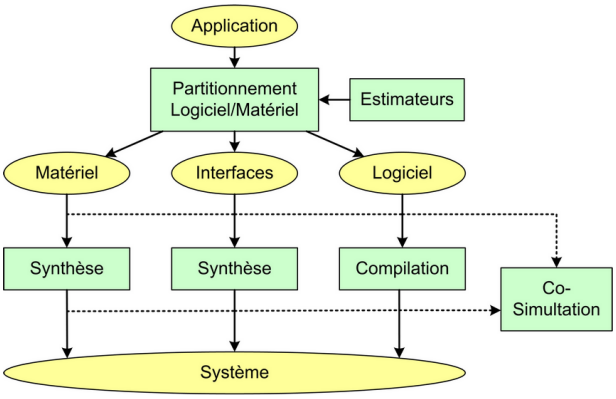

FIG. 1.4 – Flot typique de conception conjointe logiciel/matériel.

Les techniques de développement conjoint

Afin de répondre aux contraintes des applications actuelles (vitesse, consommation, surface), les concepteurs de système doivent adapter au mieux les ressources matérielles et logicielles mises en oeuvre avec les besoins de l'application. La mise en oeuvre de parties matérielles et logicielles lors de la conception d'une application également appelée *Conception Conjointe Matériel/Logiciel (ou Hw/Sw Co-Design)*, consiste à faire coopérer les deux types d'implémentations au cours de leur développement [Bala97] [DeMi02] (figure 1.4). Les méthodologies et outils de *co-design* logiciel/matériel réalisent en premier lieu une étape d'exploration architecturale afin de guider le concepteur vers une implantation efficace de chacune des fonctions composant l'application traitée. La phase d'exploration architecturale est basée sur le calcul de métriques (parallélisme, taux d'accès à la mémoire, besoins de communication, etc.). En fonction de ces métriques et de bibliothèques d'estimation, l'outil va générer un choix d'implémentation pour l'ensemble des blocs fonctionnels.

Dans un second temps, les différents modules ainsi que leurs interfaces de communication vont être implémentés en fonction des décisions prises durant l'étape de partitionnement. L'implémentation des parties logicielles et matérielles peut être réalisée manuellement (développement de code ASM ou RTL) ou à l'aide d'outils adaptés (compilateurs ou outils de synthèse).

La dernière étape consiste à valider fonctionnellement le système ainsi obtenu même si des validations fonctionnelles par co-simulation sont pratiquées durant l'ensemble du processus de raffinement du système. En effet, le temps de simulation d'un circuit numérique dépend principalement du niveau d'abstraction de la description des composants qui le composent. Il faut quelques secondes pour simuler une description d'un système avant synthèse/compilation des composants à plusieurs jours lorsque le système complet est décrit à bas niveau.

Les concepts de reprogrammation du système

Les applications radio logicielles [SDR05] ont récemment été proposées afin de répondre au problème soulevé par la coexistence de nombreuses normes de radiocommunication dans une même région géo-

graphique. Dans de telles applications, il est apparu nécessaire de pouvoir reconfigurer logiciellement et matériellement les stations de base et les terminaux afin de les adapter au mieux au contexte courant. Plus généralement, quelle que soit l'application visée, le principe de reconfiguration dynamique d'un système (ou de certaines de ses parties) permet d'adapter les capacités de calculs aux traitements à réaliser : le composant peut reprogrammer son accélérateur matériel pour qu'il s'adapte efficacement aux calculs (changement de code correcteur d'erreurs afin d'adapter le rendement à la qualité de transmission par exemple).

Cette reconfiguration partielle ou totale des circuits est actuellement possible grâce aux avancées technologiques dans le domaine de la reconfiguration dynamique sur technologie FPGA [Davi04]. La reconfiguration dynamique est obtenue par l'utilisation de FPGA dont la propriété est de permettre la modification des connexions entre cellules logiques (et donc la modification de la fonction) par un simple changement du contenu d'une mémoire interne (XILINX, ALTERA). On peut par exemple imaginer un système disposant d'un FPGA associé à différents plans mémoires, de configurations dédiées à un traitement spécifique. Si par exemple l'application est au préalable décomposée en plusieurs étapes distinctes exécutées séquentiellement, une reconfiguration du système peut être effectuée entre chaque étape d'exécution. La limitation actuelle est due principalement au temps de reconfiguration des FPGA et à leur disponibilité [Boss04]. De nouvelles générations de FPGA avec des temps de configuration très courts sont actuellement à l'étude.

La reconfiguration d'un système s'accompagne d'une reconfiguration des canaux de communication liant les blocs fonctionnels entre eux. Une solution actuellement étudiée consiste à se baser sur une infrastructure de communication elle-même configurable, à savoir un *NoC (Network on Chip)*. Le recours à ce type de solution s'impose pour les circuits de grande dimension, aussi bien du point de vue technologique (asynchronisme entre les parties éloignées d'une même puce), que du point de vue fonctionnel (réutilisation de fonctions (IP) et de blocs reconfigurables). Les communications entre les modules et le *NoC* sont généralement basées sur les standards du groupe *VSIA* (interfaçage signaux et protocole) [VSIA03].

Bilan

Sans toutefois être exhaustives ou suffisantes, les différentes solutions présentées ci-dessus permettent de répondre aux problèmes liés à l'augmentation de la complexité des systèmes. Nous nous intéressons plus particulièrement dans ce manuscrit à la première piste de progrès citée, *la réutilisation de blocs préconçus*. Avant d'aborder plus en détail les travaux de cette thèse, nous présentons différents travaux réalisés au niveau algorithmique et au niveau architectural afin d'obtenir une solution optimale à l'implémentation d'une application par l'approche dite d'*A.A.A (Adéquation Algorithme Architecture)*.

1.3 Conception adéquation algorithme architecture

1.3.1 Les différents travaux

Les nombreux travaux effectués autour de l'adéquation algorithme architecture sont de différentes natures et ciblent soit la description algorithmique de l'application, soit l'implémentation architecturale.

Adéquation algorithmique

Les recherches actuelles visent à réduire la complexité calculatoire des algorithmes en utilisant des solutions ad-hoc, c'est typiquement le cas des applications multimédias. Il existe deux axes de recherches dans le domaine de l'adéquation algorithmique.

Une solution consiste à réduire la complexité d'un même algorithme en transformant les expressions calculatoires. Afin d'optimiser les expressions calculatoires, différentes techniques existent et sont employées dans le monde de la compilation logicielle : propagation des constantes, transformations des boucles [Baco94], élimination des calculs communs, réduction de la complexité des opérateurs [Gupt03a], etc. De nouvelles méthodologies de transformation sont actuellement développées autour des *Taylor Expansion Diagrams (TED)* [Kall02] [Fey04]. Grâce à ces derniers, il est possible de décomposer l'application sous forme de graphe (TED) puis, ensuite, de reconstruire une description algorithmique sous contrainte (exploitation des propriétés des opérations : associativité, factorisation, permutabilité, ...). La description algorithmique obtenue en retour réalise la même fonctionnalité que celle de départ mais bénéficie d'une transformation des expressions calculatoires visant à rendre son implémentation optimale. Des travaux dans ce sens son actuellement en cours au LESTER dans le cadre de la thèse de J. Guillot en collaboration avec M. Ciesielski de l'Université du Massachussetts. Ce type de solution, basé sur la transformation des expressions calculatoires de l'algorithme, est automatisable mais ses résultats en terme de réduction de la complexité de l'application sont limités aux transformations mathématiques.

Une autre solution consiste en l'optimisation des algorithmes à partir de connaissances ou d'hypothèses faites autour de l'exécution du système. Pour illustrer ce type de technique nous pouvons prendre comme exemple les algorithmes d'estimation de mouvements. Une solution optimale, de forte complexité calculatoire, est la recherche exhaustive. Afin de permettre des implémentations de complexité inférieure, divers travaux ont été menés prenant, par exemple, pour hypothèse la faiblesse des mouvements. Cela a abouti au développement de la recherche en spirale avec seuil [MILP00] qui fournit un résultat équivalent à la *recherche exhaustive* avec une complexité calculatoire moindre. Des heuristiques ont aussi vu le jour telles les recherches : en *trois étapes* [Koga81], en *quatre étapes* [Po96], *orthogonale* [Puri87], en *croix* [Ghar90], etc. Ces dernières de complexité bien inférieure fournissent une solution dégradée au problème posé mais répondent au besoin d'implémentation rapide et efficace. Les transformations algorithmiques sont utilisées dans tous les domaines d'applications afin de transformer certaines propriétés d'un algorithme existant pour satisfaire les besoins d'implémentation (ex. des *décodeurs de Viterbi*, des *Turbo-Codes à roulettes* [Gnae03], etc.). Ces manipulations complexes, basées sur des hypothèses/constats, sont plutôt empiriques et ne peuvent généralement pas être automatisées. Bien souvent, la réduction de la complexité est réalisée grâce à l'insertion d'indéterminisme dans les algorithmes modifiés réagissant

aux valeurs des calculs opérés et de ce fait introduisent un indéterminisme sur l'exécution du code (structures conditionnelles, boucles non-déterministes).

Adéquation architecturale

Les recherches effectuées sur les modèles d'architectures permettant l'implémentation d'algorithmes sous contrainte temps réel, tentent d'exploiter le parallélisme entre les opérations. Dans ce cas, les architectures optimales sont typiquement obtenues pour des algorithmes réguliers, sans aléas d'exécution.

Une solution consiste en la mise au point d'architecture ad-hoc par couple (algorithme, contrainte) de manière à obtenir l'architecture d'implémentation optimum en fonction des besoins de l'application (exemple de la transformée en ondelette [Harv01] [Kott04]). Ces optimisations, visant généralement l'optimisation de la surface, de la consommation ou des performances temporelles sont réalisées à l'aide de structures architecturales complexes mises au point spécialement pour réponde aux contraintes du système. Les architectures obtenues sont optimales, mais elles nécessitent des temps de développement importants. De plus les solutions obtenues sont optimales lorsque la contrainte d'implémentation est unique (latence optimale ou consommation optimale, etc.), mais ce n'est généralement pas le cas lorsque l'on combine les contraintes (exemple : fréquence de fonctionnement et en surface).

Une autre voie consiste à générer automatiquement des architectures dédiées aux applications par exemple de traitement du signal et des images. Les architectures ciblées peuvent être distribuées et hétérogènes. Les travaux menés autour de l'outil *Syndex* [Gran00] [Kaou03] permettent de choisir l'implantation qui respecte les contraintes temps réel et minimise les ressources matérielles utilisées. L'implantation sélectionnée est ensuite transformée en un exécutif temps réel distribué. D'autres approches ciblent uniquement la génération d'accélérateurs matériels dédiés à l'implémentation de l'application sous contrainte [Gajs92] [Mar93]. Ces méthodologies, basées sur l'utilisation de méthodes de raffinement automatique produisent des solutions sous optimales en comparaison avec l'utilisation d'architectures ad-hoc, mais les temps de conception s'en trouvent grandement réduits. De plus ces techniques permettent le partage temporel et spatial des différentes ressources de calcul entre les opérations à effectuer lorsque les contraintes temporelles le permettent.

1.3.2 Axe de progrès abordé

Nous venons de présenter brièvement différentes solutions qui visent à surmonter l'augmentation de la complexité et du besoin de performances des applications actuelles. Dans le cadre des travaux présentés dans ce mémoire, nous allons nous intéresser à la génération automatique d'architectures à partir de la description algorithmique et des contraintes du système. Le prochain paragraphe présente ainsi, la synthèse architecturale sur laquelle nos travaux vont se baser.

1.3.3 La synthèse d'architecture

Nous allons dans cette partie détailler le flot typique des outils de synthèse d'architecture. L'objectif de la synthèse de haut niveau (HLS) est de produire une description architecturale de niveau RTL à partir de la

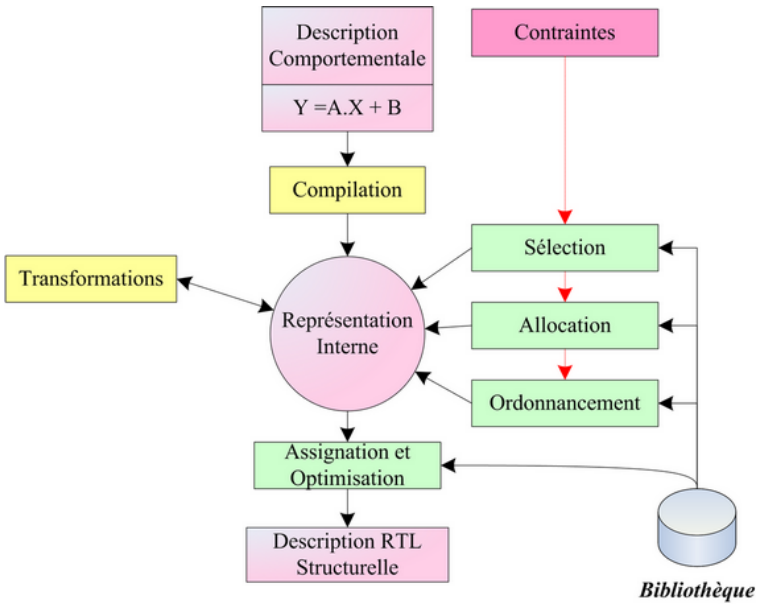

FIG. 1.5 – Flot type de la synthèse d'architecture.

description comportementale d'une application en langage de haut niveau. Ce niveau de spécification est intéressant du fait qu'il s'adresse à la fois tant à une compilation logicielle qu'à une synthèse matérielle. L'automatisation du flot de synthèse de haut niveau permet de réduire considérablement le temps entre la spécification et le silicium. L'ensemble des étapes permettant ce raffinement automatique est présenté dans la figure 1.5.

La première étape du flot de synthèse est l'étape de compilation. Cette étape réalise la vérification syntaxique et sémantique de la description algorithmique et la traduit en un format intermédiaire propre à l'environnement de synthèse. La phase de compilation réalise, de plus, les opérations telles que : l'élimination du code mort, la propagation des expressions constantes, le déroulage de boucles, la mise en ligne des fonctions [Baco94] et la parallélisation du code. Deux types de modèles sont couramment employés dans les outils de synthèse pour la représentation intermédiaire : les graphes flot de données *(Data Flow Graph DFG)* [Gajs92] et les graphes flot de données et de contrôle *(Control and Data Flow Graph CDFG)* [Gajs96]. Le premier modèle permet d'exhiber les dépendances de données et le parallélisme entre les différentes opérations (à condition que l'application soit déterministe). Cette restriction peut être levée en utilisant le modèle *CDFG* qui est plus adapté aux applications orientées contrôle.

Les différentes étapes effectuées par la suite opèrent sur le modèle de représentation interne. L'étape de sélection consiste à choisir la nature des ressources matérielles (opérateurs) qui réaliseront les opérations présentes dans l'application. Le choix des composants se fait sur des critères tels que leur surface, leur vitesse ou leur consommation. L'étape d'allocation détermine, pour chaque type d'opérateur sélectionné, le nombre de ressources à utiliser dans l'architecture finale. L'étape d'ordonnancement a pour rôle d'affecter une date d'exécution à chacune des opérations en tenant compte, d'une part, des dépendances de données et, d'autre part, des contraintes imposées par le concepteur. Ainsi, l'ordonnancement peut chercher à : minimiser le nombre d'étapes de contrôle en fonction d'une quantité de ressources, minimiser

le nombre de ressources en fonctions d'un nombre de cycles d'horloge autorisé ou bien minimiser le nombre de ressources et le nombre d'étapes de contrôle. L'étape d'assignation associe à chaque opération un opérateur matériel dans l'architecture. Notons enfin que l'ordre des étapes de synthèse peut varier selon les outils et les contraintes supportées.

L'association de ces étapes permet d'obtenir une description des différentes unités fonctionnelles au niveau *RTL* ciblant les technologies *ASIC* ou *FPGA*.

Les Différents Types de Contraintes

Pour que l'architecture générée puisse être intégrée dans le système, il est nécessaire qu'elle respecte un certain nombre de contraintes. Ces dernières peuvent être de différentes natures comme nous allons le voir maintenant.

Les contraintes temporelles - Dans le cadre des applications *TDSI*, l'intégration de systèmes se fait souvent sous contrainte temps réel. En effet, les applications doivent traiter un certain nombre d'échantillons par seconde afin de garantir à l'utilisateur une certaine qualité de service (nombre d'images par seconde s'il s'agit de traitement vidéo, etc.). Dans ce type d'applications, les contraintes prioritairement imposées lors de la synthèse sont donc généralement d'ordre temporel. Ces contraintes temporelles peuvent s'exprimer de 2 manières différentes :

1. *Contrainte de latence* - Dans ce cas, le concepteur va fournir une contrainte spécifiant le délai maximum pouvant s'écouler entre le début de l'application (début de l'arrivée des données) et sa date de complétion (fin de production des résultats).
2. *Contrainte de cadence* - Dans le cadre d'une contrainte exprimée sous forme de cadence de réitération, le concepteur spécifie une contrainte temporelle précisant le laps de temps séparant l'arrivée d'une même donnée de 2 itérations successives. Dans ce cas, la latence de traitement des données reçues peut être supérieure à la cadence imposée ; cela conduit à l'utilisation d'une architecture de type pipeline.

Les contraintes temporelles permettent de s'assurer que l'architecture générée respectera les contraintes de débits imposées par le système. D'autres types de contrainte existent, nous allons détailler les plus courantes.

Les contraintes matérielles - Certaines applications ne sont pas nécessairement contraintes par le temps, ou du moins ce n'est pas le critère prédominant. Dans ce cas, la contrainte majeure peut être la surface de silicium autorisée pour l'architecture. Le concepteur contraint l'outil en spécifiant le nombre et le type de ressources (ou la surface totale) autorisés pour implémenter l'application. Cette approche diverge fortement par rapport à la précédente car, dans ce cas, on fixe les ressources matérielles disponibles et l'objectif de la synthèse est alors de réduire la latence globale de l'architecture finale générée.

Autres types de contraintes - Afin de permettre une intégration plus aisée de l'architecture générée, le concepteur peut fournir, dans certains flots de synthèse, des contraintes supplémentaires ainsi que des orientations pour les optimisations qui seront mises en oeuvre dans le processus. Les contraintes de synthèse qui peuvent être employées afin d'ajuster le comportement de l'architecture vis-à-vis du système sont de différentes natures :

- *Entrées/Sorties* - les contraintes d'E/S présentées dans [Cous03] permettent de spécifier un ordre et une date pour la consommation des entrées et la production des sorties. Ces contraintes de niveau algorithmique permettent à l'architecture générée par l'outil de s'affranchir d'un wrapper réordonnant les données. La méthodologie associée permet réduire latence du système en permettant au circuit de commencer les calculs immédiatement après réception des données contrairement aux approches classiques qui attendent généralement l'arrivée de toutes les données d'une itération algorithmique avant de démarrer les calculs.
- *Mémorisation* - les contraintes de mémorisation telles qu'elles ont par exemple été exprimées dans les travaux de G. Corre [Corr05] permettent de contraindre la génération de l'architecture par l'intermédiaire du mapping des données en mémoire. Ce mapping permet au concepteur de spécifier l'architecture mémoire qu'il désire mettre en oeuvre. Selon les choix effectués (nombre de bancs, assignation des données dans les bancs) on autorise un certain parallélisme dans les accès à la mémoire.
- *Consommation* - avec les flots de synthèse actuels, dans le cas de la consommation d'énergie, les contraintes sont plutôt gérées comme des orientations d'optimisation. En effet, l'outil va dans un premier temps respecter les contraintes temporelles ou spatiales, puis va mettre en oeuvre un certain nombre de techniques afin de réduire la consommation d'énergie de l'architecture tout en veillant à ne pas violer les contraintes initiales [Muso95] [Gail98] [Khou98] [Corr05].

Il existe d'autres méthodes permettant d'interagir avec l'architecture ciblée afin d'optimiser particulièrement une sous-partie ou de minimiser certains paramètres de l'architecture. Ces optimisations peuvent, par exemple, viser à réduire la période d'horloge, la longueur des interconnexions [Jego00], la consommation d'énergie, la latence du circuit, les ressources de communication avec la mémoire [Wuyt96] et/ou le reste du système, etc.

Les outils de synthèse d'architecture

Nous avons précédemment évoqué la diversité des applications à intégrer (traitement de données, contrôle intensif ou mixe des deux). En fonction des applications à synthétiser, les outils utilisés pour la synthèse vont différer car les architectures d'implémentation optimales diffèrent en fonction de l'orientation des applications. Pour les applications dominées par le contrôle, les architectures communément générées par les outils de synthèse se composent généralement d'un contrôleur dit "complexe" et de quelques unités de calculs. Les applications orientées traitement intensif des données sont, quant à elles, constituées d'un contrôleur "sommaire" et d'un nombre important d'unités de calcul.

De nombreux outils de synthèse de haut niveau ont été développés au cours des quinze dernières années. Ils se différencient principalement par leur domaine d'application, leurs possibilités et les contraintes qu'ils supportent. Nous pouvons citer par exemple : *GAUT* [Mar93], *SPARK* [Gupt03a], *MMAlpha*

[Guil03], *AMICAL* [Park93], *CATEDRAL* [Raba88], *DEFACTO* [Bond99], *HYPER* [Chu89], *CADDY/DSL* [Camp89], *CALLAS* [Stol92], *PHIDEO* [Lipp91] ou *HERCULES/HEBE* [DeMi88] pour les travaux académiques et *MONET* [Elli00], *Catapult-C* [Cata04b], *PICO-NPA* [Schr00] ou *Behavioral Compiler* [Knap96] pour les produits industriels. Les caractéristiques principales de plusieurs de ces outils de synthèse (DEFACTO, SPARK, PICO-NPA et CATAPULT-C) sont présentées en annexes (A.1).

Dans la prochaine partie de ce mémoire nous allons aborder la conception de systèmes à base de composants virtuels. Nous détaillerons plus particulièrement le concept de composants virtuels de niveau algorithmique basé sur la synthèse d'architecture.

1.4 Concept d'IP algorithmique

1.4.1 Introduction

Afin de réduire l'écart entre les possibilités d'intégration et la complexité actuelle des systèmes, il est nécessaire de disposer d'un flot de transformations automatiques permettant une exploration architecturale plus importante et surtout un raffinement automatisé entre les niveaux d'abstraction.

Dans ce contexte, le LESTER a proposé le concept de *Composant Virtuel de Niveau Algorithmique* [Cass02]. Pour introduire ce concept, nous présentons au préalable le flot de conception système basé sur l'assemblage de blocs préconçus afin d'accélérer la conception des systèmes sur puce.

1.4.2 Conception par assemblage de blocs

La conception de systèmes par assemblage de blocs est une solution qui repose sur la réutilisation et l'assemblage de composants préconçus (paragraphe 2.3.1) [Cesa02]. La méthodologie consiste en l'utilisation de composants virtuels généralement de niveau RTL intégrés dans l'architecture au travers de *wrapper* (fonction d'interconnexion ou adaptateur). L'intégration est validée par un ensemble de simulations fonctionnelles. Un exemple d'assemblage de blocs élémentaires à l'aide d'adaptateurs autour d'un bus partagé est présenté dans la partie inférieure de la figure 1.6.

Cette technique de conception permet le passage plus ou moins facilement de tous types de description système à son implémentation matérielle. Un avantage majeur de cette méthodologie est qu'elle est applicable aussi bien aux applications orientées contrôle intensif, où plusieurs processeurs sont connectés les uns aux autres à l'aide de bus et de mémoires partagés, qu'aux applications orientées traitement intensif des données. Dans ce cas de figure par exemple, un processeur peut être connecté à plusieurs accélérateurs matériels comme cela est schématisé dans la figure 1.6.

L'intégration d'un composant virtuel nécessite en général d'utiliser un *wrapper* qui synchronise les composants (échanges de données, protocole), convertit si besoin est les protocoles entre blocs "incompatibles" et temporise les données pour garantir les contraintes temporelles et l'ordre des données [Keat98]. L'utilisation de protocoles standards comme VCI [OCB00] ou OCP [Son00] n'exempte cependant pas le *wrapper* de la tâche de temporisation des données.

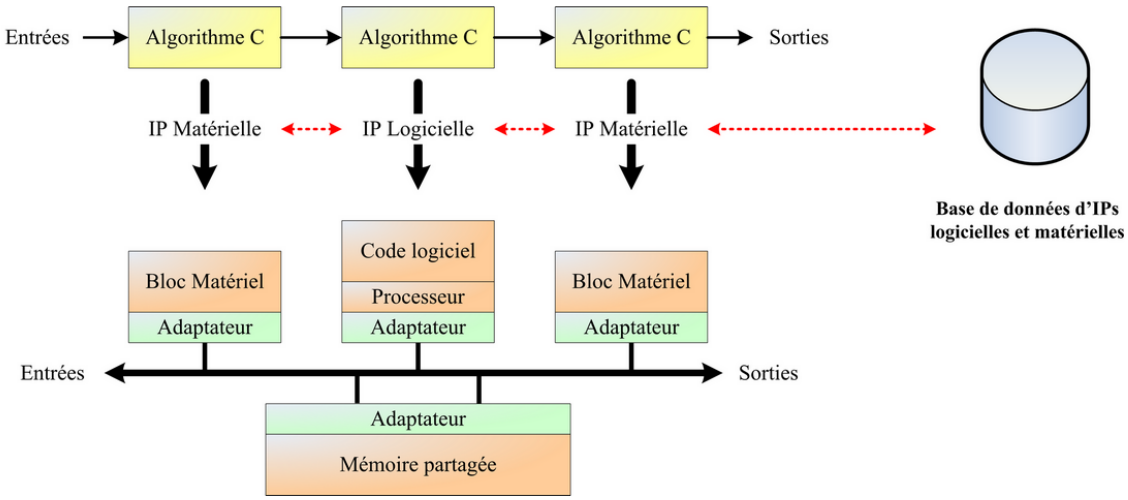

FIG. 1.6 – Exemple de conception d'un système par assemblage de blocs.

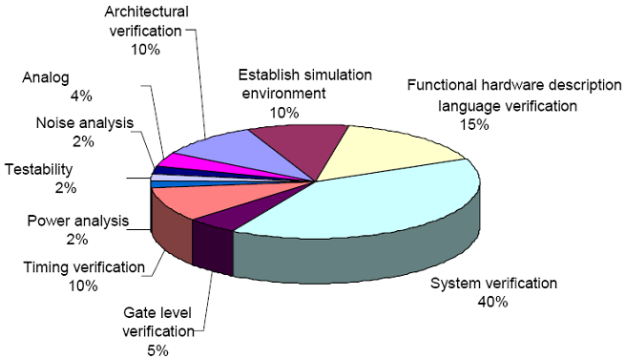

FIG. 1.7 – Répartition du temps passé dans les procédures de validation [ITRS03].

L'insertion de *wrappers* entraîne la plupart du temps une baisse des performances de l'ensemble du système : augmentation de la latence, de la consommation et de la surface. Pour ces raisons, il est nécessaire de valider le système à chaque étape du processus afin de vérifier que l'impact des moyens de communication mis en oeuvre n'aboutit pas à la violation des performances attendues du système. Le temps de simulation d'un circuit numérique dépend principalement du niveau de description des éléments qui le composent. En effet, il faut quelques secondes pour simuler une description d'un système de niveau non temporisé, quelques minutes au niveau temporisé et plusieurs heures ou plusieurs jours au niveau cycle près. Cette évolution exponentielle du temps de simulation engendre des coûts de simulation dans la méthodologie d'assemblage de blocs car les composants assemblés sont décrits plutôt à bas niveau. La figure 1.7, exposée dans les rapports de l'ITRS 2003 [ITRS03], détaille la répartition moyenne des temps de validation d'un système de complexité moyenne composé par assemblage de blocs. La somme des temps passés à valider fonctionnellement le système et à vérifier le respect des performances temporelles imposées représente globalement 50% du temps investi.

Un autre point de blocage, limitant l'utilisation de cette méthodologie dans les applications TDSI, provient de l'absence de prise en compte de l'organisation des échanges et du stockage des données dans les

composants virtuels de bas niveaux. En effet, les systèmes travaillant sur de forts volumes d'information requièrent, afin d'obtenir une solution viable d'implémentation, une évaluation et une prise en compte de l'organisation des échanges et du stockage des données. La réutilisation de composants inadéquats en terme d'entrée/sortie aboutit à une adaptation coûteuse (sinon catastrophique) en terme de latence, débit et surface des transferts de données. Cette problématique connexe aux *wrappers* est considérée dans la thèse à P. Coussy [Cous03].

Le concept de composant virtuel algorithmique présenté ci-après permet de gérer l'insertion des mécanismes de communication entre les composants virtuels. Ce concept a pour avantage de réunir les concepts de *généricité* et *réutilisation*.

1.4.3 Composant virtuel algorithmique

Le manque de flexibilité des coeurs d'IP de niveau RTL et l'usage quasi-indispensable de *wrapper* pour leur intégration est particulièrement pénalisant pour les composants virtuels orientés traitements intensifs, où l'ordre d'arrivée des données influence l'ordre des calculs et le parallélisme exploitable entre les traitements. Afin de s'adapter au mieux aux contraintes du système, l'usage de composants flexibles permettant d'adapter l'architecture d'implémentation aux besoins est nécessaire.

Les travaux présentés dans [Sava02] proposaient une approche de réutilisation des IPs pour les applications orientées traitement du signal, de l'image et des télécommunications. Cette approche est basée sur la notion de *composant virtuel de niveau algorithmique* [Cass02] définis dans le cadre du projet *RNRT MILPAT (Méthodologie et Développement pour les Intellectual Properties pour Applications Telecom)* [RNRT02].

Cette approche propose une élévation du niveau d'abstraction de la spécification d'un composant virtuel afin de bénéficier du potentiel offert par les outils de synthèse de haut niveau pour générer automatiquement une variété d'architectures, respectant une variété de contraintes de performances, à partir d'une unique description paramétrable du comportement d'un composant.

Un composant virtuel de niveau algorithmique consiste donc typiquement en une description fonctionnelle des traitements à réaliser dans un langage de haut niveau *(C, SystemC, Java, VHDL, ...)* qui ne contient aucun détail d'implantation micro-architectural.

A partir de cette description, le rôle des outils de synthèse de haut niveau est de générer une architecture de niveau RTL optimum pour un couple (composant virtuel algorithmique, contraintes d'intégration) donné.

Le flot associé à la conception d'un composant de niveau algorithmique est présenté dans la figure 1.8.

Les contraintes d'intégration qui vont interagir avec le processus de synthèse peuvent être de différentes natures (surface, latence, cadence, consommation, ...). Ce sont ces dernières qui vont contraindre l'outil de synthèse d'architecture dans la génération de l'architecture de niveau RTL réalisant la fonctionnalité souhaitée.

Le flot de synthèse système incluant les IPs algorithmiques est présenté dans la figure 1.9. Par rapport au flot précédemment présenté, l'étape permettant le passage du niveau fonctionnel au niveau architectural

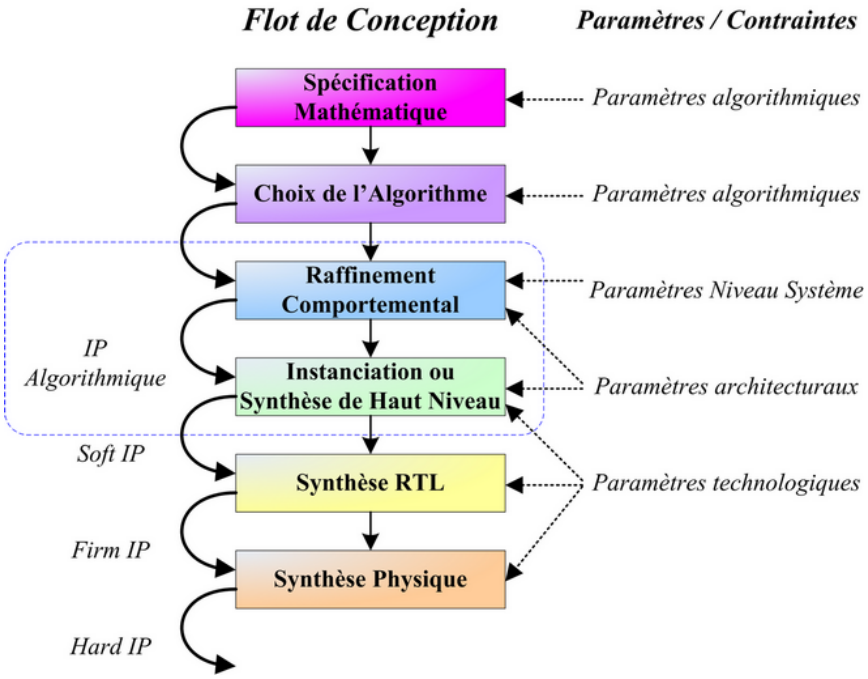

FIG. 1.8 – Flot de synthèse d'un IP Algorithmique.

est automatisée grâce à l'utilisation d'une base de données contenant un ensemble d'IP correspondant aux besoins du concepteur.

1.4.4 Conception à base d'IPs synthétisés par des outils HLS

De manière idéale, la conception de système par synthèses de haut niveau de blocs algorithmiques préconçus repose sur la synthèse de descriptions fonctionnelles élémentaires et l'assemblage de composants ainsi générés [Cass05]. A titre d'exemple, la figure 1.10 représente le passage des descriptions fonctionnelles vers l'ensemble de blocs synthétisés sous contraintes d'intégration avec des connections points à points pour communiquer (cas typique des applications de télécommunication par exemple). Cette méthodologie de conception permet une exploration large spectre de l'espace des solutions envisageables. Le raffinement des descriptions fonctionnelles en solutions architecturales est réalisé sous contrainte d'intégration (latence, cadence, consommation, surface, etc.) du composant au sein du système.

Notre travail s'attache à lever les restrictions imposées au couple (modèle de représentation, contraintes supportées) pour en étendre les domaines d'utilisation. Ainsi, nous nous intéressons au modèle de représentation pour qu'il permette une modélisation de l'ensemble des traitements à effectuer, déterministes ou non, sans restreindre les primitives algorithmiques acceptées dans la description comportementale. Cela permet au concepteur de ne pas recourir à des modifications de description fonctionnelle de l'application, évitant ainsi, des modifications involontaires du comportement.

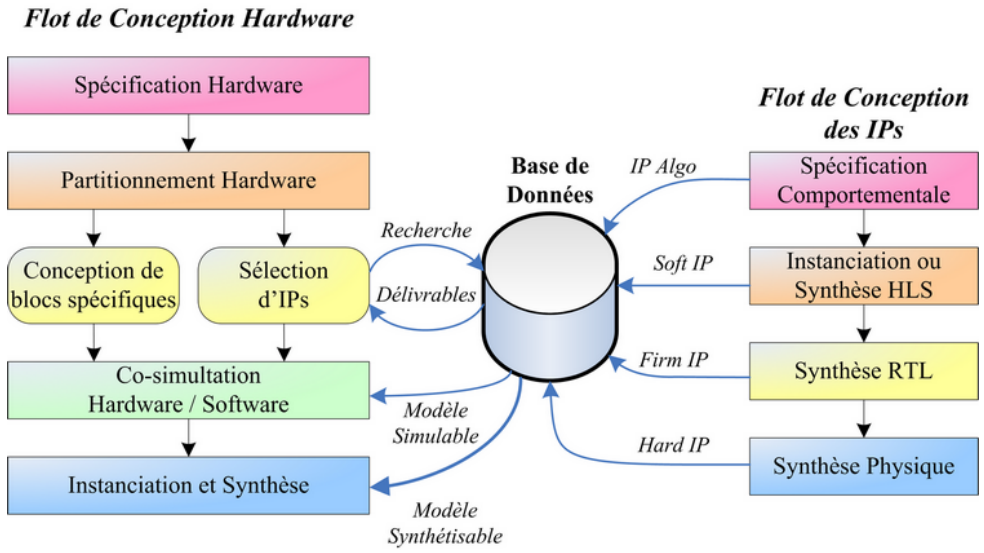

FIG. 1.9 – Flot système d'intégration des IPs.

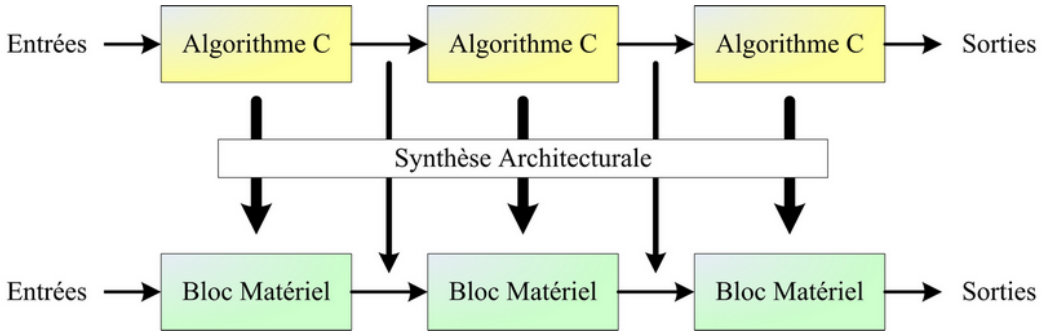

FIG. 1.10 – Exemple de conception d'un système par synthèse de haut niveau de blocs élémentaires (partie matérielle).

1.5 Conclusion

La méthodologie de développement basée sur la réutilisation de blocs de niveau RTL permet la mise en oeuvre rapide d'une architecture à partir de blocs préconçus. Ce type de méthodologie cible tous types d'applications (contrôle / calcul) à condition de posséder les blocs élémentaires nécessaires. La contrepartie de cette méthodologie est la nécessité d'utiliser des adaptateurs matériels pour interconnecter les composants, ce qui engendre des baisses de performance. La validation doit alors être réalisée à l'aide de simulations de bas niveau.

La méthodologie de conception proposée par le LESTER est basée sur la génération de composants matériels à partir de leur description algorithmique et l'usage de la synthèse de haut niveau. Le raffinement des descriptions algorithmiques modélisant le système est réalisé de manière automatique à l'aide de transformations formelles qui permettent de valider la construction obtenue au niveau RTL. La génération des composants est guidée par les contraintes d'intégration du bloc dans le système, ce qui permet entre autre un meilleur interfaçage entre les différents éléments du système. Cette méthodologie permet la génération d'architectures pour des applications déterministes à l'exécution. Toutefois, il est à noter que dans le cas inverse (boucles, conditions, adressages, ...), il devient impossible de générer des architectures temps contraint ainsi que d'optimiser les interfaces à l'aide de contraintes d'E/S.

Nous allons, dans le prochain chapitre, étudier plus particulièrement les différentes techniques de modélisation et d'implémentation des sémantiques présentes dans les descriptions algorithmiques dans les applications TDSI.

Chapitre 2

État de l'Art : Modèles Fonctionnels et Architecturaux

Nous décrivons plus en détail dans ce chapitre les méthodes proposées dans la littérature pour résoudre les points clés de la synthèse de primitives algorithmiques vers les primitives architecturales : boucles, structures conditionnelles et adressages dynamiques. Suite à cet état de l'art, nous mettrons en avant les besoins spécifiques que requiert notre approche.

2.1 Introduction

Les techniques mises en oeuvre pour modéliser et synthétiser les primitives algorithmiques au sein des applications TDSI varient. En fonction des applications et des modèles architecturaux ciblés par les outils de synthèse, les techniques d'implémentation vont également différer, permettant l'obtention de solutions variées couvrant l'ensemble des solutions architecturales. Dans ce chapitre, nous étudions plus particulièrement les techniques d'optimisation et d'implémentation des sémantiques présentes dans les descriptions comportementales des applications TDSI. Dans un premier temps, nous abordons ainsi les structures itératives. Nous continuons avec la gestion des structures conditionnelles et nous concluons par la gestion des accès aux données en mémoire.

2.2 Les structures itératives (les boucles)

Dans cette section, nous allons porter notre attention sur la gestion des boucles *déterministes* et *non déterministes* (cf. 2.1.1) dans la synthèse de haut niveau. Nous évaluerons tout d'abord les effets des techniques de déroulage partiel ou total des boucles, puis dans un second temps nous nous intéresserons aux architectures d'implémentation des boucles avant de conclure sur les méthodes de synthèse des boucles.

```
#define N 100

For(i=0, i<N, i++ ){
        S = A[i] – B;          (Ai)
        T = A[i] + B;          (Bi)
        X[i] = S / T;          (Ci)
}
```

```
#define N 100

S = A[0] – B;          (A0)
T = A[0] + B;          (B0)
X[0] = S / T;          (C0)
… … … … … … … …
S = A[99] – B;         (A99)
T = A[99] + B;         (B99)
X[99] = S / T;         (C99)
```

FIG. 2.1 – Exemple de boucles (a) non déroulée (b) déroulée totalement.

2.2.1 Transformations de niveau algorithmique

Déroulage total de boucle

Le déroulage total des boucles est une technique qui consiste à dupliquer le coeur de la boucle un nombre de fois égal au nombre d'itérations à effectuer afin, en général, de permettre une meilleure exploitation du parallélisme des opérations inter-itérations. Le déroulage total d'une boucle permet d'entrelacer l'exécution temporelle de l'ensemble des opérations des différentes itérations en fonction des ressources matérielles libres. Le déroulage total des boucles n'est applicable qu'à la condition que le nombre d'itérations à exécuter soit connu a priori (le nombre d'itérations ne peut pas varier dynamiquement en fonction de la valeur d'une entrée ou d'un résultat de calcul non prédictible). Certains outils de synthèse comme *GAUT* [Mar93] ou *MAHA* [Park86] appliquent systématiquement ce déroulage aux applications qu'ils doivent synthétiser. Un exemple de boucle roulée est présenté dans la figure 2.1a ; la même boucle déroulée est présentée dans la figure 2.1b (les opérations numérotées A_i, B_i et C_i représentent les opérations effectuées à l'itération i de la boucle initiale).

Boucles déterministes - Une boucle déterministe est une boucle contenant uniquement des opérations de types arithmétique et logique n'agissant pas sur son nombre d'itérations. Dans le cas d'une boucle déterministe (figure 2.1a), le déroulage complet de la boucle permet une exploitation complète du parallélisme inter-itérations contenu dans la description algorithmique (figure 2.1b). Le déroulage complet de la boucle permet la mise à plat des structures de contrôle contrôlant les réitérations, simplifiant ainsi l'implémentation architecturale (plus de contrôleur).

Boucles non déterministes - La description algorithmique présentée dans la figure 2.2a n'est pas *"déterministe"* car le nombre d'itérations à effectuer ne peut être connu a priori. Cependant, la connaissance des cas de fonctionnement de l'application contenant cette structure itérative permet, dans certains cas, de borner la boucle à l'aide du nombre maximum d'itérations. Un utilisateur souhaitant dérouler cette boucle peut ainsi transformer sa description algorithmique (figure 2.2b/2.2c). Cette transformation consiste à remplacer la condition de réitération en utilisant la borne maximum du nombre d'itérations exécutables et à insérer : (1) une structure conditionnelle conditionnant l'exécution du coeur de boucle (fig. 2.2b) ou (2) une instruction de sortie conditionnelle *(break)* interrompant la boucle lorsque la condition validant cette opération est *vraie* (fig. 2.2c).

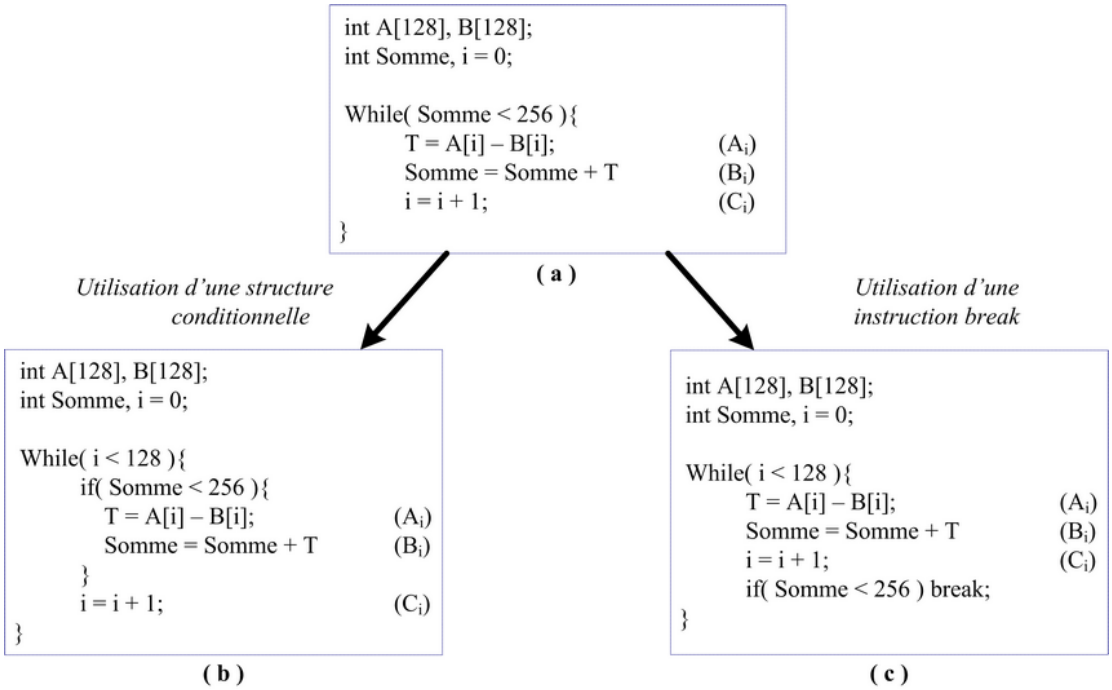

FIG. 2.2 – Transformation d'une boucle non bornée à l'aide d'une borne maximum.

L'utilisation de la borne maximum d'itérations transforme le type de la boucle. Nous obtenons ainsi une boucle déterministe car le nombre d'itérations peut être à présent déterminé statiquement. Il faut noter que le déroulage d'une boucle contenant des conditions d'exécution/d'arrêt mal spécifiées peut avoir des conséquences désastreuses sur la qualité du programme généré ainsi que sur le parallélisme réellement exploitable entre les opérations des différentes itérations. L'effet provoqué par le choix d'une transformation break/conditionnelle est exposé en figure 2.3. La figure 2.3a correspond au déroulage total de la description comportementale 2.2b *(structure conditionnelle)*, la description comportementale de la figure 2.3b correspond à la solution basée sur l'utilisation de la fonction *break* (figure 2.2c).

Comme nous pouvons le constater, lors d'un déroulage intégral ou partiel d'une boucle, l'utilisation de l'instruction *break* se traduit par une cascade de structures *if* imbriquées. Cette imbrication se traduit par des dépendances de contrôle qui limitent le parallélisme exploitable [Elli00] [Knap96]. Si l'utilisation de l'instruction break est naturelle dans le domaine du logiciel où les instructions sont naturellement exécutées en séquences, nous constatons qu'elle peut être pénalisante dans le domaine de la description de matériel.

Conclusion - Nous avons vu que le déroulage des boucles déterministes et le sous-ensemble des boucles non déterministes bornées permet de mettre en exergue le parallélisme de l'ensemble des itérations. En fonction des transformations associées, le déroulage des boucles non déterministes bornées n'est pas nécessairement intéressant car il génère des dépendances de contrôle limitant de manière drastique le parallélisme.

D'une manière générale, le déroulage total d'une boucle peut engendrer au sein du modèle de représen-

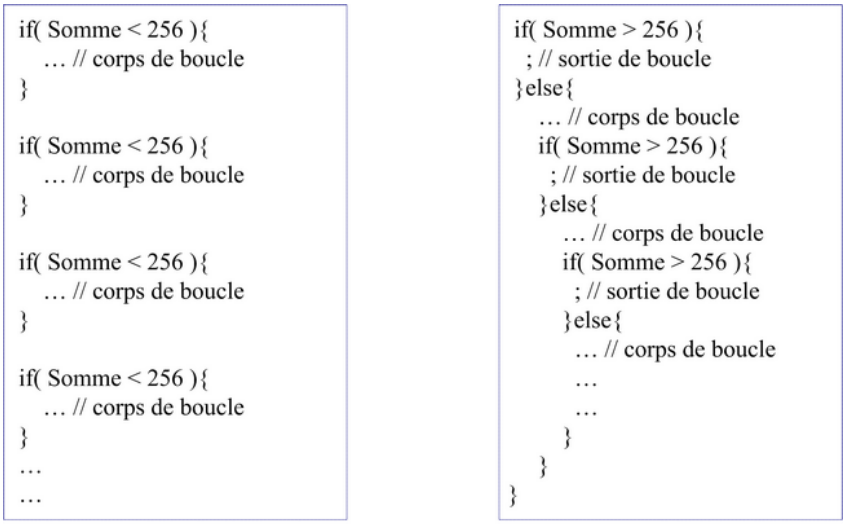

FIG. 2.3 – Déroulage des boucles bornées à l'aide de borne maximum.

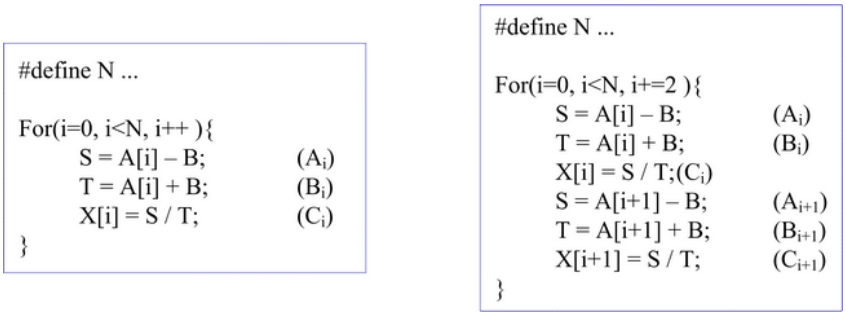

FIG. 2.4 – Exemple d'une boucle (a) originale (b) déroulée partiellement d'un facteur $n = 2$.

tation une augmentation de la complexité qui peut devenir prohibitive (due au nombre de noeuds). Les techniques de non-déroulage ou déroulage partiel peuvent répondre à ce point particulier.

Non déroulage ou déroulage partiel de boucle

Nous venons de présenter la technique de déroulage total des boucles qui permet dans le cas de boucles bornées de mettre en évidence le parallélisme inter-itérations. Dans certains cas, il est préférable de dérouler partiellement les boucles afin par exemple de limiter la complexité du modèle de représentation et les traitements mis en oeuvre sur ce modèle. Une boucle roulée est généralement modélisée par un noeud hiérarchique contenant une itération. Une boucle partiellement déroulée sera représentée par n fois le nombre de noeuds compris dans une itération avec n le facteur de déroulage de la boucle. Un exemple de déroulage partiel d'une boucle est donné en figure 2.4 : la boucle modélisée en 2.4a est déroulée d'un facteur 2, le résultat obtenu est présenté dans 2.4b.

Des études ont porté sur le calcul du facteur de déroulage optimal k qui, suivant les contraintes du concepteur, permet de réduire la latence en augmentant le parallélisme [Card04]. Cette optimisation est réalisée

en fonction des caractéristiques de la cible architecturale. Le déroulage partiel/total permet d'éviter que l'entrelacement des opérations puisse être limité par la rupture du séquencement de l'exécution liée à l'opération de branchement conditionnel. La latence d'une itération de boucle peut ainsi être réduite en minimisant le nombre de branchement conditionnels à effectuer. Il faut souligner que la mise en oeuvre de cette approche depuis un langage haut niveau sans connaissance a priori des capacités architecturales peut, à l'inverse, engendrer une diminution importante des performances. Pour cette raison, il convient, pour chaque architecture et chaque coeur de boucle, d'évaluer l'opportunité du déroulage de la boucle et d'évaluer le nombre de réplications en faveur d'un véritable apport de performances. Cette analyse est souvent confiée aux bons soins de l'utilisateur à cause de sa complexité comme cela est réalisé dans les outils *Catpatult-C* [Cata04a] et *SPARK* [Gupt04].

2.2.2 Modèles d'implémentation

Une fois les transformations de niveau algorithmique effectuées, il est possible d'optimiser l'implémentation de la structure itérative en sélectionnant un modèle d'implémentation adéquate. Nous allons aborder dans cette partie les différents modèles d'implémentation possibles pour les structures itératives.

Implémentation des boucles déroulées totalement

L'implémentation des boucles entièrement déroulées est réalisée de manière similaire à une suite d'opérations indépendantes car les relations de dépendances de contrôle nécessitées par la structure de réitération ont été supprimées. Dans ce cas, l'ordre d'exécution des opérations dépend exclusivement des dépendances de données. Le modèle d'implémentation et les techniques de synthèse mises en oeuvre pour une boucle déroulée sont identiques à celles utilisées pour synthétiser un flot de données.

Implémentation des boucles partiellement déroulées

Les structures itératives partiellement déroulées imposent la gestion d'une structure de contrôle (condition d'arrêt / de réitération). Il existe plusieurs techniques permettant d'implémenter les boucles roulées sur une architecture matérielle. Elles se distinguent par leurs complexités de mise en oeuvre et leurs influences sur la latence d'exécution de la boucle.

Exécution séquentielle des itérations d'une boucle - Les boucles roulées sont vues comme une séquence d'instructions dont l'ensemble se répète un certain nombre de fois. Le nombre des itérations peut être connu a priori dans le cas des boucles bornées, mais peut aussi être inconnu lorsque la réitération est dépendante de l'évolution des données contenues dans le coeur de la boucle. La technique d'implémentation des boucles roulées de manière dite "séquentielle" consiste à abouter les itérations de la boucle lors de l'ordonnancement. Cet aboutement des itérations est généralement employé dans le cas des nids de boucles pour les boucles externes. Ces techniques sont sous optimales car elles ne permettent pas d'exploiter le parallélisme inter-itérations (figure 2.5).

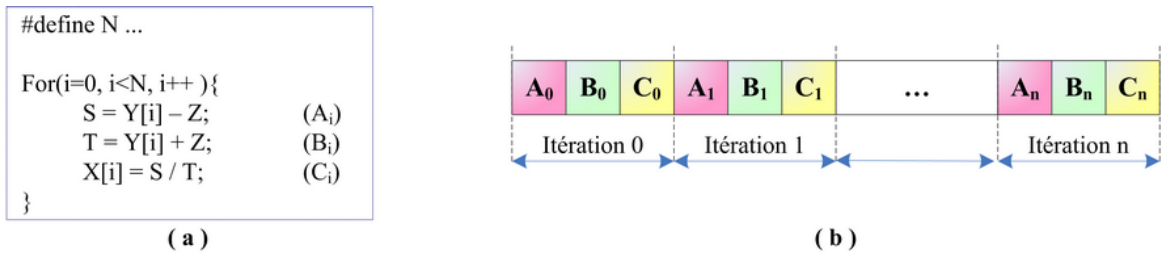

FIG. 2.5 – Implémentation séquentielle de la boucle non déroulée.

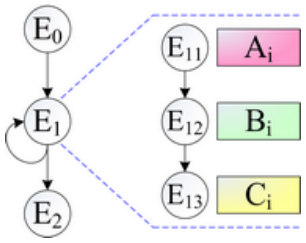

FIG. 2.6 – Contrôleur nécessaire pour une implémentation séquentielle.

Dans cet exemple, les opérations notées (A_i, B_i, C_i) présentées dans la figure 2.5a sont dépendantes les unes des autres. Dans le cadre d'une implémentation séquentielle des itérations de la boucle, nous obtenons la figure 2.5b représentant l'ordre d'exécution des différentes itérations dans le temps. La machine d'états nécessaire à l'exécution séquentielle d'une telle boucle peut être représentée de manière hiérarchique comme cela est représenté dans la figure 2.6.

Si nous réalisons l'hypothèse que chacune des opérations dure 1 cycle $(+, -, /)$, alors l'exécution de l'implémentation séquentielle de la boucle nécessite $3 \times 100 = 300$ cycles (avec $N = 100$) en ignorant le temps nécessaire au calcul et à la propagation de la condition de réitération de la boucle.

Exécution pipeline des itérations d'une boucle - Le pipeline logiciel aussi appelé *"software pipelining"* est historiquement lié aux travaux menés au début des années 80 sur les compilateurs pour les architectures *VLIW (Very Large Instruction Word)*. La technique vise l'optimisation des traitements itératifs définis au sein des boucles en favorisant une meilleure utilisation des ressources matérielles grâce à l'exploitation du parallélisme inter-itérations. Cela permet d'envisager un gain de performances important. L'idée consiste à désynchroniser *(retiming)* l'exécution des opérations de la boucle de telle sorte que plusieurs parties d'une itération de boucle évoluent en parallèle. Le principe du *"pipeline logiciel"* consiste à maximiser l'utilisation temporelle des unités de calcul, exactement à la manière d'un pipeline d'instructions matériel. Le coeur de boucle traitant différentes itérations en parallèle, le traitement nécessite une étape progressive d'initialisation appelée *prologue*, de même que, symétriquement, une partie du code vient graduellement terminer le traitement *(épilogue)*. La complexité de mise en oeuvre du pipeline logiciel est due en partie aux techniques d'ordonnancement qui doivent chercher dans les itérations des opérations qui peuvent être exécutées de manière spéculative ou retardée.

Cette technique peut être étendue à l'aide d'un déroulage partiel des itérations. Les techniques de pi-

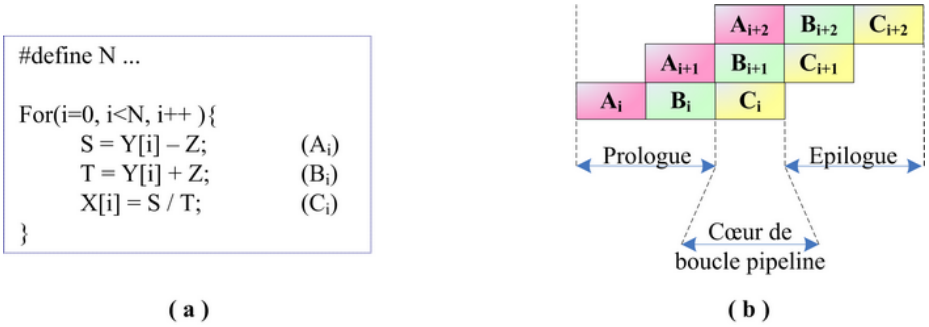

FIG. 2.7 – Implémentation pipeline de la boucle non déroulée.

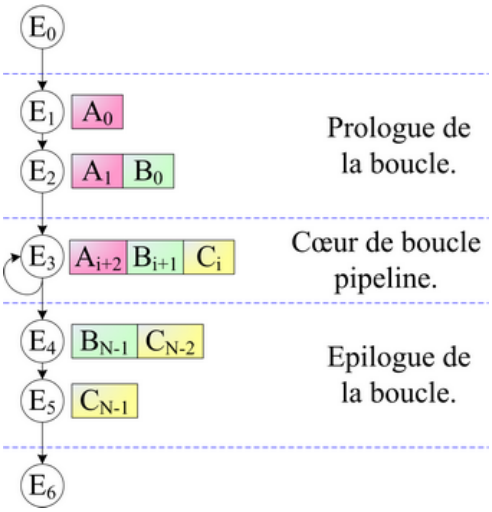

FIG. 2.8 – Machine d'états typique d'une implémentation pipeline.

pelining sont souvent complexes à mettre en oeuvre et ne considèrent généralement que les itérations $\{n-1, n, n+1\}$ afin d'extraire le prologue, le coeur de boucle et l'épilogue. L'exploitation du parallélisme se trouve donc limité aux itérations proches. Afin d'accroître le parallélisme exploitable entre les diverses itérations il est possible de dérouler partiellement les boucles d'un facteur k. Ce déroulage permet de mettre en évidence les dépendances liant les itérations ainsi que le parallélisme entre les opérations. Si nous reprenons l'exemple présenté dans la figure 2.7a et que nous réalisons la synthèse sur ce principe, nous obtenons l'arrangement temporel présenté dans la figure 2.7b, et la machine d'état pilotant l'architecture lors de l'implémentation de la boucle est composée de 5 états (figure 2.8). Le nombre de cycles nécessaire pour l'exécution de la boucle doit être calculé maintenant en prenant en considération le prologue et l'épilogue de la boucle. Dans notre exemple il faut $2+1\times 98+2=102$ cycles. Le gain en nombre de cycles est important dans notre exemple.

Modèle d'implémentation systolique

Les architectures systoliques sont des architectures composées d'éléments de calculs élémentaires interconnectés localement et permettant d'atteindre des taux de parallélisme particulièrement élevés pour des

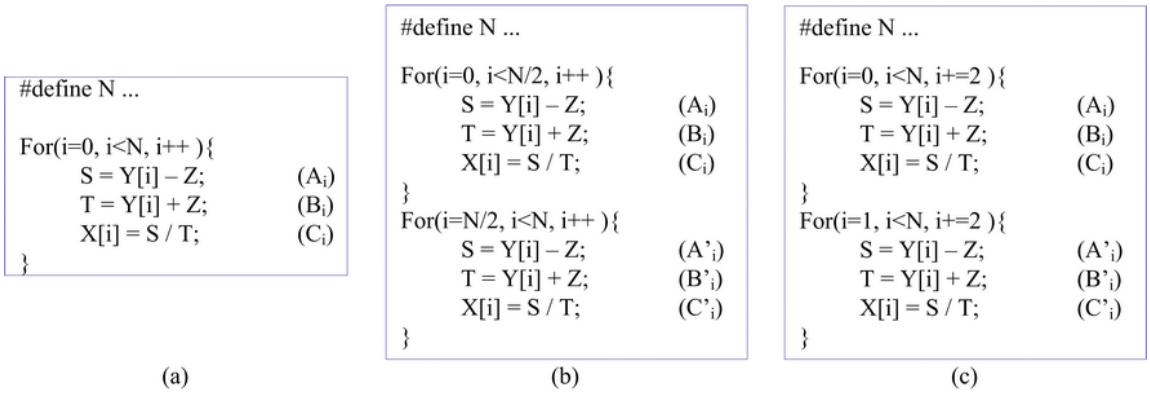

FIG. 2.9 – Exemple de transformations de boucle réalisée afin de distribuer les calculs.

calculs pouvant être mis sous forme récurrente [Quin89]. Les architectures systoliques sont présentées dans les annexes (Annexes *PICO*) de ce mémoire car elles sont ciblées par des outils de synthèse tels *MMAlpha* [Wild94] et *PICO-NPA* [Schr02]. Ces architectures peuvent être efficacement employées dans les applications *TDSI* composées de nids de boucles parfaits. Dans ce cas chaque unité élémentaire de calcul a en charge l'exécution de $\{1 \rightarrow n\}$ itération(s) de la boucle. Grâce à cela l'ensemble des itérations de la boucle à implémenter peut être réparti uniformément sur l'ensemble des unités de calculs disponibles. La boucle à implémenter peut être décomposée de différentes manières afin d'obtenir des itérations indépendantes pouvant être distribuées sur les ressources disponibles. Dans la figure 2.9 nous montrons les transformations qui peuvent être appliquées à la description algorithmique (fig. 2.9a) de la boucle afin de l'implémenter notre exemple sur une architecture systolique composée de deux éléments. La première solution (figure 2.9b) consiste à diviser le nombre d'itérations à réaliser en deux sous ensembles contiguës, la seconde méthode (figure 2.9c) réalise quant à elle un découpage de l'ensemble des itérations sous la forme d'un poinçonnement.

L'implémentation des boucles sur des architectures systoliques bénéficie des techniques de *"software pipelining"*. Par rapport aux solutions déjà proposées auparavant, il faudrait $2+(1\times 98)/n+2=53$ cycles en considérant deux unités élémentaires $(n=2)$ composées d'un opérateur de chaque type pour exécuter l'intégralité de la boucle. En fonction du nombre d'unités élémentaires mises en parallèle, le temps nécessaire va décroître linéairement. Cette augmentation particulièrement importante des performances doit être mise en balance avec le coût prohibitif de mise en oeuvre de cette solution qui peut devenir important (chaque unité élémentaire coûte le prix d'une architecture contenant un contrôleur plus ou moins sophistiqué, plus des unités de calcul nécessaires et un contrôleur global).

2.2.3 Techniques d'ordonnancement des boucles

Nous allons regrouper dans cette partie les techniques d'ordonnancement utilisées pour ordonnancer les boucles. Dans la partie dédiée aux boucles entièrement déroulées nous présentons des techniques qui ont été développées à l'origine pour ordonnancer les flots de données ne contenant pas de structures de contrôle. Nous regrouperons donc dans cette partie l'ensemble des techniques d'ordonnancement dédiées aux flots de données issus ou non de boucles déroulées car leurs traitements sont identiques. Une

introduction à ces algorithmes pourra être trouvée dans [Vemu95] [Jerr96].

Ordonnancement des boucles déroulées

La méthode d'ordonnancement nommée *"List Scheduling"* [Su85] est une méthode d'ordonnancement classant les opérations par liste. Les opérations à ordonnancer présentes dans le modèle de représentation initial sont classées suivant un critère de priorité. Les critères de priorité varient en fonction des objectifs, ils peuvent viser l'optimisation en latence, en consommation d'énergie, etc. A chaque cycle d'ordonnancement, les opérations éligibles sont stockées dans une liste et ordonnées par une *"fonction de priorité"*. Les opérations éligibles de la liste sont assignées si les ressources matérielles nécessaires sont disponibles, sinon elles sont retardées. On recommence ainsi l'opération jusqu'à avoir ordonnancé toutes les opérations présentes dans le modèle de représentation. Les critères de priorité typiquement utilisés lors de l'ordonnancement sous contrainte temporelle sont : la mobilité restante des opérations, l'appartenance au chemin critique, le nombre de successeurs de l'opération, la combinaison mobilité + nombre de successeurs, le quotient du nombre de fonctions du même type qui reste à ordonnancer divisé par le nombre de ressources capables d'exécuter la fonction [Su85]. Une variante de cette technique d'ordonnancement nommée *"Static List Scheduling"* [Jain91] permet de réduire la complexité algorithmique de l'ordonnancement. La réduction de la complexité est basée sur un tri unique des opérations en fonction de la fonction de priorité (priorité non réévaluée en cours de processus). Cette modification impacte sur les performances de l'ordonnancement qui produira des résultats potentiellement dégradés par rapport à la méthode *"List Scheduling"*.

Paulin et Knight [Paul89] ont proposé un algorithme d'ordonnancement permettant de réduire le nombre de ressources matérielles nécessaires pour effectuer un ordonnancement sous contrainte temporelle en répartissant au mieux leurs utilisations sur le nombre de cycles d'horloge autorisés. L'intervalle de positionnement d'une opération est compris entre ses dates d'exécution au plus tôt $(ASAP)$ et au plus tard $(ALAP)$. A chaque opération on associe une probabilité d'être ordonnancée à un cycle d'horloge avec : *Probabilité = 1 / (ALAP - ASAP)*. Pour chaque type d'opérateurs on va créer un graphe de distribution montrant les besoins en terme de calculs de chaque type sur l'ensemble des cycles. Cette information permet de déterminer le nombre de maximum de ressources nécessaires afin d'ordonnancer l'ensemble des opérations tout en respectant les contraintes temporelles. Afin de diminuer le nombre de ressources nécessaires, il faut équilibrer les graphes pour chaque type d'opérateur. Cela est réalisé à l'aide des "forces propres" qui vont permettre de mesurer les variations de la distribution suivant l'ordonnancement des opérations. La force ne reflète que le mouvement d'une opération et ne prend pas en compte les effets de bord engendrés par cette dernière sur l'ensemble du graphe. Une "force totale" est définie pour prendre en considération les répercutions de l'assignation de l'opération sur ses successeurs. Cet algorithme permet de lisser l'utilisation des opérateurs sur l'ensemble des cycles autorisés par la contrainte temporelle fixée.

La méthode *"Iterative ReScheduling"* présentée par Park [Park91] est utilisée pour améliorer la qualité des ordonnancements générés à l'aide d'heuristiques de type *"Static List Scheduling"*. La méthode dite *"Iterative ReScheduling"* va réaliser un réordonnancement de certaines opérations du graphe afin de réduire la métrique de coût employée (latence, surface). Dans la méthode de réordonnancement, on

prend des décisions locales basées sur le déplacement d'une seule opération à la fois : (1) On prend un ordonnancement précédemment réalisé et toutes les opérations de cet ordonnancement (graphe) vont être réordonnancées. Elles vont être déplacées de leur position actuelle soit vers le cycle d'horloge précédant, soit dans le suivant (sans violer les dépendances de données). (2) On choisit "aléatoirement" l'opération à bouger puis on la bloque temporairement sur une position, puis on réalise la même chose avec les autres opérations jusqu'à ce que toutes les opérations soient bloquées. (3) Le coût total de ces mouvements est calculé, et le mouvement produisant le plus grand gain est choisi (et tous les mouvements de la séquence). Puis les opérations sont débloquées, et l'on recommence la procédure sur ce nouvel ordonnancement.

La méthode de réordonnancement nommée *"Simulated Annealing"* [Deva87] réalise le même type d'optimisation sur un ordonnancement déjà effectué. La méthode de recuit simulé va modifier itérativement l'ordonnancement initial en déplaçant une à une les opérations. L'intérêt du déplacement d'une opération est évalué à l'aide d'une fonction de coût. Le coût de déplacement est pondéré de manière à éviter de tomber dans un minimum local. Cette technique permet d'agrandir l'espace des solutions explorées. Les gains que peut apporter une telle méthode sont toutefois contre balancés par une forte complexité algorithmique rendant la méthode inopérante sur des applications complexes.

Ordonnancement séquentiel des itérations

Camposano proposa en 1991 dans [Camp91] un des premiers algorithmes permettant l'ordonnancement des applications orientées flot de contrôle. Ce dernier fut nommé *"Path-Based Scheduling"* car il réalise un ordonnancement indépendant de chacun des chemins d'exécution de l'application. Chaque chemin est optimisé en latence. La complexité de l'ordonnancement est importante $O(2^C)$ avec c le nombre de structures conditionnelles (boucles, branches conditionnelles) contenues dans le graphe à synthétiser. Le flot d'ordonnancement et de synthèse du contrôleur est composé de 3 étapes : (1) Le graphe modélisant l'application à synthétiser est dérivé sous forme d'un ensemble de chemins. Cet ensemble de chemins représente l'ensemble des scénarios d'exécution. (2) Chacun des chemins est ordonnancé indépendamment des autres. Cela permet d'obtenir des chemins optimums en nombre de cycles d'exécution. (3) Ensuite, il est nécessaire de regrouper l'ensemble des chemins ordonnancés de manière optimale afin de réduire la complexité du contrôleur. Cette méthode a pour avantage de permettre une formulation générale de la contrainte que subit l'application : en effet, la contrainte peut être soit matérielle, soit temporelle. Le type de la contrainte ne modifie que la fonction de coût qui décide où couper les chemins lors de leurs fusions. De plus cette méthode gère implicitement le partage d'opérateurs entre les opérations mutuellement exclusives. La latence de chacun des chemins est minimale car ils sont ordonnancés de manière indépendante. En contre partie la méthode souffre d'une augmentation rapide de la complexité qui est directement liée au nombre de chemins évoluant de manière exponentielle avec le nombre de conditions. Comme par ailleurs l'approche ne modifie pas l'ordre des opérations (entre les étapes de contrôle), la meilleure optimisation globale possible (augmentation des délais, ou augmentation du nombre de variables nécessaires pour stocker les résultats intermédiaires) n'est pas garantie.

La méthode d'ordonnancement nommée *"Loop Directed Scheduling (LDS)"* proposée par Bhattacharya dans [Bhat94] prend en considérations les boucles et les branches conditionnelles au sein d'un modèle de représentation interne de type CFG. La méthode est destinée à la synthèse d'applications de type contrôle

intensif. La technique d'ordonnancement se fait sur le même principe que le *"Path-Based Scheduling"* avec un ordonnancement sous contrainte matérielle. Ici aussi, on va chercher tous les chemins potentiellement exécutables dans la description comportementale, et on va les ordonnancer indépendamment les uns des autres. Une métrique est utilisée pour déterminer la qualité de l'ordonnancement ; il s'agit du nombre de cycles du chemin le plus court, divisé par le nombre de cycles du chemin le plus long. Un point important de la méthode est la présence d'une phase de *profiling* du code qui permet de calculer les probabilités d'exécution de chacune des structures conditionnelles afin de pouvoir favoriser certains chemins par rapport aux autres. L'ordonnancement du CFG modélisant l'application se fait de manière équivalente à ce que réalise Camposano dans la méthode appliquée pour le *"Path-Based Scheduling"*. La modification majeure apportée lors de la phase d'ordonnancement, provient du déplacement des opérations à travers des structures conditionnelles lorsque les dépendances le permettent. Ce mouvement autorisé de certaines opérations permet dans le cas des boucles de faire ressortir un prologue et un épilogue et de pouvoir ainsi exploiter le pipeline logiciel. Cette approche n'est pas possible dans l'approche proposée par Camposano car les structures itératives sont mises à plat lors de l'ordonnancement (perte des dépendances de données inter-itérations).

Huang présente dans [Huan93] un algorithme d'ordonnancement nommé *"Tree-Based Scheduling"* dédié à l'ordonnancement d'applications dominées par les structures conditionnelles. L'application à ordonnancer est modélisée en interne à l'aide d'un arbre où chaque noeud et chaque feuille représentent une structure conditionnelle (itération d'une boucle ou branche conditionnelle). A partir de ce modèle de représentation, un certain nombre de transformations va être appliqué. Pour éviter de faire des calculs inutiles, on va déplacer les opérations dans les noeuds où l'on consomme leurs résultats. Les opérations indépendantes des indices de boucle (constantes pour toutes les itérations) vont être déplacées dans l'entête de cette même boucle. Chacun des chemins reliant le noeud *root* de l'arbre à ses feuilles représente ainsi un chemin d'exécution possible comme dans l'approche *path-based*. L'appartenance d'une opération à un chemin ou non permet de prendre en compte l'exclusion mutuelle. L'algorithme d'ordonnancement se déroule en deux phases distinctes : (1) ordonnancement de l'arbre de haut en bas : on optimise chacun des chemins en ordonnançant indépendamment chaque *"basic bloc"*. (2) on réalise un regroupement des états quand cela est possible pour diminuer le nombre d'états total et le coût du contrôle. La gestion des opérations identiques mutuellement exclusives se fait dans cette dernière étape où, lors du regroupement des états, on fusionne les états ayant des opérations identiques.

Ordonnancement pipeline

Les techniques d'ordonnancement pipeline des structures itératives sont nombreuses tant dans le domaine de la compilation logicielle pour processeur que dans le domaine de la synthèse matérielle. Certaines méthodes ont été développées afin de réduire le temps d'exécution des coeurs de boucle. On peut citer *"Percolation Algorithm"* [Nico85] qui a été réutilisé par la suite dans [Nico91] afin de minimiser le temps d'exécution de la boucle à l'aide de transformations locales, cela sous contrainte matérielle. D'autres méthodes ont été proposées afin de combiner des transformations algébriques et des techniques de *retiming* [Potk94] et [Choa94] (associativité et distributivité des opérations). Ces algorithmes peuvent être employés lors de la phase de prétraitement de la description d'entrée, lorsque l'on génère le graphe.

Une méthode d'ordonnancement [Lee92] a été développée afin de proposer une solution à l'ordonnancement sous contrainte temporelle (latence). La méthode va d'abord ordonnancer le graphe en veillant à respecter la contrainte de latence, puis, après une étape de *retiming*, elle tente de réduire les besoins en ressources matérielles. Dans l'outil Cathedral II [Goss89], le graphe flot de donnée modélisant la boucle est *retimé* afin de respecter la contrainte temporelle estimée sans contraindre les ressources dans un premier temps. Ensuite un graphe est dérivé de ce premier ordonnancement et on effectue sur ce dernier un *retiming*, contraint par le nombre des ressources disponibles. De manière itérative, on réduit la contrainte temporelle (latence) acceptée lors de la première phase jusqu'à obtenir un ordonnancement sous contrainte qui convienne aux besoins du concepteur.

La méthode d'ordonnancement *"Percollation Based Scheduling"* (PBS) [Pota90] [Nico91] déroule la boucle de manière incrémentale afin de trouver un motif répétitif dans l'ordonnancement. L'ordonnancement est réalisé sous contrainte matérielle avec, comme unique critère d'ordonnancement, les dépendances de données entre les opérations. Cette unique contrainte permet aux opérations de l'itération n de remonter dans l'ordonnancement (vers l'itération $n-m$) si aucune dépendance de donnée n'est violée par ce déplacement. Ensuite, lorsqu'un nombre suffisant d'itérations a été ordonnancé, il faut trouver le motif de réitération. Une fois le motif identifié, il faut extraire de l'ordonnancement réalisé le prologue et l'épilogue. Le motif représente le coeur de la boucle pipelinée dont la latence est minimum en fonction des contraintes matérielles imposées. Cette technique est étendue afin de prendre en considération les structures conditionnelles contenues dans les boucles. Dans la technique nommée *"Perfect Loop Pipelining"*, la prise en considération des structures conditionnelles est assurée par le traitement de tous les chemins possibles. Dans le cas d'une structure *if-then-else*, deux coeurs de boucle différents sont extraits et pipelinés de manière indépendante.

La technique *"Rotation Scheduling"* présenté par Chao dans [Chao93] répond au problème de synthèse des boucles pipelinées sous contrainte matérielle. Le modèle de représentation utilisé est un graphe de type *DFG* nommée *DAG (Directed Acyclic Graph)* modélisant une itération de la boucle. Le *DAG* est cyclique et ses arcs sont pondérés par le nombre d'itérations liant les dépendances de données. La méthode a pour but d'ordonnancer le graphe acyclique en utilisant la méthode dite de *"Loop Pipelining"* afin de réduire la latence du chemin critique du coeur de boucle (implique une réduction de la latence totale de la boucle). Cette méthodologie ne prend pas en considération les structures conditionnelles. La méthode dite de *"Rotation"* est équivalente à une technique de *retiming* opérant au travers de la structure de contrôle *(Control-Step)* qui symbolise la réitération de la boucle. En réalisant plusieurs *rotations*, un prologue et un épilogue vont ressortir de la première et la dernière itérations. En même temps, un motif correspondant au coeur de boucle va apparaître. De manière équivalente au *retiming*, le graphe doit être préordonnancé. Chao propose un algorithme de type liste contraint par un nombre de ressources matérielles limitée. On peut modéliser l'ensemble des n itérations de la boucle en concaténant les n itérations les unes après les autres (comme dans une exécution séquentielle des itérations). Ensuite l'étape de *rotation* vise à déplacer les opérations de l'itération n progressivement vers les itérations $[n-p, n+q]$. Ces déplacements successifs d'opérations provoquent un compactage du nombre de cycles nécessaires à la réalisation d'une itération. La technique de *rotation* est employée de manière cyclique jusqu'à l'obtention d'un ordonnancement optimal. L'extraction du prologue ainsi que de l'épilogue se fait de manière transparente car ils sont dérivés des décalages du coeur de boucle. L'inconvénient de la méthode provient

du fait qu'elle est basée sur des méthodes de réordonnancement par *retiming* impliquant une complexité algorithmique élevée.

La méthode d'ordonnancement nommée *"Wavesched"* présentée par Jha dans [Laks97] vise les applications dominées par le contrôle. L'objectif de la méthode d'ordonnancement est de réduire le temps moyen d'exécution de l'application sous contrainte matérielle. L'algorithme d'ordonnancement va réduire la latence des différents coeurs de boucles tout en maximisant l'exploitation du parallélisme. Le modèle de représentation utilisé est un *CDFG*. Par rapport aux approches précédemment présentées qui traitent du même problème, la méthode *"Wavesched"* a pour particularité d'ordonnancer de manière parallèle les structures de contrôle indépendantes si les coeurs de boucles ou les branches conditionnelles peuvent partager des ressources matérielles. Les autres algorithmes d'ordonnancement d'applications dominées par le contrôle sont incapables d'exécuter en parallèle des boucles indépendantes car les techniques employées s'interdisent de déplacer les opérations au travers des *"Blocs Linéaires d'Instructions (BLI)"*. L'algorithme d'ordonnancement *"Waveshed"* se base sur une analyse des dépendances entre les entrées et les sorties des structures conditionnelles afin d'identifier les *BLI* pouvant être exécutés en parallèle. Cette analyse est réalisée grâce à la génération d'un arbre hiérarchique représentant explicitement les dépendances entre les structures conditionnelles. Cet arbre permet une modélisation explicite des dépendances ou indépendances entre les structures de contrôle. Au niveau hiérarchique supérieur de cet arbre sont placées les opérations qui ne dépendent d'aucune structure conditionnelle (branche conditionnelle ou boucle). Ensuite, pour chaque structure de contrôle, on crée un niveau hiérarchique. Une fois le graphe hiérarchique modélisant les dépendances de contrôle obtenu, les opérations sont regroupées par blocs. Elles sont ensuite ordonnancées et assignées sur les ressources matérielles disponibles. Une fois qu'une structure de contrôle est ordonnancée, on recherche alors, ses successeurs qui sont maintenant devenus éligibles. Le choix des opérations à ordonnancer au cycle courant est effectué à l'aide d'une heuristique basée sur la longueur des chemins (plus la longueur restant à parcourir par ce chemin est importante pour arriver au noeud puits, plus l'opération est prioritaire). Une fois l'ordonnancement effectué, il faut compacter l'ordonnancement qui vient d'être réalisé en prenant en compte la probabilité d'exécution de chaque chemin. La latence des circuits produits par cette méthode d'ordonnancement est meilleure que celle obtenue par les méthodes de type *Path-Based Scheduling*. En contre partie, le nombre d'états du contrôleur est bien plus important comme le montre la figure 2.10. Cette différence s'explique par le nombre de scénarios d'exécution possibles.

2.2.4 Bilan des différentes approches

Nous venons de présenter un ensemble de techniques d'ordonnancement sous contraintes matérielle et temporelle permettant le passage d'un modèle fonctionnel contenant des structures itératives vers un modèle architectural permettant son implémentation. La majorité des techniques présentées opèrent sous contrainte matérielle visant la réduction de la latence moyenne de l'application après ordonnancement. Les ordonnancements obtenus sont souvent sous-optimaux car les modèles de représentation utilisés basés sur le modèle *CDFG* ne permettent pas un ordonnancement concurrent des structures conditionnelles (*BLI*) indépendantes. La méthode *wavesched* lève cette restriction en permettant l'ordonnancement de boucles en parallèle à condition qu'elles ne possèdent pas de relations de dépendance. Cette méthode

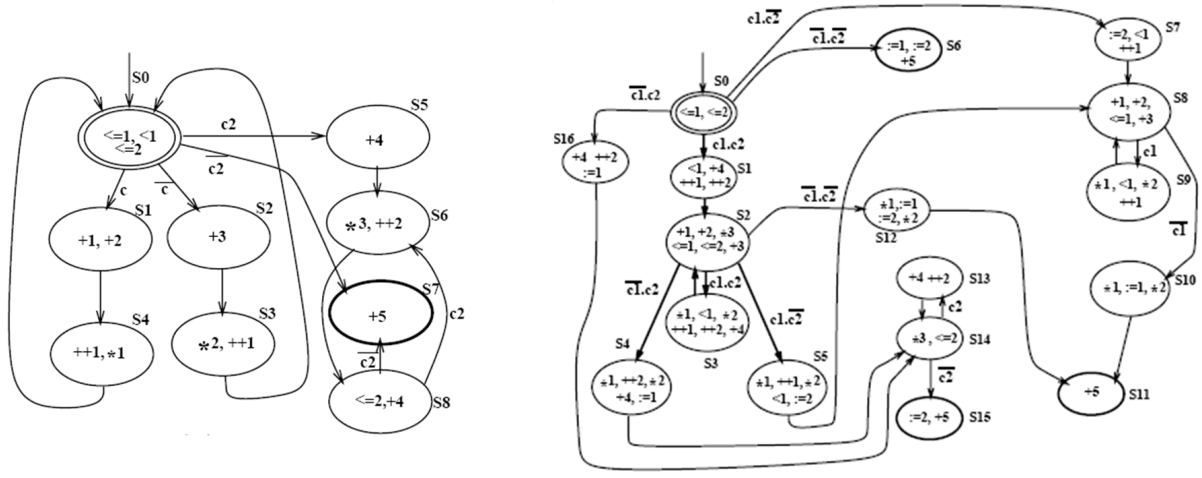

FIG. 2.10 – Machine d'état générée pour une même application [Laks97] (a) Path-Based Scheduling (b) Wavesched.

a cependant pour inconvénient majeur de produire des contrôleurs complexes. Les autres méthodes répondent au problème de l'ordonnancement des boucles sous contraintes temporelles. Parmi l'ensemble des approches présentées pour la synthèse des structures itératives, aucun travail ne cible le problème de l'implémentation de boucles sous contrainte de cadence où l'objectif n'est pas nécessairement une réduction de la latence.

2.3 Les structures conditionnelles

Cette partie du chapitre est consacrée à la prise en compte des structures conditionnelles exprimées à l'aide des sémantiques de conditionnement des opérations *if-then-else* ou *switch-case* dans les descriptions algorithmiques. Nous allons dans un premier temps décrire les techniques décomposant l'application sous la forme d'un ensemble de chemins différents. Dans un second temps nous abordons les optimisations qui peuvent être appliquées à l'application afin de réduire la latence ou bien minimiser le nombre de ressources nécessaires à son implémentation.

2.3.1 Techniques "classiques"

Parmi les techniques permettant l'ordonnancement des structures conditionnelles, nous avons déjà présenté dans la partie précédente, consacrée aux structures itératives, les méthodes nommées *"Path-Based Scheduling"* [Camp91], les méthodes équivalentes *"Tree-Based Scheduling"* [Huan93] et *"Loop Directed Scheduling (LDS)"* [Bhat94]. Ces techniques permettent de prendre en considération les structures conditionnelles lors de l'étape d'ordonnancement grâce au découpage du graphe sous forme de chemins unitaires. Ces méthodes sont basées sur des modèles de représentation de type *CDFG*. Ce modèle est ensuite dérivé afin d'obtenir toutes les séquences (chemins) d'exécution possibles. Ensuite, le traitement de chacun des chemins obtenus de manière indépendante garantit la gestion de l'exclusion mutuelle.

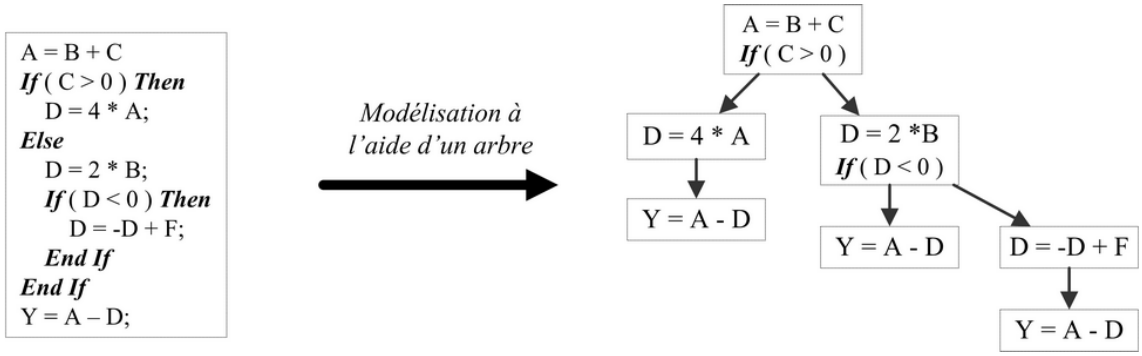

FIG. 2.11 – Représentation des chemins mutuellement exclusifs sous forme d'arbre.

L'inconvénient majeur de la méthode est la nécessité de traiter tous les chemins obtenus (complexité exponentielle en fonction du nombre de chemins).

La méthode d'ordonnancement *"Most Often Used Path Scheduling"* proposée par Ciaran [Ciar01] cible principalement les applications type flot de données contenant des branches conditionnelles. La méthode présentée est une extension du travail de Camposano ; la différence majeure avec la technique *"Path-Based"* est que si deux opérations identiques s'exécutant sur un même opérateur appartiennent à des scénarios d'exécutions mutuellement exclusifs, alors les opérations seront ordonnancées dans le même cycle au final (partage des opérateurs) après fusion des chemins pour générer la machine d'états. Cet algorithme possède une complexité inférieure à celui du *"Path-Based"*. La méthode ne produit plus les chemins les plus rapides comme le fait le *"Path-Based Scheduling"* mais diminue la complexité du contrôleur à mettre en oeuvre. Afin de limiter l'impact du partage sur les performances du circuit, le concepteur a la possibilité de restreindre l'algorithme dans ses choix et de lui indiquer l'allongement maximum autorisé pour le chemin critique (cela revient à fixer la latence maximum de l'application).

Huang dans le flot d'analyse et d'ordonnancement présenté dans [Huan93] se base sur une dérivation du graphe de contrôle modélisant l'application sous la forme d'un arbre afin de représenter chacun des chemins réalisable au sein de l'application (figure 2.11). Une fois la décomposition sous forme d'arbre réalisée, chaque branche représente un chemin mutuellement exclusif vis-à-vis de ses voisins. Lors de la construction de l'arbre, seules les structures conditionnelles sont utilisées afin de créer les chemins. Aucune analyse n'est cependant réalisée afin de trouver les exclusions mutuelles dues à ces espaces d'inéquation vides.

Wakabayashi utilise dans sa méthode [Waka89] une dérivation de la description comportementale de l'algorithme sous la forme d'un arbre de contrôle ou chaque feuille représente une structure conditionnelle. L'arbre ainsi formé constitue un ensemble de noeuds reliés par leurs relations d'inclusion. Chacun des noeuds est annoté par un vecteur de condition *(CV : Condition Vector)*. Ensuite chacune des opérations de la description initiale est marquée par le *CV* associé à la structure conditionnelle à laquelle elle appartient. Finalement, afin de chercher les opérations mutuellement exclusives, il suffit de comparer leurs *CV*. Cette méthode, comme la précédente, ne réalise qu'une analyse structurelle des branches conditionnelles afin d'extraire les exclusions mutuelles.

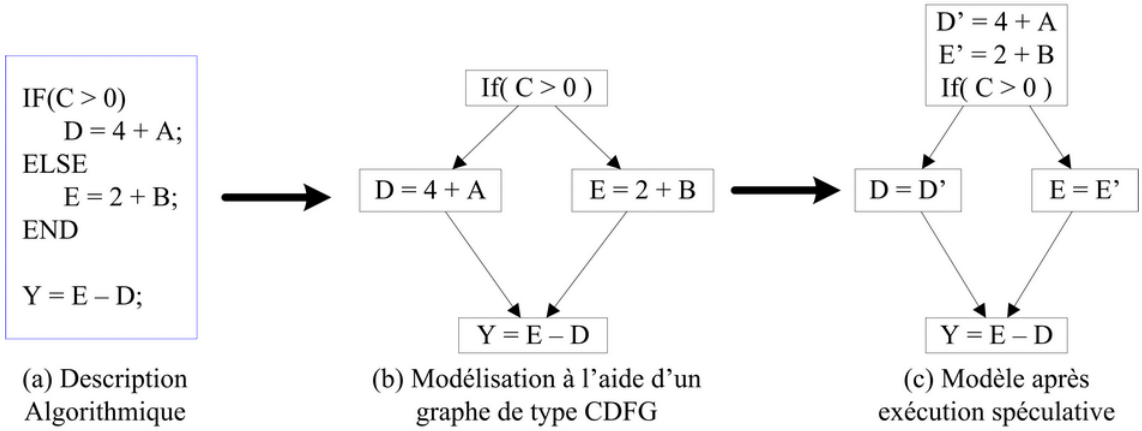

FIG. 2.12 – Transformation d'un CDFG par exécution spéculative.

2.3.2 Optimisation de la latence

Une grande partie des travaux de la communauté scientifique, dans le domaine des structures conditionnelles, s'est portée sur la réduction de la latence de ces applications lors de leur synthèse sous contrainte de ressources. Afin de réaliser cet objectif, des techniques permettant l'exécution spéculative des opérations appartenant aux branches conditionnelles ont été développées. L'intérêt de l'exécution spéculative réside dans la mise en exergue du parallélisme entre les opérations spéculées. Sans ces transformations, l'exploitation de ce parallélisme aurait été impossible à cause des barrières érigées lors de l'ordonnancement entre les opérations appartenant à différentes structures conditionnelles. Des travaux relatifs à ces techniques figurent dans [Waka92] [Holt93] [Mahl93] [Holt95]. Nous allons dans la suite de ce paragraphe nous intéresser plus particulièrement aux optimisations réalisées sur ces structures conditionnelles par l'outil de synthèse de haut niveau SPARK [Gupt03].

Optimisations préordonnancement

L'outil de synthèse SPARK [Gupt01] présenté dans les annexes (A.1) utilise dans son flot des techniques de transformation de niveau algorithmique. Une des méthodes d'optimisation vise à augmenter le parallélisme exploitable entre les différents blocs linéaires d'instructions composant l'application. L'optimisation est réalisée en regroupant les opérations conditionnées à l'extérieur de leurs structures de contrôle comme cela peut être observé dans la figure 2.12.

Dans cet exemple, les opérations comprises dans les deux branches conditionnelles de la structure *if-then-else* peuvent être exécutées en parallèle si l'architecture possède assez de ressources matérielles pour cela. Dans le cas contraire, l'exécution spéculative permet tout de même d'éviter en partie les pénalités dues aux retours d'états, car ce temps d'attente est alors employé pour réaliser les calculs qui auraient dû être réalisés après sélection de la branche conditionnelle à exécuter. Dans le cas présent, seul l'affectation du résultat temporaire vers le registre contenant la variable nécessitant une mise à jour dépend du retour d'état. La méthode augmente le parallélisme exploitable entre les opérations et tente par-là de réduire la latence de l'architecture après l'ordonnancement en sacrifiant le partage possible des

ressources matérielles entre les opérations.

La technique nommée *"Speculation Execution"*, présentée par Gupta [Gupt01] [Gupt03], fait référence à une transformation permettant de transformer des calculs conditionnés en calculs spéculatifs. Dans certains cas le déplacement des opérations peut être inversé [Gupt03b], on appelle alors cette technique la *"Reverse Speculation"*. Il est parfois intéressant de déplacer une opération qui n'appartenait pas à un bloc conditionnel à l'intérieur de ses *n* branches, en dupliquant l'opération. Ainsi, celle-ci devient dépendante conditionnellement de la branche dans laquelle elle a été placée. Cette technique permet d'accroître le parallélisme exploitable dans les structures conditionnelles lorsqu'il n'est pas opportun de réaliser des exécutions spéculatives des opérations se situant dans ces structures conditionnelles. Les structures conditionnelles sont modélisées par des blocs linéaires d'instructions (couramment nommés *basic blocs*) servant de points de synchronisation. Ces points de synchronisation sont employés par le contrôleur matériel afin de décider quelles structures conditionnelles doivent être exécutées. Le type de représentation utilisé permet de modéliser la dépendance conditionnelle entre l'opération et sa condition d'exécution. La première opération à être ordonnancée sera la condition, ensuite grâce au résultat de cette condition, on pourra sélectionner et exécuter les opérations qui doivent l'être. Ainsi le partage de ressources est possible entre les opérations mutuellement exclusives. En contrepartie, le temps d'exécution des branches est important car les opérations conditionnées ne peuvent pas être ordonnancées avant que le résultat de la condition soit connu.

Dans le cadre des approches orientées traitement intensif de données, les modèles de représentation et d'implémentation sont différents. Les modèles de représentation utilisés pour modéliser les applications contenant des structures conditionnelles sont de type DFG. Cette modélisation nécessite des transformations des structures de contrôle. Les structures conditionnelles sont donc mises à plat afin de permettre leur modélisation à l'aide de sémantiques dédiées au traitement de données [Rim92]. Ces structures mises à plat sont basées sur des opérations agissant sur l'affectation des résultats calculés (comme dans l'exécution spéculative). Lors de l'exécution, toutes les opérations sont exécutées en parallèle et l'affectation des résultats est effectuée après l'évaluation du résultat de la condition. L'avantage de cette méthode est l'augmentation du parallélisme au niveau de l'exécution des opérations. Cette méthode d'implémentation est intéressante lorsque les calculs contenus dans les branches conditionnelles ne peuvent pas être partagés (type d'opérations différent). La figure 2.13 nous représente la description comportementale d'une structure conditionnelle (fig. 2.13a) avec son équivalence une fois représentée à l'aide d'un modèle de représentation de type DFG (fig. 2.13b). La description algorithmique correspondant au modèle ainsi obtenu est présentée figure 2.13c.

Le noeud noté par un point d'interrogation dans le graphe représente un composant de sélection (implémenté par exemple à l'aide d'un multiplexeur) qui, en fonction de la valeur du résultat de $(C > 0)$, va assigner dans Z soit son ancienne valeur si la condition est fausse, soit T_2 qui est la nouvelle valeur si la condition évaluée est vraie.

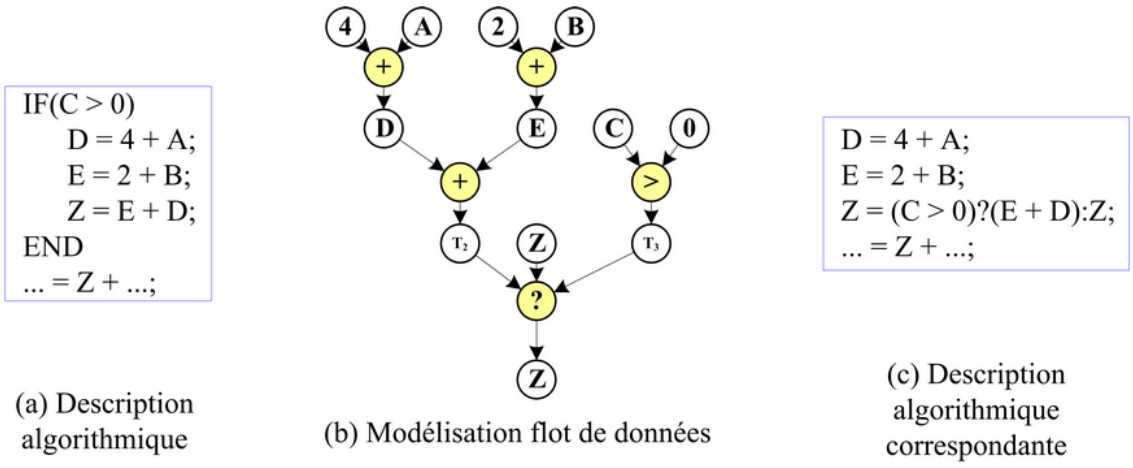

FIG. 2.13 – Modélisation des structures conditionnelles dans les flots de données.

Optimisations durant l'ordonnancement

Les transformations de graphe à l'aide d'exécutions spéculatives d'opérations conditionnées sont généralement réalisées lors d'une phase de transformation préordonnancement comme nous l'avons vu précédemment. Dans l'outil de synthèse de haut niveau *SPARK*, la phase d'ordonnancement essaye d'améliorer l'exécution spéculative des opérations. Pour une meilleure efficacité de l'exécution spéculative une autre technique nommée *"Early Condition Execution"* [Gupt03b] impose une exécution aussitôt que possible des conditions. La priorité maximum donnée aux opérations conditionnelles s'explique par l'importance du nombre de leurs successeurs ainsi que du délai nécessaire entre leur complétion et le début de l'exécution de leurs successeurs. En pratique, dans la méthode présentée, les opérations conditionnelles sont exécutées de manière préemptive avant même la validation des branches conditionnelles où ces dernières sont situées.

Lakshminarayana [Laks98] propose d'inclure lors de l'ordonnancement des techniques d'exécution spéculatives pour les opérations dépendantes d'une ou de plusieurs structures conditionnelles afin d'exploiter de manière plus efficace le parallélisme entre les opérations. La méthode présentée se compose de deux étapes : (1) le modèle de représentation est analysé et les noeuds sont marqués de manière statique par un marqueur indiquant si le noeud peut bénéficier d'une exécution spéculative, (2) lors de l'ordonnancement du modèle de représentation, en fonction des ressources matérielles disponibles, certaines opérations sont exécutées de manière spéculative afin d'augmenter le parallélisme d'exécution.

2.3.3 Optimisation des ressources entre opérations mutuellement exclusives

Les techniques d'exécutions spéculatives permettent de réduire la latence de l'application en augmentant le parallélisme exploitable entre les diverses structures conditionnelles. Cette augmentation du parallélisme s'accompagne par l'exécution d'opérations dont les résultats ne seront pas exploités par la suite en raison d'une non-validation de la condition qui régit leur exploitation. Nous allons maintenant aborder d'autres approches ciblant principalement les applications orientées flot de données utilisant des modèles

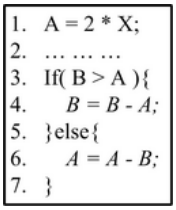

FIG. 2.14 – Opérations Mutuellement Exclusives.

de représentation où l'ensemble des opérations est mis à plat. L'objectif de ces méthodes est d'utiliser la propriété spécifiant que deux branches conditionnelles mutuellement exclusives ne peuvent s'exécuter durant un même scénario d'exécution. Cette propriété permet lors de l'assignation de placer plusieurs opérations distinctes lors d'un même cycle sur un seul opérateur. Le partage des ressources matérielles entre opérations mutuellement exclusives *(ME)* est incompatible avec les techniques d'exécution spéculative présentées dans la partie 3.2 de ce chapitre.

Détection des opérations mutuellement exclusives

La détection des opérations mutuellement exclusives au sein d'une application peut conduire à une réduction des ressources matérielles nécessaires ou de la latence. L'exploitation de l'exclusion mutuelle d'un couple d'opérations nécessite toutefois la mise en oeuvre de techniques de détection de l'exclusion mutuelle. Nous définissons un certain nombre de notions qui seront reprises par la suite :

Définition 2.3.1 (*Branches Mutuellement Exclusives*)

Dans un graphe deux branches sont dites mutuellement exclusives si dans un même scénario d'exécution de l'application, l'exécution d'une branche implique la non exécution de la seconde. Dans ce cas elles partagent une condition commune qui dans un cas vaut une valeur et dans l'autre cas son opposé.

Définition 2.3.2 (*Opérations Mutuellement Exclusives*)

Deux opérations sont dites mutuellement exclusives si et seulement si elles appartiennent chacune à des branches mutuellement exclusives.

Si nous considérons l'exemple présenté dans la figure 2.14, il est naturel à la vue de la description algorithmique que les soustractions (4) et (6) ne peuvent pas être effectuées lors d'une même exécution de l'application. Cela serait en effet contraire à la sémantique d'une structure conditionnelle *if-then-else* qui ne permet qu'à une seule branche d'être évaluée en fonction du résultat du test conditionnel (3). Dans le cas présent, nous pouvons exprimer une relation entre les opérations (4) et (6) qui est une relation de mutuelle exclusion. Cette information pourra permettre ensuite, suivant les critères d'ordonnancement et d'assignation, de placer les deux opérations de soustraction sur le même opérateur en même temps car il ne peut pas y avoir de recouvrement des deux opérations au sein d'une même exécution.

Différentes techniques ont été développées afin de permettre la détection des opérations mutuellement exclusives au sein d'une description algorithmique. Les méthodes employées se différencient par leurs

```
Algorithme d'ordonnancement

OrdonnancementASAP_ALAP( G )
ListeCluster = RegrouperLesOperationsMECompatibles( G );
Pour tous les clusters
   ClusterEligibles = EnleverLesClustersNonEligibles( ListeCluster );
   Lcp = TrierLesClusterParPriorité( ClusterEligibles );
   Pour chaque cluster appartenant à Lcp
        Opr = RechercherUnOpérateurPourAssigner( cluster );
        Si ( Opr != null ) alors
                EnleverDeLaListe( ListeCluster, cluster);
                AssignerOperations( cluster );
        Fin si
   Fin pour
   FaireAvancerLeTemps();
   LibererLesRessourcesMatérielles();
Fin pour
```

```
Algorithme d'ordonnancement

Pour tous les états nœuds du graphe G
   En = ListeNœudsEligiblesDansLeGraphe( G );
   Lnp = ClasserLesNoeudsParPriorité( En );
   Pour chaque nœuds appartenant à Lnp
        Opr = RechercherUnOpérateurPourAssigner( nœud );
        Si ( Opr != null ) alors
                ME = RechercherLesOperationsME( nœud );
                EnleverDeLaListe( Lnp, (nœud+ME) );
                AssignerOperations( nœud+ME );
        Fin si
   Fin pour
   FaireAvancerLeTemps();
   LibererLesRessourcesMatérielles();
Fin pour
```

FIG. 2.15 – Algorithme d'ordonnancement privilégiant (a) le partage des opérateurs (b) l'exploitation du parallélisme.

techniques d'analyse, leurs complexités et les résultats obtenus (taux de détection des opérations mutuellement exclusives). On trouvera en annexes quelques exemples de méthodes possibles.

Méthodes d'ordonnancement associées

Une fois les couples d'opérations mutuellement exclusives détectées dans la description comportementale, il faut mettre en oeuvre une méthode de synthèse permettant d'utiliser ces informations durant le processus d'ordonnancement du graphe, d'assignation des opérations sur les ressources matérielles et durant la génération du contrôleur. Nous allons étudier comment utiliser les informations relatives aux opérations *ME* dans les algorithmes d'ordonnancement et d'assignation.

Exploitation du partage des opérateurs - Une première approche consiste à regrouper le plus grand nombre d'opérations mutuellement exclusives sur un même opérateur. Afin d'augmenter le regroupement des opérations mutuellement exclusives, il est parfois nécessaire de ne pas ordonnancer certaines de ces opérations au plus tôt afin de pouvoir les regrouper avec d'autres n'étant pas encore éligibles. Au final on peut aboutir à un allongement du chemin critique dû au regroupement optimal des opérations mutuellement exclusives.

L'algorithme proposé dans la figure 2.15a est basé sur un ordonnancement par liste. Nous réalisons l'hypothèse que les étapes de sélection et d'allocation des ressources ont déjà été effectuées. Nous disposons d'une structure contenant l'ensemble des couples d'opérations mutuellement exclusives [Pena00]. La première étape consiste à ordonnancer le graphe à l'aide des dates au plus tôt et au plus tard pour tous les noeuds. Ces dates vont nous permettre de calculer l'intervalle de mobilité des opérations (cet intervalle sera utilisé comme métrique de priorité). Ensuite, afin de maximiser la réutilisation des opérateurs il est important de réaliser les regroupements les plus importants entre les opérations mutuellement exclusives. Pour faire cela, il est possible d'utiliser un algorithme de type *"Clique-Covering"* qui recherchera les plus grandes cliques d'opérations mutuellement exclusives au sein des opérations non ordonnancées.

Une fois que les clusters contenant $\{1 \rightarrow n\}$ opérations ont été créés, ils vont être analysés afin de déterminer lesquels sont éligibles au cycle courant. Ensuite, la liste contenant l'ensemble des clusters va être triée par une fonction de priorité. Pour chaque cluster éligible dans l'ordre des priorités, un opérateur libre sera recherché afin d'assigner le cluster d'opérations ME sur une unique ressource matérielle. Les clusters assignés sont ensuite retirés de la liste des clusters éligibles, puis lorsque qu'il n'y a plus de clusters éligibles pour le cycle courant, le temps est alors avancé d'une durée égale à un cycle d'horloge et les ressources matérielles sont libérées. La méthode réitère tant qu'il reste des clusters à ordonnancer. L'inconvénient majeur de la méthode est la complexité de la recherche des clusters d'opérations ME.

Exploitation du parallélisme entre les opérations - Une seconde approche consiste à exploiter en priorité le parallélisme entre les opérations et ensuite leur capacité à être exécutées sur un même opérateur. Ainsi les opérations sont ordonnancées et assignées lorsqu'elles sont éligibles et que la métrique de décision les désigne comme étant des opérations à ordonnancer. Lorsque la décision de les exécuter est entérinée, on cherche alors dans la liste des opérations éligibles si l'on trouve une ou des opérations ME avec elle. Si des opérations ME sont trouvées, elles sont ordonnancées et assignées sur le même opérateur. Cette méthode réduit le partage d'opérateurs par rapport à la première ; en contrepartie, elle permet d'exploiter pleinement le parallélisme entre les opérations évitant ainsi d'allonger la durée du chemin critique.

Les premières étapes de l'algorithme d'ordonnancement (figure 2.15b) sont identiques à celle présentées précédemment. Ici, les noeuds ME ne sont pas regroupés sous forme de clusters afin d'augmenter le partage des opérateurs. La liste des opérations à ordonnancer est filtrée afin de ne conserver que les opérations éligibles au cycle courant puis cette liste est classée par ordre de priorité. Pour chacune des opérations présentes dans la liste, on cherche un opérateur libre implémentant la fonction souhaitée. Si un opérateur est trouvé, on recherche alors dans la liste des opérations éligibles des opérations qui seraient ME avec l'opération élue. Si des opérations ME sont trouvées, elles sont assignées sur l'opérateur. Puis ces opérations assignées sont retirées de la liste des opérations éligibles. Le processus réitère jusqu'à épuisement de la liste des opérations à ordonnancer.

Structure des contrôleurs générés

La gestion des opérations conditionnelles peut être effectuée de différentes manières comme nous venons de le montrer. Une constante demeure tout de même dans l'ensemble de ces techniques : les architectures ciblées par toutes ces méthodes doivent disposer d'un contrôleur permettant de prendre en considération les retours d'états. Ces contrôleurs doivent, suivant les techniques mises en oeuvre durant la synthèse, décider de : l'ensemble des calculs qui doivent être exécutés ou bien, dans le cas des exécutions spéculatives, si les affectations doivent être validées ou non.

2.3.4 Bilan de ces approches

Dans cette partie nous avons présenté deux techniques orthogonales. Dans un premier temps, nous avons abordé différents travaux portant sur l'utilisation de techniques d'exécution spéculative qui exploitent

le parallélisme implicite aux applications contenant des structures conditionnelles. L'identification des opérations mutuellement exclusives permet de partager les ressources matérielles lors des phases d'ordonnancement et d'assignation. Ces différentes méthodes s'appuient souvent sur un modèle de représentation basé sur des graphes orientés. Ces modélisations limitent le parallélisme exploitable entre les différentes structures conditionnelles indépendantes. L'objectif des méthodes proposées est clairement de réduire la latence des applications sous contrainte de ressources. Dans les applications de TDSI, les contraintes imposées aux applications sont généralement formulées sous forme soit d'une contrainte de latence, soit d'une contrainte de cadence. Les travaux présentés ne prennent pas en considération la contrainte de cadence, la réduction de la latence n'étant pas nécessairement l'objectif principal. De plus l'ensemble de ces méthodes ne prend pas en considération les problèmes d'accès aux données structurées placées dans les mémoires. Cette absence de gestion des accès à la mémoire peut devenir un point bloquant par exemple lorsque l'on cible des architectures systoliques. Nous allons donc aborder dans le prochain paragraphe les méthodes permettant de gérer les accès aux mémoires durant les étapes du processus de synthèse.

2.4 Gestion des accès aux données en mémoire

Pour un grand nombre d'applications TDSI, les accès à la mémoire durant l'exécution sont caractérisés par certains paramètres tels le déterminisme des séquences d'accès, la répétition des accès, le parallélisme possible entre les accès, etc. Nous allons présenter différentes techniques de gestion de la mémoire avant, pendant et après la synthèse d'architecture afin de montrer la relation étroite existante entre la synthèse de l'unité de traitement et la synthèse des unités de mémorisation.

Dans un premier temps, nous introduisons quelques définitions utiles pour la suite du document.

Définition 2.4.1 (*Accès statique*)

Un accès à la mémoire dit statique est un accès pour lequel l'adresse de la donnée accédée peut être connue avant ou pendant le processus compilation/synthèse de la description comportementale. L'accès à la mémoire ne dépend pas du contexte d'exécution de l'application.

Définition 2.4.2 (*Accès dynamique*)

Un accès à la mémoire dit dynamique est un accès dont l'adresse de la donnée accédée est inconnue avant ou pendant le processus compilation/synthèse de la description comportementale. L'accès à la mémoire dépend du contexte d'exécution de l'application (adresse calculée en fonction des entrées ou accès conditionnel à la mémoire).

Définition 2.4.3 (*Séquence d'accès déterministe*)

Une séquence d'accès déterministe est une séquence d'accès à la mémoire où toutes les adresses des données lues ou écrites peuvent être connues avant ou pendant le processus compilation/synthèse de la description comportementale. Les séquences d'accès dites déterministes sont composées uniquement d'accès statiques.

Définition 2.4.4 (*Séquence d'accès indéterministe*)

Une séquence d'accès indéterministe est une séquence d'accès à la mémoire où toutes les adresses

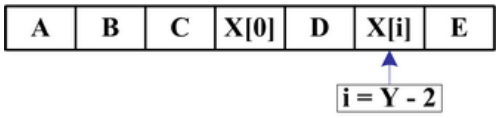

FIG. 2.16 – Séquence d'Accès Indéterministe à la mémoire.

des données lues ou écrites ne peuvent pas être résolues avant ou pendant le processus compilation/synthèse de la description comportementale. L'indéterminisme repose aussi bien sur la taille de la séquence que sur les données accédées. Les séquences d'accès dites indéterministes peuvent être composées d'accès statiques et dynamiques.

Exemple de séquences d'accès déterministe et indéterministe - Nous allons illustrer les deux définitions que nous venons d'énoncer par un exemple. Dans la figure 2.16, nous considérons une séquence d'accès en lecture à la mémoire. Si nous considérons la sous-séquence d'accès formée par la lecture des éléments $\{A, B, C, X[0], D, E\}$, on peut dire que cette dernière est déterministe car on connaît de manière formelle la longueur de la séquence et les données (leurs adresses) auxquelles on va devoir accéder. L'élément de la séquence que nous n'avons pas cité jusqu'à présent, $X[i]$, va lui rendre la séquence d'accès indéterministe. $X[i]$ est un accès dynamique si par exemple $i = Y - 2$ avec Y une entrée du système. Nous avons une connaissance a priori de la structure à laquelle on va devoir accéder (X) mais pas l'indice/adresse de l'élément, ce dernier dépendant du contexte d'exécution.

Remarque

Dans certains cas, en fonction des connaissances que l'on possède à priori sur les calculs et leurs dynamiques, il peut être possible de borner l'indice i afin de relâcher les contraintes sur les éléments qui ne peuvent mathématiquement pas être accédés.

Dans cette section dédiée à la gestion de la mémoire durant les différentes étapes de la synthèse d'architecture, nous allons d'abord aborder les optimisations mémoires présynthèse. Puis nous détaillerons les architectures utilisées pour implémenter les applications. Nous finirons cette section par les techniques de synthèse prenant en considération la mémoire afin de passer du niveau algorithmique au niveau architectural.

2.4.1 Optimisations présynthèse

Placement des données

Le placement des données dans les bancs mémoire est une phase importante de la synthèse car l'exploitation du parallélisme existant entre les accès à la mémoire (et donc des calculs) va en dépendre. En effet, le nombre de ports que possède une mémoire est restreint. Cela implique que le nombre d'accès par cycle d'horloge à une même mémoire est limité. Les applications TDSI manipulant généralement un nombre important de données, le nombre de mémoires allouées et le placement des données dans ces dernières affectent donc grandement les performances des circuits obtenus [Corr05].

Mapping des données au sein des bancs mémoire (gestion hiérarchique) - Dans [Holm94] [Holm95], Holmes et Gajski présentent une méthode d'exploration de l'espace architectural pour les architectures pouvant contenir une arborescence de mémoires. L'approche vise à explorer toutes les solutions architecturales possibles allant de la plus rapide et la plus coûteuse, à la plus lente. Ensuite le concepteur choisit quelle solution il compte conserver et exploiter. Le modèle d'estimation des performances des architectures générées pour l'exploration architecturale prend en compte : le nombre de niveaux de hiérarchie mémoire autorisé (vitesse + coût des éléments de mémorisation), la configuration des bus de données et d'adresse entre les différents éléments, la période d'horloge, une estimation des ressources matérielles nécessaires (unités fonctionnelles pour les calculs) ainsi que la granularité de la recherche entreprise [Holm94]. Au final le concepteur possède un éventail des solutions architecturales pour son application. Il lui faudra donc choisir une solution en fonction de ses critères de décision : surface, consommation, latence... D'autres méthodes du même type [Jain92] [Shar93] [Timm93] ont pour objectif de réaliser une estimation des performances a priori afin de guider le concepteur dans son raffinement. Benini [Beni01] quant à lui expose une technique permettant de partitionner de larges mémoires en un sous-ensemble de mémoires de tailles inférieures afin de diminuer l'énergie consommée par chaque accès. Dans [Seo02], l'auteur présente une méthode d'exploration de l'allocation des bancs mémoire et d'assignation des données au sein des unités de mémorisation disponibles. L'exploration des solutions est guidée par une contrainte de latence spécifiée par le concepteur. Seo [Seo03] fait évoluer sa méthode d'exploration afin de prendre en considération les répercussions de la stratégie mémoire au sein du processus de synthèse de l'architecture (particulièrement dans le processus d'ordonnancement). L'exploration peut alors utiliser dans une même solution des mémoires possédant des caractéristiques différentes (temps d'accès, vitesse). Cela augmente considérablement l'espace des solutions explorées.

Mapping des données au sein des bancs mémoires avec minimisation des conflits d'accès - Afin d'exploiter pleinement les capacités de l'architecture mémoire, il est important lors du mapping des données dans les bancs que les confits d'accès soient pris en considération. Pour cela des techniques d'ordonnancements de type *"Forced Direct Scheduling"* sont utilisées sur les séquences d'accès à la mémoire afin de lisser les accès (réduire le nombre d'accès parallèles). L'approche proposée par Wuytack [Wuyt96] [Wuyt99] vise ensuite à minimiser le nombre de conflits d'accès aux bancs mémoire de manière préventive avant l'étape de synthèse. La minimisation des conflits d'accès aux données se fait par l'intermédiaire d'un graphe de conflits qui guidera l'assignation des données au sein des bancs mémoires qui seront alloués en fonction des besoins (mémoires double ports, simple port, nombre d'entités, etc.). L'objectif est une minimisation de la bande passante nécessaire entre les mémoires et les unités de traitement sans détériorer les performances temporelles (augmentation de la latence). Afin d'optimiser efficacement la sélection des unités de mémorisation, un ordonnancement des accès à la mémoire est réalisé avant la synthèse du chemin de données sous contrainte d'un nombre de cycles d'horloge maximum.

Génération d'architectures mémoires (GAUM) - D. Chillet [Chil97] a proposé une approche équivalente de conception d'unité de mémorisation (sélection, allocation, mapping) après synthèse du chemin de données. Le travail s'effectue cependant en aval de la synthèse de l'unité de traitement et les données

manipulées sont des scalaires. Ce travail s'inscrivait dans le projet GAUT qui avait à l'origine comme objectif de fournir une description architecturale de l'unité de traitement en priorité. L'ensemble des informations fournies par la synthèse de l'unité de traitement propageait alors ses contraintes vers la synthèse des autres unités fonctionnelles. Dans ce contexte, l'auteur propose une méthode et un outil s'insérant en aval de la synthèse de l'unité de traitement. La conception de l'unité de mémorisation est basée sur trois étapes : la distribution des données dans les bancs mémoire, le placement des données dans chaque banc mémoire et la génération des adresses. Les solutions retenues tentent de minimiser le coût de la solution architecturale de l'unité de mémorisation en fusionnant les générateurs d'adresses. Les générateurs d'adresses et les mémoires sont décrits au niveau structurel.

Conclusion - Nous venons de présenter un ensemble de techniques permettant de répartir les données au sein de bancs mémoires avant synthèse afin de minimiser certains critères (surface, consommation, latence ...). Malheureusement ces techniques ne peuvent placer l'ensemble des éléments d'un même vecteur/tableau qu'au sein d'un seul et unique banc mémoire ce qui limite drastiquement le nombre d'accès simultanés possibles aux éléments des structures. Par ailleurs les techniques d'optimisations associées ne sont applicables que dans le cadre d'accès statiques à la mémoire.

Optimisation des accès dynamiques à la mémoire

Les accès dynamiques à la mémoire sont plus complexes à gérer que les accès statiques durant les étapes de synthèse : dans un premier temps, il est nécessaire de calculer l'adresse de la donnée à laquelle on souhaite accéder et, dans un second temps, il faut réaliser l'accès sans générer de conflits avec des accès concurrents.

Afin de réduire la complexité des calculs d'adresses dynamiques, diverses approches [Kita91] [Lipp91] proposent l'ajout d'unités arithmétiques dédiées au sein des processeurs. Ces unités arithmétiques sont associées aux bancs mémoires dans lesquels les données dynamiquement adressées se situent. Ces approches sont adaptées aux applications composées de nids de boucles car il est nécessaire de posséder autant d'unités arithmétiques dédiées qu'il n'existe de manière d'accéder dynamiquement aux données dans l'application.

Une autre approche proposée par Gupta dans [Gupt00] vise la transformation de la description comportementale de l'application à synthétiser. Dans le cas des applications multimédias cela permet de simplifier les calculs d'adresses dynamiques et donc leurs implémentations à l'aide d'opérateurs dédiés. Les techniques d'optimisation des expressions sont indépendantes de la cible sur laquelle on souhaite implémenter l'algorithme (replacement des invariants de boucle, analyse des variables d'induction et des transformations algébriques). Les techniques les plus efficaces sont l'élimination des sous expressions communes *(Common Sub-expression Elimination)* qui visent à remplacer des calculs identiques effectués plusieurs fois par une variable mémorisant le résultat, ainsi que les techniques de propagation de constantes [Aho86] [Gupt02]. L'autre étape importante de la méthode est la réduction de la complexité des opérateurs. Cela est effectué en remplaçant par exemple certaines multiplications par des décalages et additions, des divisions par des multiplications inversées, etc. Il est à noter que ces transformations ont pour effet d'augmenter l'utilisation de données temporaires.

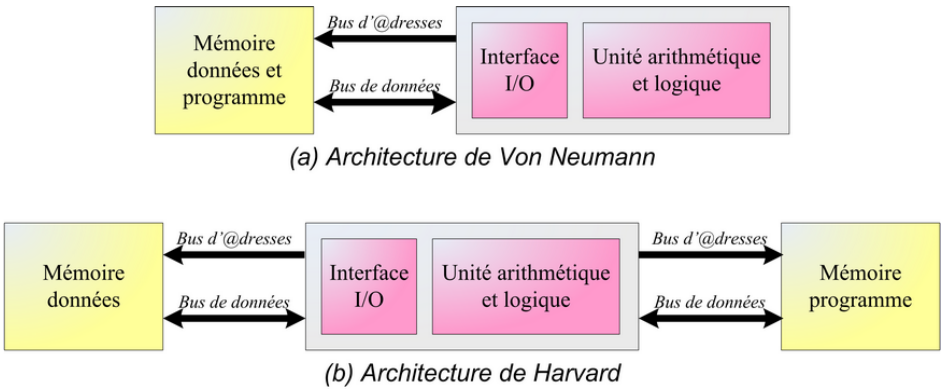

FIG. 2.17 – Architectures typiques des processeurs avec accès direct à la mémoire.

Si ces méthodes permettent de gérer les accès dynamiques à la mémoire, en réduisant la complexité des calculs d'adresses dynamiques, la présence d'accès dynamiques à la mémoire au sein d'une application empêche les processus de synthèse d'exploiter pleinement le parallélisme entre les différents accès à la mémoire (conflits d'accès possibles). Nous allons maintenant aborder les architectures mémoires générées pour implémenter des applications possédant des séquences d'accès déterministes et indéterministes.

2.4.2 Architectures générées par la synthèse

Les architectures ciblées par les outils de synthèse d'architecture peuvent varier en fonction des domaines d'applications ciblés. Suivant la nature de ces applications les solutions apportées à la gestion des accès mémoire varient.

Séquences d'accès indéterministes

Les accès directs à la mémoire afin d'accéder à des données sont souvent employés par les applications dominées par le contrôle. Dans ces applications, le calcul des adresses des données auxquelles on désire accéder se fait lors de l'exécution de l'application. A ce moment là, le chemin de données (où une des unités spécialisées dans le calcul d'adresses) va présenter l'adresse à la mémoire qui va décoder le numéro de ligne et de colonne avant de répondre à la requête en fournissant la donnée souhaitée (figure 2.17a - 2.17b). Le modèle architectural des processeurs généralistes fait souvent référence pour ce type d'accès.

Dans les applications nécessitant d'accéder à de grandes quantités de données, il est généralement nécessaire d'adjoindre aux circuits des mémoires externes qui sont généralement d'autant plus lentes que leur taille est grande. Dans ce cas, l'accès aux données est généralement réalisé par l'intermédiaire d'une architecture hiérarchique composée de caches et de *scratchpads* comme cela est indiqué dans la figure 2.18. Les mémoires rapides de taille plus faible sont contrôlées grâce à des composants spécialisés (DMA, contrôleur).

Afin de précharger les structures de données nécessaires aux calculs, des techniques ont été proposées

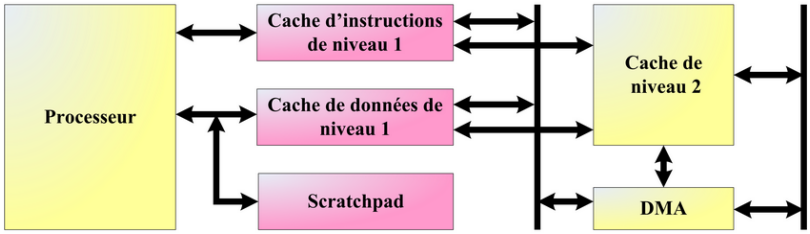

FIG. 2.18 – Architecture avec accès indirect à la mémoire.

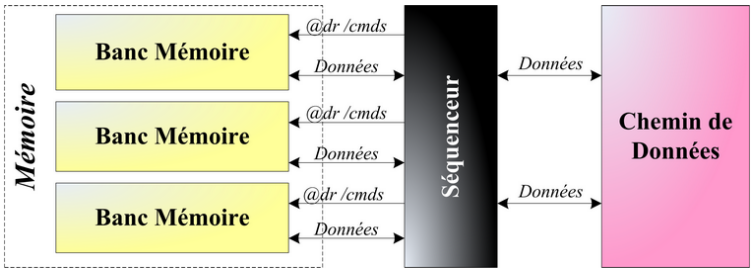

FIG. 2.19 – Exemple d'architecture à base de séquenceur mémoire.

afin d'insérer, parmi les instructions qui forment le traitement à effectuer, des opérations permettant de dialoguer avec les contrôleurs des différentes mémoires. Cela permet de réduire considérablement la latence des applications ainsi optimisées. Ces travaux ont été étendus dans [Cart99] [Huan01] avec la possibilité de précharger directement des scalaires en lieu et place des structures grâce à un mécanisme de tables de translation des adresses logiques vers les adresses physiques. Malgré les gains en performances apportés, ces méthodes nécessitent de la part du concepteur des connaissances sur les structures de données manipulées et sur le fonctionnement du contrôleur mémoire disponible dans l'architecture ciblée.

La gestion des accès à la mémoire dans les architectures à base de processeurs généralistes ou dédiés est une tâche complexe car l'optimisation des accès nécessite, d'une part, du déterminisme au niveau de l'application traitée, et d'autre part, une connaissance de l'architecture cible et des capacités du contrôleur mémoire. Nous allons maintenant nous intéresser aux architectures mémoire générées lorsque les séquences d'accès sont déterministes.

Séquences d'accès déterministes

Dans les applications dominées par le traitement de données où les séquences d'accès sont déterministes a priori, la gestion des accès à la mémoire est généralement indirecte. On utilise pour cela des séquenceurs d'accès à la mémoire [Chil97] [Corr05] comme cela est décrit dans la figure 2.19.

Le séquenceur a pour rôle de fournir au chemin de données les données qui lui sont nécessaires pour la réalisation des calculs. De plus, il doit récupérer les données produites afin de les mémoriser dans les bancs mémoire. Ce type d'architecture nécessite de connaître a priori les séquences d'accès à la mémoire (phase d'analyse ou synthèse de l'application). L'architecture interne du séquenceur varie en fonction

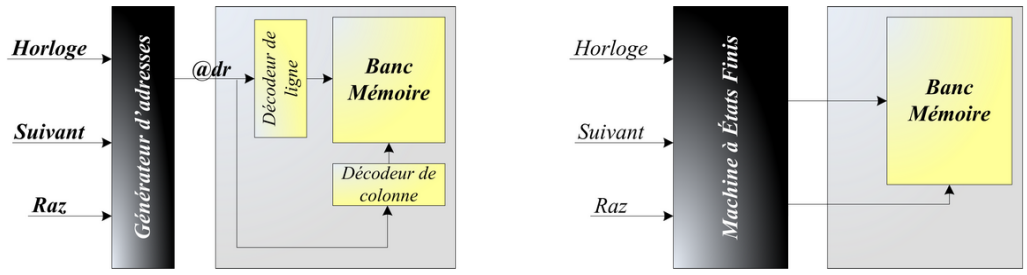

FIG. 2.20 – Séquenceurs mémoire : (a) à base de compteur (b) à base de machine à état finis.

des besoins. Généralement, les séquenceurs d'accès à la mémoire sont soit composés d'un générateur d'adresses basé sur un compteur (figure 2.20a), soit d'une machine à états finis (figure 2.20b).

Ces architectures de base peuvent être optimisées afin de respecter des contraintes système. Koegst [Koeg96] présente des techniques d'optimisation pour la conception de machines d'états. La méthode proposée est basée sur la technique de *"clock gating"* sur les entrées de la machine d'états évitant les commutations inutiles. A cela l'auteur ajoute des techniques de codage des états impliquant de faibles commutations lors des transitions, réduisant ainsi la consommation en énergie. Des détails supplémentaires sur les techniques d'optimisation des machines d'états après synthèse peuvent être trouvés dans [Mont98]. Une autre architecture proposée par Moon [Moon02] permet dans les applications TDSI de réduire la consommation en énergie de l'ensemble. L'architecture employée pour le séquenceur d'accès mémoire utilise des séquenceurs de ligne à la place des traditionnels décodeurs d'adresses afin d'optimiser la latence tout en respectant la contrainte de faible consommation. Il prouve, par un jeu de mesures, que la consommation d'énergie est indépendante de la taille de l'espace mémoire piloté par son séquenceur.

Dans [Hett02a], l'auteur présente un nouveau modèle d'implémentation pour les générateurs d'adresses des bancs mémoire. L'idée est de découpler le générateur d'adresses des bancs mémoire à proprement parler. Une telle séparation permet ainsi d'optimiser la génération du générateur d'adresses de manière indépendante des bancs mémoire. L'implémentation du générateur d'adresse est basée sur l'utilisation de registres à décalage à la place de la traditionnelle machine à états finis (figure 2.21), cela leur permet d'obtenir des temps de propagation plus faibles au sein du générateur d'adresses. Les latences observées au sein de ce nouveau générateur d'adresses ont aussi l'avantage d'être quasiment constantes et cela peu importe la taille de la séquence d'accès.

Park et Diniz présentent dans [Dini00] [Park01], une architecture de contrôleur mémoire permettant de faire abstraction des caractéristiques des mémoires lors de la synthèse d'architecture (figure 2.22). Ils partent du constat que les outils de synthèse d'architecture commerciaux ont un support inefficace des interfaces mémoires, rendant très complexe l'utilisation des opérations de lecture ou d'écriture pipelines implémentées dans les mémoires actuelles. Afin de résoudre ce problème, ils s'attachent à générer une architecture de contrôleur mémoire générique permettant de piloter les accès aux mémoires externes en utilisant ou non les modes pipelines. Ce séquenceur mémoire est construit à partir des informations délivrées par l'outil de synthèse d'architecture *DEFACTO* [Bond99]. Les capacités du séquenceur sont étendues et permettent d'anticiper les lectures au sein de l'unité de mémorisation (ce qui implique une

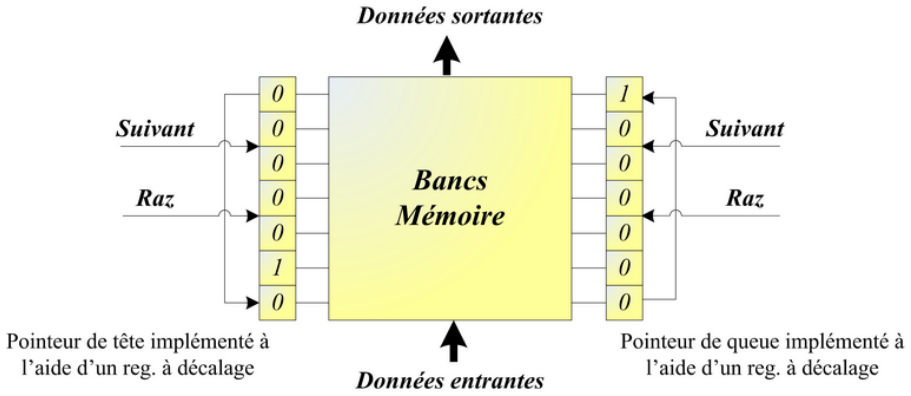

FIG. 2.21 – Architecture mémoire à base de registres à décalage employée afin de réduire la latence.

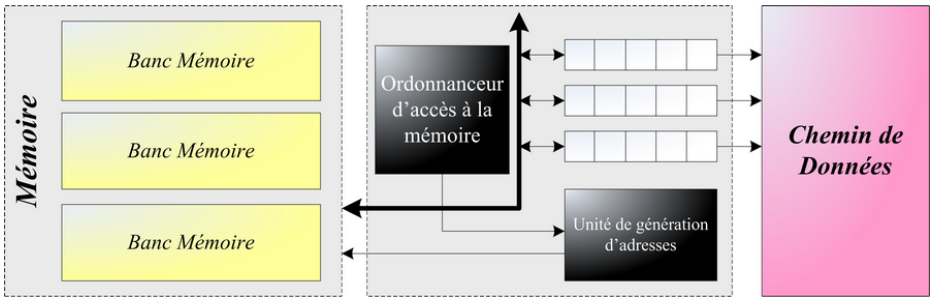

FIG. 2.22 – Séquenceur servant d'interface modulaire entre la mémoire et le chemin de données.

limitation de la bande passante nécessaire).

Architecture du séquenceur de l'outil GAUT - Les travaux de G. Corre [Corr05] au sein de l'outil GAUT ont permis de mettre en oeuvre une architecture générique de séquenceur mémoire permettant dans un premier temps de contraindre le processus de synthèse et dans un second temps d'assurer la production/consommation de données imposées par l'unité de traitement. La figure 2.23 représente l'architecture générique du séquenceur mémoire tel qu'il est actuellement défini pour l'outil GAUT. Cette architecture basée sur la définition d'une adresse logique et d'une adresse physique permet à cette architecture de gérer une distribution des données non contiguë (multi-bancs). La distribution non contiguë des données permet d'exploiter, durant les phases d'ordonnancement, le parallélisme d'accès entre les diverses données ainsi placées, ce qui est impossible dans le cas d'une allocation mono-banc.

L'architecture permet une gestion du vieillissement interne au séquenceur mémoire (avec une translation des adresses) grâce à une adresse logique pointant sur une case mémoire contenant l'adresse physique de la variable à accéder (indirection de l'adresse). Le séquenceur permet la gestion des applications contenant des séquences d'accès déterministes. Dans le cas de séquences indéterministes dues à des transferts conditionnels, une transformation visant à les rendre inconditionnels est appliquée (contrainte imposée par l'unité de traitement cf. présentation de l'outil GAUT dans le paragraphe 4.2). Cela permet d'obtenir une séquence d'accès parfaitement linéaire simplifiant l'architecture du contrôleur. L'ensemble

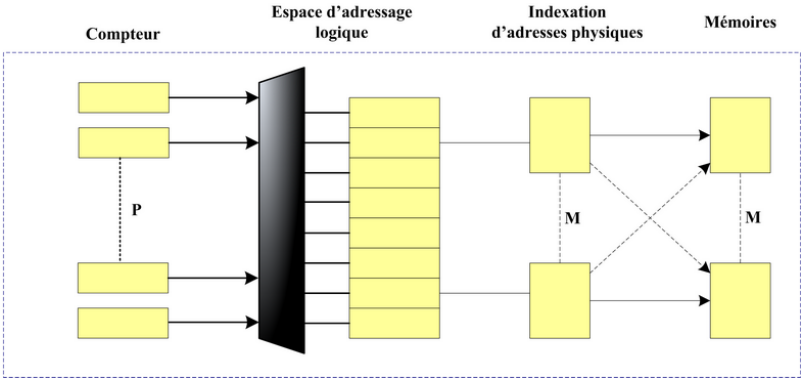

FIG. 2.23 – Architecture générique du séquenceur de l'outil GAUT.

des données *potentiellement* utiles est donc transféré.

Conclusion - Nous venons de présenter dans cette partie un certain nombre de travaux dédiés aux séquenceurs d'accès. Les séquenceurs permettent de dissocier la synthèse du chemin de données de la synthèse de la mémoire. Ils sont utilisables dans le cas d'applications où les séquences d'accès sont déterministes. Le contrôleur du séquenceur est alors implémenté sous la forme d'une machine à états finis ou d'un compteur. Lorsque l'application contient un minimum d'indéterminisme (accès dynamiques), les solutions à base de séquenceurs ne sont plus utilisables, il faut alors se tourner vers des architectures basées sur l'utilisation de caches/scratchpads.

2.4.3 Techniques d'optimisations durant la synthèse

Prise en compte des temps d'accès à la mémoire - Panda [Pand98] part du constat que les outils de synthèse de haut niveau ne prennent pas en considération les modes d'accès disponibles dans les RAMs externes présentes sur le marché. Il est par exemple possible de réaliser des lectures pipelines ainsi qu'un grand nombre d'accès optimisé permettant des gains de 40% par rapport à des *RAM* traditionnelles. Pour pouvoir tirer parti des mémoires actuelles, son modèle de représentation $CDFG$ incorpore des noeuds d'accès à la mémoire qui sont traités par la suite comme étant des opérations multi-cycles. Cela permet lors de l'ordonnancement de considérer de manière fine toutes les possibilités de lecture pipeline. L'architecture ciblée n'exploite pas les capacités des séquenceurs car les accès peuvent être indéterministes. De plus, l'ensemble des données appartenant à une même structure sont assignées dans un même banc mémoire, le parallélisme d'accès à ces données n'est pas optimum.

Optimisation de la consommation au niveau des transferts de données - Un grand nombre de recherches a été mené afin de réduire la consommation d'énergie induite par la partie mémorisation. Une grande partie de l'énergie dissipée dans les circuits intégrés est imputable aux commutations sur les interconnections entre les composants (par exemple les bus entre les mémoires et les unités de traitement). Afin de limiter ces phénomènes pénalisant fortement l'autonomie des systèmes embarqués, des travaux

ont été effectués [Dasg95] [Dasg98]. Les techniques d'ordonnancement et d'assignation proposées minimisent les transitions sur les bus de données reliant les unités de calculs aux mémoires. Malheureusement les algorithmes mis en oeuvre possèdent en général une forte complexité calculatoire (NP Complet). Cela limite l'application de telles techniques aux designs de taille modeste. Hong [Hong00] propose une heuristique permettant d'assigner des bus aux données à transférer tout en minimisant les transitions (l'assignation et l'ordonnancement sont réalisés conjointement). D'autres travaux visant les même objectifs ont été proposés plus récemment par Lyut [Lyuh03] [Lyuh04]. L'ensemble de ces méthodes est néanmoins contraint par la connaissance a priori des valeurs des données avant synthèse, tout comme les travaux connexes visant à limiter le taux de commutation à l'entrée des opérateurs de calculs [Gail98].

Gestion préemptives des accès à la mémoire (kanban) - En association avec les travaux présentés précédemment dans le domaine des séquenceurs d'accès à la mémoire, G. Corre [Corr05] a mis au point une technique d'ordonnancement anticipant et/ou retardant les accès aux bancs mémoire afin de lisser les accès durant l'exécution. Cette méthode permet d'optimiser l'utilisation du parallélisme d'accès aux données présentes en mémoire et augmente le parallélisme réellement exploitable au sein du chemin de données. La méthode gère l'accès aux données en mémoire de manière équivalente à la méthode kanban utilisée en production. Cette technique est actuellement utilisée dans le cadre de séquences d'accès déterministes dans l'outil de synthèse GAUT.

2.4.4 Bilan de ces approches

Nous venons de présenter dans cette partie un ensemble de techniques et de méthodologies permettant de gérer les accès aux données en mémoire avant/pendant/après la synthèse d'architecture. Ces méthodologies permettent de réaliser le placement des données en mémoire et l'allocation des bancs nécessaires avant et après synthèse de l'unité de traitement. L'utilisation de séquenceurs mémoire permet de décorréler la synthèse des parties mémoire/traitements et d'optimiser les transferts entre ces deux unités de manière à réduire la consommation globale du circuit. D'une manière générale, l'ensemble de ces méthodes permettant d'aboutir à des implémentations efficaces n'est toutefois applicable que dans le cadre de séquences d'accès déterministes. Pour les applications TDSI contenant des structures conditionnelles, des structures itératives et des accès indéterministes, ces méthodes ne sont donc malheureusement pas utilisables.

2.5 Conclusion

Dans ce chapitre nous avons présenté, dans un premier temps, différents travaux portant sur l'optimisation et la synthèse des structures itératives. Les différentes méthodologies s'appuient souvent sur une modélisation des boucles basée sur les graphes CDFG. Les transformations à appliquer sur de tels graphes afin de permettre l'exploitation parallèle des structures itératives indépendantes sont complexes. Pour cette raison, la majorité des flots de synthèse gérant les boucles se contente d'utiliser des techniques de *software pipelining* afin de réduire la latence globale de l'application en sous-exploitant le parallélisme

intrinsèque entre les structures conditionnelles (boucles et branches conditionnelles).

Dans un second temps, nous avons abordé la gestion et l'optimisation des structures conditionnelles dans les flots de synthèse. Deux approches orthogonales existent : (1) Minimiser la latence de l'application en exécutant de manière spéculative les opérations contenues dans les structures conditionnelles et en ne conservant de conditionnel que l'affection des résultats. (2) Réduire le nombre d'opérations à exécuter en employant des techniques réservées aux opérations mutuellement exclusives. Ces deux méthodes permettent d'optimiser à leur manière la synthèse de l'unité de traitement, mais ne gèrent pas le problème connexe d'accès à la mémoire (parallélisme d'accès pour l'exécution spéculative, accès conditionnels pour les opérations mutuellement exclusives).

Finalement, nous avons abordé, dans un troisième temps, différentes techniques dédiées à la gestion des accès aux données en mémoire lors de la synthèse. Le type d'application à synthétiser influe directement sur les techniques exploitables ainsi que sur les architectures d'implémentation. Les techniques d'optimisation peuvent être appliquées avant ou après synthèse. Une partie de ces techniques n'est utilisable que dans le cas bien particulier des applications déterministes. Dans ce cas, l'utilisation de séquenceurs d'accès aux bancs mémoire est courante. Cela permet de réduire la consommation globale (induite par les transferts de données) ainsi que la surface (lissage des accès mémoire et donc réduction des besoins en nombre de bus). Le point limitatif dans l'utilisation de ces méthodologies vient de la nécessité d'avoir des séquences de transferts déterministes a priori.

Dans les applications de TDSI que nous visons, bien que la majorité des calculs et séquences d'accès soit régulière et déterministe, il est important de pouvoir prendre en considération les traitements partiellement indéterministes. Cette prise en charge ne doit cependant pas être réalisée au détriment de la gestion du parallélisme des calculs/accès mémoires possibles pour les applications déterministes.

Nous avons choisi dans nos travaux de nous concentrer sur la conception adéquate des composants en vue de leur intégration. Nous proposons une approche dans laquelle les applications partiellement indéterministes sont prises en charge de manière équivalente aux applications déterministes. Cette approche s'applique aux applications TDSI fortement orientées flot de données où des structures conditionnelles ont été insérées afin de réduire leur complexité calculatoire (exemple : détection de mouvement, compression audio, ...).

Aussi, le chapitre suivant présente une modélisation formelle originale permettant la spécification des applications TDSI dominées par les traitements et contenant de l'indéterminisme.

Chapitre 3

Modèle de représentation

3.1 Introduction

Différentes familles de modèles de représentation existent ; nous pouvons citer le modèle *CFG (Control Flow Graph)* [Aho86] employé dans le cadre des applications dominées par le contrôle et le modèle *DFG (Data Flow Graph)* [Gajs92] utilisé, pour sa part, dans les applications dominées par les traitements de données. De l'union de ces deux modèles est apparu le modèle *CDFG (Control-Data Flow Graph)* [Gajs92] qui permet la modélisation des sémantiques de contrôle tout en assurant une exploitation du parallélisme entre les opérations non dépendantes. Les modèles DFG et CDFG sont employés dans bon nombre d'outils de synthèse d'architecture. Malheureusement, ces deux modèles possèdent des restrictions qui contraignent les techniques d'optimisation et de synthèse. Le *DFG* ne permet pas la modélisation des sémantiques de contrôle autrement que par une transformation qui vise à exécuter toutes les opérations conditionnées de manière spéculative. Le *CDFG* quant à lui limite l'exploitation du parallélisme aux structures conditionnelles (nommées *basic-blocs* dans la littérature) où se situent les opérations. Un nombre important de transformations vise à modifier ce comportement mais celles-ci sont relativement complexes à mettre en oeuvre et spécifiques. La règle d'exécution séquentielle des structures conditionnelles comprises dans le modèle *CDFG* limite l'exploitation du parallélisme global de l'application. De plus, le style d'écriture du concepteur va impacter sur la représentation.

Afin de remédier à ces problèmes, d'autres modèles de représentation ont été proposés. Chaiyakul propose en 1992 le modèle *ADD (Assignement Decision Diagram)* [Chai92a] [Chai92b]. Ce modèle permet de supprimer l'impact du style d'écriture de la description algorithmique sur la modélisation. Les structures conditionnelles sont représentées de manière équivalente à ce qui a été proposé pour étendre les sémantiques des *DFG* (mise en parallèle des branches à exécuter). Cela a pour effet de mettre à plat les différents calculs à effectuer dans chacune des branches conditionnelles (exécution spéculative puis choix du résultat) [Chai93]. Les transformations appliquées empêchent toutefois l'utilisation d'un certain nombre de techniques d'optimisation/synthèse à cause de la forte corrélation entre le modèle et l'architecture ciblée. Une extension de ce modèle est développée par Juan [Juan94]. Ce modèle nommé *GC (Condition Graph)*, a pour objectif premier de représenter les conditions d'exécution des opérations afin de permettre le partage d'opérateurs entre opérations mutuellement exclusives. Une modélisation dé-

diée aux applications dominées par le contrôle sous forme d'arbre a été également présentée par Huang [Huan93].

En fonction des besoins en terme de modélisation/optimisation, d'autres modèles ont également été proposés afin de modéliser les structures conditionnelles de type *if-then-else* ou *switch-case*. On peut citer le modèle de représentation nommé *HCDG (Hierarchical Conditional Dependence Graph)* développé par Kuchcinski [Kuch01]. Ce dernier permet la modélisation des applications écrites dans le langage *SIGNAL*. Ce graphe orienté flot de données permet de modéliser les opérations et leurs conditions d'exécution à l'aide de deux niveaux de hiérarchie. Toutefois la problématique des structures itératives et des accès aux données présentes en mémoire (scalaires ou tableaux) n'est pas abordée.

A coté des graphes définis pour la modélisation des descriptions fonctionnelles, de nombreux autres modèles dédiés sont apparus pour modéliser de manière spécifique les séquences d'accès à la mémoire, le placement des données, les accès aux E/S, etc. Les problèmes de modélisation des accès à la mémoire sont traités en outre par les travaux de G. Corre avec l'utilisation de graphes de contraintes mémoire *MCG (Memory Constraint Graph)* et de conflits d'accès "rotatif" noté *GCA (Graphe de Conflits d'Accès)* [Corr05]. Ces derniers ne prennent cependant pas en considération les conditions d'accès ni la dynamicité de ces accès. Des travaux connexes sont présentés dans la littérature des compilateurs pour *DSP* où l'utilisation de graphe de compatibilité/d'accès est opérée afin de placer les données en mémoire et réduire le coût des calculs/chargements d'adresses pour accéder aux données [Wess97].

Une description plus complète des différents modèles pourra être trouvée dans les annexes (section A2).

Le modèle de représentation que nous allons définir dans ce chapitre repose sur le constat que les modèles de représentation utilisés dans les techniques d'optimisations des boucles, des structures conditionnelles et des accès mémoires diffèrent. Nous proposons donc un modèle de représentation unifié pour la spécification algorithmique qui permet la modélisation d'applications complexes contenant ces différentes primitives.

3.2 Définition du graphe

La spécification d'entrée de la synthèse de haut niveau est une description fonctionnelle ou algorithmique d'une application. Relativement à un cadre sémantique de cette spécification algorithmique, il est possible d'extraire un graphe flot de signaux ou Signal Flow Graph (SFG). Le graphe flot de signaux fait apparaître le parallélisme à grain fin des traitements (qui feront l'objet d'un ordonnancement lors de la synthèse), et le retard "applicatif" (opérateur Z^{-1} en automatique et TDSI) qui porte en particulier sur le vieillissement des signaux de la spécification.

Définition 3.2.1 (*Graphe flot de signaux - GFS*)

Un graphe flot de signaux est un graphe polaire orienté bi-partite $GFS(V,E)$ où l'ensemble fini des noeuds $V = \{v_0,\ldots,v_n\}$ représente les données et les opérations. Les noeuds v_0 et v_n sont respectivement le noeud source et le noeud puits. L'ensemble d'arcs $E \subseteq V \times V$ est un ensemble d'arcs reliant les noeuds qui représentent des dépendances de données existant entre les noeuds du graphe. L'ensemble des noeuds du graphe noté V est composé des noeuds données et opérations tel que

$V = V_{Donnees} \cup V_{Operations}$.

Nous allons étendre le modèle de représentation GFS afin de pouvoir modéliser les contraintes liées à l'utilisation de structures conditionnelles (branches conditionnelles, boucles non bornées, ...), ainsi que les accès non déterministes, sous la forme d'un graphe de dépendance de données. Pour cela nous allons définir un modèle *Control and Structure Flow Graph (CSFG)* qui dérivera de la description algorithmique.

Définition 3.2.2 (*Graphe de contrôle et de structure (CSFG)*)

Un graphe flot de contrôle et de structure est un graphe polaire orienté $CSFG(V,E)$ où l'ensemble des noeuds $V = \{v_0, \ldots, v_n\}$ représente les données et les opérations, v_0 et v_n étant respectivement le noeud source et le noeud puits. L'ensemble d'arcs $E = \{(v_i, v_j)\}$ représente les dépendances entre les noeuds. Le graphe contient $|V| = n+1$ noeuds. Un noeud représente soit une des opérations (opération arithmétique, logique ou délai), soit une donnée. Le graphe est bi-partite, un noeud de type opération sera toujours précédé et suivi par un noeud de type donnée. Un arc $e_{i,j} = (v_i, v_j)$ représente une dépendance de données directe de $v_j \leftarrow v_i$ entre les opérations v_i et v_j telle que pour toute itération du $CSFG$, l'opération v_i doit démarrer son exécution avant celle de v_j. L'ensemble des noeuds, noté V, peut être défini comme $V = V_{Operations} \cup V_{Donnees}$. Par la suite nous détaillerons l'ensemble $V_{Donnees}$ qui est une composition d'ensemble $V_{Donnees} = V_{Variables} \cup V_{Structures} \cup V_{Var-Cond}$. L'ensemble des opérations $V_{Operations}$ dont nous détaillons les composantes dans la suite de ce chapitre est composé de la manière suivante : $V_{Operations} = V_{Opr} \cup V_{Opr-Cond} \cup V_{Hierarchiques}$.

Dans notre modèle de représentation nommé CSFG nous allons définir des modélisations adaptées à la représentation des primitives algorithmiques suivantes :
- Les structures conditionnelles (*If-Then-Else*, *Switch-Case*),
- La modélisation des structures de données et des accès à la mémoire,
- Les structures itératives (boucles roulées avec ou sans borne maximum),
- La gestion de la hiérarchie au sein du graphe,

3.2.1 Définition générique d'un noeud

Sémantique

Les noeuds présents dans le graphe *CSFG* représentent l'ensemble des variables et des opérations nécessaires à la représentation des descriptions algorithmiques que l'on désire modéliser. L'ensemble des noeuds, noté V, peut être décomposé en deux sous-ensembles distincts : $V_{operations}$ représente l'ensemble des noeuds de type opération et $V_{donnees}$ l'ensemble des noeuds de type donnée contenus dans le graphe. Ces deux ensembles respectent les propriétés énoncées par les équations (3.1) et (3.2).

$$V = V_{operations} \cup V_{donnees} \tag{3.1}$$

$$V_{operations} \cap V_{donnees} = \emptyset \tag{3.2}$$

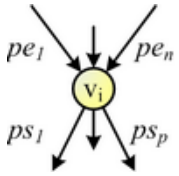

FIG. 3.1 – Représentation d'un noeud possédant plusieurs ports d'entrée et de sortie.

Syntaxe

Un noeud est un élément atomique du graphe (figure 3.1), il est composé d'un ensemble de ports d'entrée et de ports de sortie qui lui permettent d'interagir avec les autres noeuds contenus dans le graphe. Un noeud noté $v_i \in V$ peut posséder $\{1 \rightarrow n\}$ port(s) d'entrée et $\{1 \rightarrow p\}$ port(s) de sortie. L'ensemble des ports d'entrée du noeud v_i est noté $In(v_i) = \{pe_0, pe_1, \ldots, pe_n\}$. Il comprend l'ensemble des ports d'entrée où pe_j représente le j$^{\text{ième}}$ port d'entrée du noeud v_i avec $1 \leq j \leq n$. L'ensemble des ports de sortie du noeud v_i est noté $Out(v_i) = \{ps_0, ps_1, \ldots, ps_p\}$. Il comprend l'ensemble des ports de sortie dont dispose le noeud v_i où ps_k représente le k$^{\text{ième}}$ port de sortie avec $1 \leq k \leq p$. L'ensemble des ports d'entrée et des ports de sorties d'un noeud v_i est noté $InOut(v_i)$ et est défini par $InOut(v_i) = In(v_i) \cup Out(v_i)$.

Temps d'exécution d'un noeud

Chaque noeud du graphe est pondéré par un temps d'exécution noté $\lambda(v_i)$ (figure 3.4). Ce temps correspond au temps maximum entre la validation (consommation) de l'ensemble des ports d'entrée du noeud et la validation (production) du dernier port de sortie. Le modèle d'exécution externe aux noeuds est localement synchrone.

Le temps d'exécution relatif nécessaire à la validation d'un port de sortie ps_j (production d'une valeur) en fonction des différents ports d'entrée du noeud peut être différent de la durée nécessaire pour produire l'ensemble des autres sorties du noeud v_i. Cette durée notée $\lambda(pe_i, ps_j)$ exprime le temps nécessaire à la validation du port de sortie ps_j de manière relative à la validation du port d'entrée pe_i. Le modèle d'exécution interne aux noeuds est localement asynchrone. Le temps relatif entre la validation de ps_j appartenant à v_i à partir du moment ou l'ensemble des entrées de v_i ont été validées s'écrit comme présenté dans l'équation 3.3. Le temps d'exécution d'un noeud v_i équivaut au maximum de l'ensemble des durées des chemins qui le compose (équation 3.4).

$$\lambda(In(v_i) \rightarrow ps_j) = \max_{r \in In(v_i)} (\lambda(r \rightarrow ps_j)) \tag{3.3}$$

$$\lambda(v_i) = \max_{r \in Out(v_i)} (\lambda(In(v_i) \rightarrow r)) \tag{3.4}$$

En résumé, le temps d'exécution absolu maximum du noeud v_i, noté $\lambda(v_i)$, modélise la latence du chemin critique contenu dans le noeud v_i.

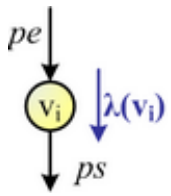

FIG. 3.2 – Modélisation du temps d'exécution d'un noeud.

Offsets de consommation et de production

Les ports d'entrée et les ports de sortie du noeuds v_i sont pondérés par des offsets temporels. Ces offsets représentent des délais (positifs ou négatifs) exprimant les retards temporels relatifs entre la date de consommation/production d'une entrée/sortie et le début/fin d'exécution du noeud auquel les ports appartiennent. Nous distinguons deux catégories distinctes d'offsets :

- *Les offsets de consommation :* les offsets de consommation modélisent les temps relatifs associés aux ports d'entrée des noeuds du graphe. Ces offsets représentent la consommation tardive de la valeur portée par un port d'entrée pe_j au plus tôt $\delta(e_j)$ après le début de l'exécution du noeud v_i avec $pe_j \in In(v_i)$. Ces offsets de consommation tardive sont aussi nommés *retard de consommation*.
- *Les offsets de production :* les offsets de production modélisent les temps relatifs associés aux ports de sortie des noeuds du graphe. Ces offsets représentent la production précoce d'une valeur d'un port de sortie ps_k au plus tard $\delta(ps_k)$ avant la complétion de l'exécution du noeud v_i $ps_k \in Out(v_i)$. Ces offsets de production précoce sont aussi nommés *retard de production* (bien que l'on devrait plutôt parler d'avance de production).

Les offsets ici présentés permettront dans les phases d'exploitation du graphe de relâcher les contraintes liées aux prédécesseurs et aux successeurs des noeuds possédant des offsets. Il sera expliqué par la suite comment calculer les retards de consommation et de production associés ports. Enfin nous détaillerons une technique d'ordonnancement du graphe qui permettra d'exploiter ces informations.

Règle d'exécution

L'ensemble des noeuds décrits dans cette partie possèdent une règle d'exécution qui leur est propre. Nous pouvons toutefois spécifier que l'exécution partielle ou totale des noeuds est fonction de leur nature et de la validation d'un ou plusieurs de leurs ports d'entrée. Le modèle d'exécution utilisé est basé sur des machines à états finis de type *Moore* et se rapproche des réseaux de Pétri temporisés [Bacc91].

Changement d'état des ports d'entrée/sortie

Avec notre graphe nous définissons que les ports d'entrée et les ports de sortie des noeuds peuvent se trouver dans deux états distincts qui appartiennent à l'ensemble *Etat = { validé, non-validé }* . Afin de simplifier par la suite les écritures nous définissons que l'état *validé* est transposé dans l'ensemble booléen à la valeur $true$ et *non-validé* est transposé à la valeur $false$. Nous pouvons donc réécrire l'ensemble

état comme : $Etat = \{true, false\}$.

Nous définissons la fonction ε qui permet de consulter et fixer l'état d'un port d'entrée ou d'un port de sortie. Nous définissons la fonction $\varepsilon : Etat \leftarrow InOut(v_i)$ qui renvoie l'état du port d'entrée ou de sortie considéré. Afin d'exprimer les règles d'exécution des noeuds nous définissons deux opérateurs permettant d'appréhender une transition d'état sur un port d'entrée pe_j et de provoquer une transition d'état sur un port de sortie ps_k.

- L'opérateur "?" permet d'appréhender le changement d'état d'un port d'entrée pe_j du noeud v_i. Nous définissons la fonction $? : boolean \leftarrow In \times Etat$ qui pour une transition de l'état $false$ à l'état $true$ va renvoyer la valeur $true$, et la valeur $false$ sinon. Par exemple, la notation $?\varepsilon(pe_j)$ représente une attente d'une transition de l'état du port d'entrée pe_j ; le résultat obtenu sera égal à la valeur $true$ si et seulement si une transition positive a lieu sur le port pe_j.
- L'opérateur "!" permet de spécifier le changement d'état d'un port de sortie ps_k du noeud v_i. Nous définissons la fonction $! : Out \times Etat \leftarrow Etat$ qui permet de faire changer l'état de validation actuel du port de sortie ps_k appartenant au noeud v_i. Par exemple, la notation $!\varepsilon(ps_k) \leftarrow \neg\varepsilon(ps_k)$ représente un changement d'état de la sortie ps_j (sa négation).

Considérons un noeud v_i possédant un unique port d'entrée noté pe et un unique port de sortie noté ps. La règle d'exécution du noeud considéré est : lorsque le port d'entrée pe est validé, on valide le port de sortie ps après le temps nécessaire à la complétion du noeud v_i noté $\lambda(v_i)$. Cette règle d'exécution peut être mise en équation (équation 3.5).

$$!\varepsilon(ps)_{|\lambda(v_i)} \leftarrow ?\varepsilon(pe) \tag{3.5}$$

Lorsque l'on observe un changement d'état du port d'entrée ($?\varepsilon(pe)$), on procède à la validation du port de sortie ($!\varepsilon(ps)$) après le temps $\lambda(v_i)$ nécessaire à la complétion du noeud.

Valeurs acquises/transmises par les ports d'entrées/sorties

Les ports d'entrée et de sortie autorisent les noeuds à recevoir des valeurs de leurs prédécesseurs et à émettre des valeurs qui serviront à leurs successeurs. Afin de lire et d'écrire des valeurs sur les ports d'entrée et de sortie des noeuds, nous définissons la fonction $\nu : E_{Donnees} \leftarrow E_{Donnees}$ avec $E_{Donnees}$ l'ensemble des données qui peuvent transiter entre les noeuds du graphe (données structurées, scalaires, variables, complexes, ...). La relation entre les ports d'entrée et de sortie du noeud v_i peut être écrite sous forme d'équations (3.6, 3.7).

$$\nu(ps_j) = f_x(\nu(pe_1), \nu(pe_2), \ldots, \nu(pe_n)) \tag{3.6}$$

$$\nu(ps_j) = \nu(pe_k) \tag{3.7}$$

Le cas général est présenté dans l'équation (3.6) où la valeur transmise par le port de sortie ps_j est une fonction f_x des valeurs reçues sur les ports d'entrée $\{pe_1, pe_2, \ldots, pe_n\}$ du noeud. L'équation 3.7 est

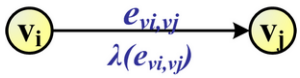

FIG. 3.3 – Liaison entre les noeuds v_i et v_j.

un cas particulier qui exprime la relation impliquant que la sortie du noeud noté ps_j possède la même valeur que l'entrée pe_k du noeud v_i. Les affectations de valeurs ainsi exprimées sont réalisées lors de la validation des sorties après complétion du noeud.

3.2.2 Les arcs reliant les noeuds

Sémantique

L'ensemble $E \subseteq V \times V$ est un ensemble d'arcs orientés reliant les ports des noeuds du graphe. L'arc noté e_{v_i,v_j}, présent dans le graphe G, représente la dépendance de donnée qui existe entre les noeuds v_i et v_j (figure 3.3). Les arcs sont ordonnés et il existe au maximum un arc noté e_{v_i,v_j} entre le port de sortie $ps_p \in Out(v_i)$ et $pe_q \in In(v_j)$.

Les arcs ont un double rôle dans la modélisation fonctionnelle : tout d'abord ils modélisent les dépendances de données (variable, scalaire, structure, ...) entre les différentes opérations/affectations. Ensuite, ils sont utilisés dans le modèle d'exécution afin de transmettre les validations entre les noeuds dont la complétion est achevée et leurs successeurs pour que ces derniers puissent commencer leurs propres exécutions.

A un arc sera associé un moyen de liaison/connexion entre les différents éléments qui composent l'architecture matérielle finale. Afin de modéliser les contraintes d'ordre temporel dans l'architecture finale, les arcs sont pondérés d'un poids noté $\lambda(e_{v_i,v_j})$, qui modélise le temps nécessaire au transfert de la donnée entre le noeud v_i et le noeud v_j.

Syntaxe

La notion de connexion exprime une relation directe de dépendance de données (précédence d'exécution) entre deux noeuds du graphe et est représentée par un arc. Une connexion entre v_i et v_j est dénotée par le triplet noeud-arc-noeud noté $(v_i, e_{i,j}, v_j)$. Cette notation est simplifiée par e_{v_i,v_j}. Soit E l'ensemble des arcs contenus dans le graphe. Un arc $e_{v_i,v_j} \in E$ modèlise la dépendance de données orientée entre les couples de noeuds v_i et v_j.

Un arc $e_{v_i,v_j} \in E$ est donc une connexion orientée entre un port de sortie et un port d'entrée, appartenant à 2 noeuds du graphe G. Un arc possède *un unique* prédécesseur (nommé aussi entrée de l'arc) et un unique successeur (nommé aussi sortie de l'arc). Si un noeud sur un de ses ports de sortie nécessite plusieurs connexions vers différents ports d'entrée de ses successeurs, alors il est nécessaire d'utiliser autant d'arcs qu'il doit y avoir de connections différentes (couples : port de sortie, port d'entrée) comme cela est montré dans la figure 3.4.

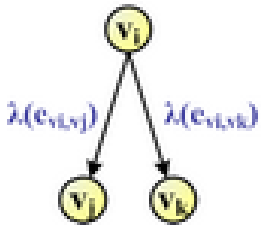

FIG. 3.4 – Liaison d'une sortie vers plusieurs entrées.

FIG. 3.5 – Connexion $e_{i,j}$ entre 2 noeuds (v_i, v_j).

Règle d'exécution

Un arc est exécuté (validé) lorsque son unique noeud prédécesseur v_i est arrivé à complétion et que la sortie $s_k \in Out(v_i)$ qui lie l'arc e_{v_i,v_j} au noeud a été validée. A ce moment là, l'arc valide l'entrée $e_k \in Out(v_j)$ avec v_j successeur de l'arc e_{v_i,v_j} après un délai égal au poids de l'arc $\lambda(e_{v_i,v_j})$.

Si nous considérons l'arc e_{v_i,v_j}, qui relie un port du noeud v_i à un port du noeud v_j (fig. 3.5), la règle d'exécution de l'arc peut être décrite par l'équation 3.4.

$$!\varepsilon(ps)_{|\lambda(e_{v_i,v_j})} \leftarrow ?\varepsilon(pe) \tag{3.8}$$

Dans l'équation pe représente la source de l'arc e_{v_i,v_j} (la sortie du noeud v_i) et ps représente la destination de l'arc (l'entrée du noeud v_j) (figure 3.5).

Modèle d'implémentation

L'implémentation physique d'un arc au sein de l'architecture matérielle finale ne correspond pas obligatoirement à l'instanciation d'un composant matériel à proprement parler. Un arc sera implémenté comme étant une réservation sur un canal de communication (fil, bus, mux, ...) entre les différents opérateurs matériels utilisés par les opérations qu'il connecte.

3.2.3 Définitions générales relatives au graphe

Notion de connexion

Il y a une connexion avant entre deux noeuds v_i et v_j s'il existe un arc e_{v_i,v_j} dans le graphe G allant d'un port de sortie ps_p du noeud v_i à un port d'entrée pe_q noeud v_j.

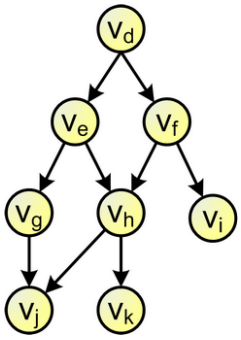

FIG. 3.6 – Chemin entre le noeud v_i et v_j.

Notion de chemin

La notion de chemin exprime une relation indirecte orientée entre deux noeuds du graphe et est représentée par une chaîne de connexions reliant ces deux noeuds. Un chemin relie deux noeuds v_f et v_j s'il existe une suite de noeuds connectés dans le graphe allant d'un port de sortie ps_p de v_f à un port d'entrée pe_q de v_j (figure 3.6). Soit C_{v_f,v_j}, un chemin allant de v_f à v_j dans le graphe G. Un chemin $C = (V_c, E_c)$ avec $V_c \in V$ et $E_c \in E$ est un chemin allant de v_f vers v_j si et seulement si :

$$v_f \in V, v_j \in V \exists ! e_{v_j,v_k} \in E_c \; et \; (V_c/v_j, E_c/e_{v_j,v_k}) \in C^b_{v_k,v_f} \; ou \; E_c = \{e_{v_f,v_j}\} \; et \; V_c = \{v_j, v_f\} \tag{3.9}$$

Le chemin C doit contenir v_f et v_j, et il existe un et un seul arc noté $e_{f,k}$ partant de v_f tel que le chemin C privé du noeud v_f et de l'arc $e_{f,k}$ est un chemin de v_k vers v_j.

Notion de route

La notion de route exprime une relation indirecte entre deux noeuds v_i et v_j du graphe. Cette relation est composée de $\{1 \rightarrow n\}$ chemin(s) reliant un port du noeud v_i à un port du noeud v_j. Un route relie deux noeuds v_i et v_j s'il existe au moins un chemin dans le graphe allant d'un port de sortie ps_p de v_i à un port d'entrée pe_q de v_j.

Soit R_{v_i,v_j} la route allant de v_i à v_j dans le graphe G. La route R_{v_i,v_j} contient la liste de tous les chemins $C_s = (V_{cs}, E_{cs})$ permettant d'aller du noeud v_i vers v_j. La longueur de la route R_{v_i,v_j} est définie comme étant le temps maximum nécessaire entre le début de l'exécution du noeud v_i et le début de l'exécution de v_j. Cet intervalle de temps est imposé par le plus long des chemins compris dans la route R_{v_i,v_j}. Cette durée peut être exprimée par :

$$\lambda(R_{v_i,v_j}) = \max_{C_{x,y} \in R_{v_i,v_j}} \lambda(C_{x,y})$$

Notion de prédécesseur

Un prédécesseur du noeud v_j est un noeud v_i qui possède au moins un chemin C_{v_i,v_j} dans le graphe $G(V,E)$ reliant un port de sortie du noeud v_i à un port d'entrée du noeud v_j. L'ensemble des noeuds qui possèdent au minimum un chemin avec v_j sont nommés : "ensemble des prédécesseurs du noeud v_j" ou "graphe amont de v_j". et est noté : $Pred_{all}(v_j)$. Nous définissons la fonction $Pred_{all}(v) : V \leftarrow V$ la fonction qui, pour le noeud v_i, renvoie la liste de tous les prédécesseurs du noeud $v_i \in V$. Dans l'exemple fourni en figure 3.6, l'ensemble des prédécesseurs du noeud $v_h \in V$ noté $Pred_{all}(v_h) = \{v_d, v_e, v_f\}$.

Notion de prédécesseur immédiat

Un prédécesseur immédiat d'un noeud v_j est un noeud v_i qui possède un arc e_{v_i,v_j} avant direct dans le graphe $G(V,E)$ allant d'un port de sortie ps_p du noeud v_i à un port d'entrée pe_q du noeud v_j. L'ensemble des noeuds correspondant à ces critères est nommé : "ensemble des prédécesseurs immédiats du noeud v_j" et est noté : $Pred(v_j)$. Nous définissons la fonction $Pred(v) : V \leftarrow V$ la fonction qui, pour le noeud v_i, renvoie la liste de tous les prédécesseurs immédiats du noeud $v_i \in V$. Dans l'exemple fourni en figure 3.6, l'ensemble des prédécesseurs du noeud $v_h \in V$ noté $Pred(v_h) = \{v_e, v_f\}$.

Notion de successeur

Un successeur du noeud v_i est un noeud v_j qui possède au moins un chemin C_{v_i,v_j} dans le graphe $G(V,E)$ allant du noeud v_i au noeud v_j. Les noeuds possédant au minimum un chemin partant de v_i et les atteignant sont nommés : "successeurs du noeud v_i" ou bien "graphe aval de v_j". L'ensemble de ces noeuds est noté : $Succ_{all}(v_i)$. Nous définissons la fonction $Succ_{all}(v) : V \leftarrow V$ la fonction qui, pour le noeud v_i, renvoie la liste de tous les successeurs du noeud $v_i \in V$. Dans l'exemple fourni en figure 3.6, l'ensemble des successeurs du noeud $v_e \in V$ est égal à $Succ_{all}(v_e) = \{v_g, v_h, v_j, v_k\}$.

Notion de successeur immédiat

Un successeur immédiat du noeud v_i est un noeud v_j qui possède un arc $e_{i,j}$ direct dans le graphe $G(V,E)$ liant un port de sortie ps_p du noeud v_i à un port d'entrée pe_q du noeud v_j. L'ensemble des noeuds correspondant à ces critères est nommé : "ensemble des successeurs immédiats du noeud v_i" et est noté : $Succ(v_i)$. Nous définissons la fonction $Succ(v) : V \leftarrow V$ la fonction qui, pour le noeud v_i, renvoie la liste de tous les successeurs immédiats du noeud $v_i \in V$. Dans l'exemple fourni en figure 3.6, l'ensemble des successeurs du noeud $v_e \in V$ noté $Succ(v_e) = \{v_g, v_h\}$.

Chemin critique dans un graphe

La notion de chemin critique noté $C_r(v_i, v_j)$ au sein d'un graphe G exprime la relation indirecte la plus longue reliant les noeuds v_i et v_j. Le chemin critique représente le chemin dont la durée est maximum

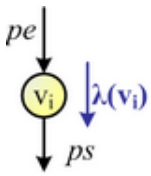

FIG. 3.7 – Modélisation d'un noeud variable.

entre l'exécution du noeud v_i et l'exécution du noeud v_j. Le chemin critique entre les noeuds v_i et v_j peut être exprimé par l'équation 3.10

$$\lambda(C_r(v_0, v_n)) = \lambda(\max_{p(v_0,v_n)}(C_p)) \tag{3.10}$$

La notion de chemin critique noté C_r appliqué à un graphe G exprime la relation indirecte la plus longue reliant les noeuds v_0 et v_n. La durée du chemin critique $C_r(G)$ du graphe G se notera $\lambda(G)$.

3.3 Définition des noeuds du graphe

3.3.1 Le noeud variable

Sémantique du noeud

Nous définissons tout d'abord les noeuds de type *variable* notés $V_{variables} \in V_{donnees}$. Ces noeuds représentent l'ensemble des variables contenues dans le graphe. La variable notée v_i présentée dans la figure 3.7 représente une donnée qui sera écrite et/ou lue dans le graphe en fonction des arcs entrants/sortants qu'elle possède. Les noeuds variables représentent des points de mémorisation qui seront implémentés en matériel par des registres. Au sein de notre modèle de représentation CSFG, une variable ne peut être accédée en écriture qu'une seule fois. Les variables accédées plusieurs fois en écriture sont systématiquement renommées lors de la phase de génération graphe afin de résoudre les conflits d'accès au cours du temps (mise à jour $\leftrightarrow$ dépendance écriture/écriture et écriture/lecture).

Syntaxe

Un noeud variable noté $v_i \in V_{variables}$ est un noeud qui possède un unique port d'entrée noté $In(v_i) = \{pe\}$ et possède un unique port de sortie noté $Out(v_i) = \{ps\}$. Le port de sortie ps du noeud variable peut être relié en direction de ports d'entrée différents grâce à l'utilisation d'un arc distinct par destination. Le noeud variable noté v_i est pondéré par son temps d'exécution noté $\lambda(v_i)$.

Dans la figure 3.7, nous avons modélisé un noeud variable possédant un port d'entré et un port de sortie.

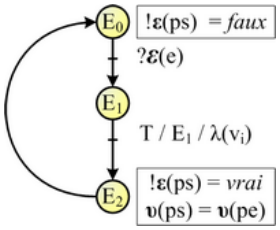

FIG. 3.8 – Modélisation du comportement d'un noeud variable.

Règle d'exécution

L'exécution d'un noeud variable dépend directement de l'exécution de son unique prédécesseur. Un noeud variable doit tout d'abord avoir son port d'entrée de validé par son prédécesseur ; ensuite il peut s'exécuter et valider son port de sortie après avoir attendu $\lambda(v_i)$. Cette règle d'exécution est traduite par l'équation (3.11).

$$!\varepsilon(ps)_{|\lambda(v_i)} \leftarrow ?\varepsilon(pe) \tag{3.11}$$

L'équation signifie que lorsque le port d'entrée *pe* du noeud variable v_i est validé par son prédécesseur alors le noeud v_i peut s'exécuter et valider son port de sortie *ps* après le temps nécessaire à la complétion de son exécution noté $\lambda(v_i)$. Le processus d'exécution du noeud v_i peut être exprimé à l'aide d'une machine à états finis temporisée composée de 3 états. Cette machine d'états est exposée dans la figure 3.8.

Règle de calcul des sorties

Lorsque le noeud variable v_i est exécuté, la valeur du port de sortie notée $\nu(ps)$ est fonction de la valeur présente sur le port d'entrée du noeud notée $\nu(pe)$. Dans le cas d'un noeud variable, la valeur présente sur le port d'entrée du noeud est recopiée sur le port de sortie. Cette propriété est résumée par l'équation 3.12.

$$\nu(ps) = \nu(pe) \tag{3.12}$$

Règle d'implémentation

Les noeuds "variable" présents dans la modélisation fonctionnelle ont une implémentation de type registre ou mémoire dans l'architecture matérielle finale. Le temps $\lambda(v_i)$ du noeud variable v_i représente le temps nécessaire à la mémorisation de la donnée (écriture).

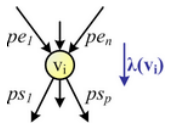

FIG. 3.9 – Modélisation d'un noeud de type opération.

3.3.2 Le noeud opération

Sémantique du noeud

Nous définissons les noeuds de type *opération* notés $V_{opr} \in V_{operations}$. Ces noeuds représentent l'ensemble des opérations arithmétiques ou logiques contenues dans le graphe. Le noeud de type opération noté v_i (figure 3.9) représente une opération qui sera exécutée dans le graphe en fonction des arcs entrants/sortants qu'il possède. Les noeuds opérations représentent des opérations qui seront implémentées sur des opérateurs matériels.

Syntaxe

Un noeud opération, noté $v_i \in V_{opr}$, est un noeud qui possède $\{1 \rightarrow n\}$ port(s) d'entrée noté(s) $In(v_i) = \{pe_1, \ldots, pe_n\}$ et possède $\{1 \rightarrow p\}$ port(s) de sortie noté(s) $Out(v_i) = \{ps_1, \ldots, ps_p\}$. Tous les noeuds de type opération $v_i \in V_{opr}$ possèdent une durée d'exécution $\lambda(v_i)$ qui représente le temps nécessaire à la complétion intégrale de l'opération v_i après le début de son exécution.

Règle d'exécution

L'exécution d'un noeud opération dépend directement de la complétion de l'ensemble de ses prédécesseurs. Un noeud opération nécessite, afin de valider son ou ses port(s) de sortie, d'avoir son ou ses port(s) d'entrée validée(s) par l'ensemble de ses prédécesseurs. Cette règle d'exécution est traduite par l'équation 3.13.

$$!(\varepsilon(ps_1), \varepsilon(ps_2), \ldots, \varepsilon(ps_p))_{|\lambda(v_i)} \leftarrow ?(\varepsilon(pe_1) \wedge \varepsilon(pe_2) \wedge \varepsilon(pe_n)) \tag{3.13}$$

Cette équation signifie que lorsque l'ensemble des ports d'entrée $\{pe_1 \rightarrow pe_n\}$ du noeud opération v_i sont validés par l'ensemble des prédécesseurs du noeud, alors le noeud opération v_i peut être exécuté et l'ensemble de ses ports de sortie $\{ps_1 \rightarrow ps_p\}$ sera validé après un temps d'exécution noté $\lambda(v_i)$. Le processus d'exécution du noeud opération v_i peut être modélisé à l'aide d'une machine à états finis temporisée composée de 3 états cycliques comme exposé dans la figure 3.10.

Règle de calcul des sorties

Lorsque le noeud opération v_i est exécuté, les valeurs des ports de sortie, notées $\nu(ps_i)$ sont fonction des valeurs présentes sur les ports d'entrée du noeud notée $\nu(pe_i)$. Dans le cas d'un noeud opération, chaque

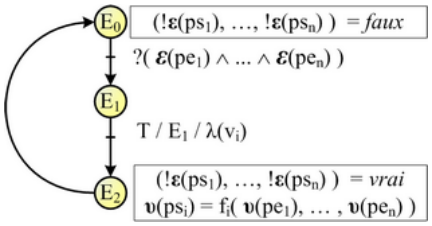

FIG. 3.10 – Modélisation du comportement d'un noeud opération.

port de sortie du noeud est une fonction des différentes valeurs présentes sur les ports d'entrée. Cette propriété peut s'exprimer de la manière suivante :

$$\nu(s_1) = f_1(\nu(e_1), \nu(e_2), \ldots \nu(e_n)) \tag{3.14}$$

$$\nu(s_i) = f_i(\nu(e_1), \nu(e_2), \ldots \nu(e_n)) \tag{3.15}$$

Les fonctions de calcul, notées f_i, dans l'équation 3.15 sont des fonctions arithmétiques ou logiques dépendant de la nature des opérations réalisées par le noeud v_i. Suivant la nature des opérations effectuées, le temps nécessaire pour l'activation de certaines sorties en fonction de la date d'activation des entrées du noeud peut varier.

Règle d'implémentation

Les noeuds de type opération présents dans la modélisation fonctionnelle ont une implémentation de type opérateur dans l'architecture finale. Le poids $\lambda(v_i)$ associé au noeud opération v_i représente le temps de traversé de l'opérateur physique (sa latence pour un opérateur classique et sa cadence un opérateur pipeline).

3.3.3 L'opération conditionnelle

Sémantique du noeud

Afin de modéliser les structures conditionnelles présentes dans les descriptions algorithmiques, nous devons définir un modèle de représentation adapté à l'exécution conditionnelle de certaines opérations. Pour cela, nous avons été amenés à définir une sous-classe des noeuds de type opération dont la condition de validation des sorties a évolué par rapport aux opérations dites "classiques". Cela permet de valider uniquement la sortie adéquate correspondant à la branche conditionnelle à exécuter.

Définition 3.3.1 (*Les Opérations Conditionnelles*)

Les opérations conditionnelles nommées aussi noeuds conditionnels, notées $V_{opr-cond} \in V_{operations}$, sont définies comme étant le sous ensemble des opérations du graphe qui a pour but de rendre l'exécution d'un ensemble de noeuds (calcul ou affectation) conditionnelle à la valeur du résultat de

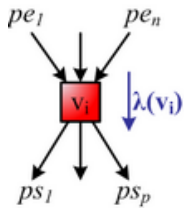

FIG. 3.11 – Modélisation d'une opération conditionnelle.

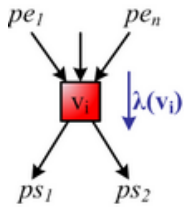

FIG. 3.12 – Modélisation simplifiée d'une structure *If-Then-Else*.

l'opération conditionnelle. L'ensemble des opérations conditionnelles, noté $V_{opr-cond}$, est l'ensemble des opérations du graphe qui sont de type conditionnel et qui ont pour but de conditionner, grâce aux dépendances de données, un certain nombre d'exécution de noeuds.

Syntaxe

Un noeud *opération conditionnelle*, noté $v_i \in V_{opr-cond}$, appartient à l'ensemble des noeuds opérations du graphe. Ce noeud possède $\{1 \rightarrow n\}$ port(s) d'entrée et $\{2 \rightarrow p\}$ ports de sortie. Ses sorties sont mutuellement exclusives de telle sorte qu'un seul port de sortie peut être validé lors d'une exécution du noeud v_i. La figure 3.11 modélise un noeud de type opération conditionnelle.

Dans le cas particulier d'une structure de type $If-Then-Else$, le noeud "opération conditionnelle" possède uniquement 2 ports de sortie. La modélisation d'un tel noeud est présentée dans la figure 3.12.

Règle d'exécution

L'exécution d'un noeud de type opération conditionnelle nommé v_i dépend de la complétion de l'ensemble de ses $\{1 \rightarrow n\}$ prédécesseurs. Le noeud v_i nécessite la validation de l'ensemble de ses ports d'entrée $\{pe_1 \rightarrow pe_n\}$ afin de valider un de ses ports de sortie en fonction du résultat de l'opération. Le choix du port de sortie à valider se fait en fonction d'un ensemble de fonctions de calcul internes. Ces fonctions vont permettre de : calculer la valeur à transmettre par l'intermédiaire des ports sorties et de décider si les ports de sortie doivent être validés ou non.

$$!\varepsilon(ps_1)_{|\lambda(v_i)} \leftarrow g_1(\nu(pe_1),\ldots,\nu(pe_n)) \wedge ?(\varepsilon(pe_1) \wedge \ldots \wedge \varepsilon(pe_n)) \tag{3.16}$$

$$!\varepsilon(ps_2)_{|\lambda(v_i)} \leftarrow g_2(\nu(pe_1),\ldots,\nu(pe_n))\wedge?(\varepsilon(pe_1)\wedge\ldots\wedge\varepsilon(pe_n)) \quad (3.17)$$

$$\ldots \quad (3.18)$$

$$!\varepsilon(ps_i)_{|\lambda(v_i)} \leftarrow g_i(\nu(pe_1),\ldots,\nu(pe_n))\wedge?(\varepsilon(pe_1)\wedge\ldots\wedge\varepsilon(pe_n)) \quad (3.19)$$

Si nous analysons l'équation générale 3.19 qui exprime la règle d'exécution d'un port de sortie ps_i d'un noeud opération conditionnelle, nous constatons que la validation d'une sortie est bien dépendante de la validation des ports d'entrée. La valeur attribuée à un port de sortie résulte d'une relation particulière entre les valeurs présentes sur les ports d'entrée. La différence majeure avec un noeud opération "classique" émane de la condition d'exécution (fonction notée g_i) qui est différente pour chacune des sorties. La validation des ports de sortie du noeud v_i est donc dépendante des entrées du noeud, mais aussi d'une fonction de validation notée g_i. Les fonctions g_i fournissent après calcul une valeur booléenne appartenant à l'ensemble $\{vrai, faux\}$. En fonction du nombre de ports de sortie que possède le noeud opération conditionnelle et des fonctions g_i, ont peut distinguer différents cas :

- Si $g_1 = g_2 = \ldots = g_p = vrai$ alors le noeud v_i est un noeud opération "classique" et non un noeud opération de type conditionnel.
- Si $p = 2$ et $g_1 = \overline{g_2}$ alors le noeud v_i est un noeud opération de type conditionnel utilisé afin de modéliser une structure conditionnelle de type *If-Then-Else*.
- Si $p > 2$ et que les fonctions g_i sont différentes par couple (si $\forall i \forall j\ (i \neq j) \Leftrightarrow g_i \neq g_j$) alors le noeud v_i est un noeud opération de type conditionnel utilisé afin de modéliser une structure conditionnelle de type $switch-case$. Dans ce cas là, les sorties $Out(v_i)$ du noeud v_i considéré sont mutuellement exclusives.

Dans le cas d'une structure conditionnelle composée d'une unique branche *If*, une modélisation de type *If-Then-Else* avec la branche *Then* vide est opérée. Cette modélisation permet de conserver une représentation de mise à jour des données cohérente lorsque l'on exécute ou non la branche conditionnelle.

Définition 3.3.2 (*Sorties Mutuellement Exclusives (IF)*)

Deux ports de sortie d'un noeud du graphe sont dits mutuellement exclusifs si et seulement si leur condition de validation au sein d'une même structure conditionnelle est différente : les opérations partagent la même condition qui dans un cas vaut une valeur booléenne $b_i \in \{true, false\}$ et dans l'autre son opposé $b_j \in \{true, false\}/b_i$.

$$g_1 \wedge g_2 = \emptyset$$

$$g_1 \vee g_2 = 1$$

Définition 3.3.3 (*Sorties Mutuellement Exclusives (CASE)*)

Les n ports de sortie d'un noeud opération conditionnelle sont dits mutuellement exclusifs si et seulement si tous les ports de sortie du noeud sont mutuellement exclusifs en couple $(ps_i, ps_j$ avec $i, j \in Out(v_i)/\{i = j\}$. De plus, pour chaque combinaison d'entrées possibles du noeud, une unique sortie est validée par exécution du noeud : si $s_i = true$ alors $\forall ps_k \in Out(v_i)/\{ps_j\} \Rightarrow ps_k = false$.

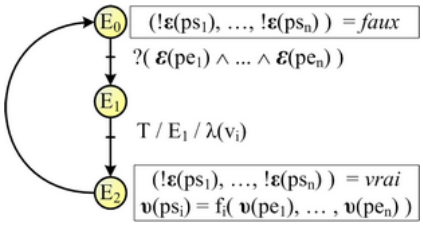

FIG. 3.13 – Machine d'états finis modélisant le comportement d'une opération conditionnelle.

$$\forall i, \forall j, g_i \wedge g_j = \emptyset$$

$$\forall i, \forall j, g_i \vee g_j = 1$$

Règle de calcul des sorties

Le port de sortie $ps_j \in Out(v_i)$ est validé après la complétion de l'exécution du noeud v_i si la valeur du résultat de la fonction g_j associée à ps_j conditionnant sa validation est positive ($true$). Dans ce cas, la valeur transmise par le port de sortie ps_j au successeur du noeud peut s'écrire comme indiqué dans 3.20.

$$\nu(ps_i) = g_i(\nu(pe_1), \ldots, \nu(pe_n)) \tag{3.20}$$

On peut remarquer que le port de sortie ps_i est validé après complétion de l'exécution du noeud v_i. Ce port de sortie transmet à ses successeurs la valeur $true$ tandis que les autres ports de sortie transmettent la valeur $false$.

Règle d'implémentation

Afin d'implémenter les structures conditionnelles, nous pouvons mettre en oeuvre différentes stratégies d'implémentation comme cela a été détaillé dans le chapitre 2 (exécution parallèle et partage des opérateurs). La stratégie d'implémentation sera choisie en fonction des capacités de l'architecture cible (possibilité d'effectuer des retours d'états) et des contraintes de synthèse.

3.3.4 Le noeud variable conditionnée

Sémantique du noeud

L'ensemble des *variables conditionnées*, noté $V_{var-cond} \in V_{donnees}$, est l'ensemble des noeuds de type variable qui possède comme particularité de posséder plusieurs sources d'affectation possibles résultant d'une structure conditionnelle. Les ports d'entrée d'un noeud variable conditionnée doivent être mutuellement exclusifs afin de préserver l'hypothèse faite au départ sur l'unicité de l'affectation des variables.

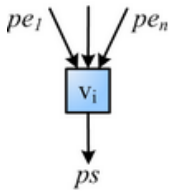

FIG. 3.14 – Modélisation d'un noeud variable conditionnée dans le cadre d'une structure *Switch-Case*.

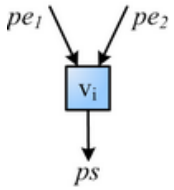

FIG. 3.15 – Modélisation d'un noeud variable conditionnée dans le cadre d'une structure *If-Then-Else*

Définition 3.3.4 (*Les Variables Conditionnées*)

Les variables conditionnées sont des noeuds de type variable définis comme étant le sous-ensemble des variables possédant $\{2 \rightarrow n\}$ ports d'entrée et un port de sortie. L'ensemble des variables conditionnées noté $V_{var-cond} \in V_{donnees}$ est l'ensemble des noeuds du graphe dont l'exécution est conditionnée par uniquement une de ses n entrées qui sont obligatoirement mutuellement exclusives.

Syntaxe

Une variables conditionnée possède $\{2 \rightarrow n\}$ ports d'entrée, chacun d'entre eux est relié à un chemin conditionnel différent (figure 3.14). Lors de l'exécution du noeud v_i, seul un de ces ports d'entrée peut être validé à un instant d'exécution donné (l'affectation ne peut s'effectuer qu'à partir d'une branche conditionnelle).

La représentation d'un noeud variable conditionnée dans le cadre de la modélisation d'une structure *Switch-Case* peut être simplifié pour représenter une structure *If-Then-Else*. Dans ce dernier cas, le noeud variable conditionnée ne nécessite que 2 ports d'entrée. La modélisation pour une structure *If-Then-Else* est montrée en figure 3.15.

Règle d'exécution

L'exécution d'un noeud variable conditionnée dépend directement de l'exécution de ses prédécesseurs. Un noeud variable conditionnée, afin de valider son port de sortie, doit avoir un et un seul de ses ports d'entrée validé (se trouvant dans l'état *true*). Après complétion de l'exécution du noeud variable conditionnée v_i, le port sortie *ps* est validé afin de propager l'ordre d'exécution au travers du graphe. Cette règle d'exécution peut être représentée par l'équation 3.21. Cette équation exprime que lorsque le noeud variable conditionnée est validé par un de ses prédécesseurs, alors il est exécuté et son port de sortie *s* est

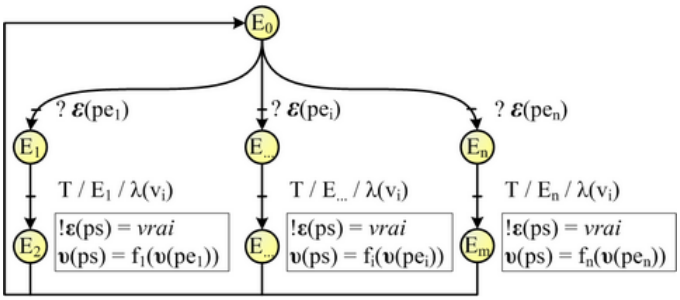

FIG. 3.16 – Modélisation du comportement d'une variable conditionnée.

validé après un temps d'exécution $\lambda(v_i)$.

$$!\varepsilon(ps)_{|\lambda(v_i)} \leftarrow ?(\varepsilon(pe_1) \vee \varepsilon(pe_2) \vee \ldots \vee \varepsilon(pe_n)) \tag{3.21}$$

$$!\varepsilon(ps)_{|\lambda(v_i)} \leftarrow ?(\varepsilon(pe_1) \vee \varepsilon(pe_2) \tag{3.22}$$

Nous pouvons simplifier l'équation 3.21 dans le cas d'une structure conditionnelle à deux branches modélisant une structure $IF-THEN-ELSE$. Dans ce cas, nous obtenons l'équation 3.22.

Le processus d'exécution d'un noeud variable conditionné v_i peut être vu comme une machine à états finis temporisée composée de 7 états cycliques comme exposé dans la figure 3.16.

Règle de calcul des sorties

Lorsque le noeud variable conditionnée v_i est exécuté, la valeur transmise par son port de sortie ps notée $\nu(ps)$ est fonction de la valeur présente sur les ports d'entrée pe_k du noeud noté $\nu(pe_k)$. Dans le cas d'un noeud variable conditionnée, la valeur présente sur le port d'entrée validé est recopiée sur le port de sortie. Cette propriété peut être exprimée par l'équation 3.23.

$$\nu(ps) = f(\nu(pe_1), \nu(pe_2), \ldots, \nu(pe_n)) \tag{3.23}$$

$$\nu(ps) = f(\nu(pe_1), \nu(pe_2)) \tag{3.24}$$

Une simplification peut être apportée à cette équation lorsque la modélisation de la structure conditionnée est celle d'une structure $IF-THEN-ELSE$. Dans ce cas, l'équation simplifiée est donnée par 3.24.

Règle d'implémentation

Les noeuds de type variable conditionnée présents dans la modélisation fonctionnelle ont une implémentation de type registre ou mémoire dans l'architecture finale. Différentes méthodes permettent la

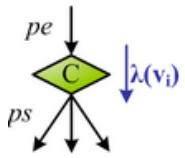

FIG. 3.17 – Modélisation d'un noeud de structure.

sélection de la valeur adéquate lors de l'exécution. Le choix de la méthode à implémenter sera réalisé lors de l'étape de synthèse du modèle.

3.3.5 Le noeud structure de données

Sémantique du noeud

Afin permettre la modélisation et la gestion des adressages statiques et dynamiques au sein du graphe, il est nécessaire de prendre en compte les structures de données de manière identique lors des accès déterministes et non déterministes. Pour cela, nous utilisons des noeuds spécifiques nommés noeuds de structure, $V_{structures} \in V_{donnees}$, afin de représenter les données structurées (les vecteurs à $n \times m$ dimensions) employés dans la description algorithmique. Le modèle de représentation présenté permet de considérer les conflits d'accès aux structures (lecture avant écriture et lecture après écriture) lors des accès : la règle des affectations uniques est appliquée aux noeuds de structure entraînant la création d'un nouveau noeud renommé lors de la phase de compilation à chaque accès en écriture. Cela permet de lever toute ambiguïté sur les dépendances/conflits d'accès.

Accès aux données structurées

En association à la représentation des données structurées nous définissons des noeuds de type opération permettant la modélisation des contraintes temporelles produites par les opérations de lecture/écriture au sein de la mémoire du système pour accéder aux données structurées. Ces noeuds opération dédiés à la lecture et à l'écriture des données sont respectivement noté $@_w$ et $@_r$. Un des avantages d'une modélisation explicite des accès à la mémoire provient du fait qu'il est ainsi possible de modéliser des temps d'accès différents suivant les mémoires qui sont accédées.

Syntaxe

Un *noeud de structure*, noté $v_i \in V_{structure}$, est un noeud possédant un seul port d'entrée et un port de sortie. Le port d'entrée modélise la relation de précédence avec le noeud qui active et produit la structure que modélise v_i. Le port de sortie permet de modéliser la relation de dépendance avec les noeuds qui vont consommer les données contenues dans la structure. La relation vers plusieurs successeurs du noeud de structure est réalisée à l'aide de plusieurs arcs qui sont connectés à l'unique port de sortie de la structure.

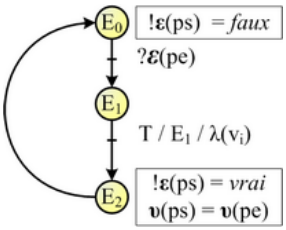

FIG. 3.18 – Modélisation du comportement de l'exécution d'un noeud de structure.

Le temps d'exécution modélisé par $\lambda(v_i)$ représente dans le cas d'un noeud de structure le temps nécessaire à la mémorisation de données dans la structure (écriture).

Règle d'exécution

La règle d'exécution d'un noeud de structure est comparable à celle d'un noeud variable : elle dépend directement de la complétion de son unique prédécesseur (et de la validation de son port d'entrée). Un noeud de structure nécessite, afin de valider son port de sortie, d'avoir son port d'entrée de validé par son prédécesseur. L'équation 3.25 exprime que le port d'entrée *pe* du noeud de structure v_i doit être validé par le prédécesseur du noeud afin que le noeud de structure v_i s'exécute et valide son port de sortie *ps* après un temps d'exécution noté $\lambda(v_i)$. Le processus d'exécution du noeud de structure v_i peut être vu comme une machine à états finis composée de 3 états cycliques comme exposé dans la figure 3.18.

$$!\varepsilon(ps)_{|\lambda(v_i)} \leftarrow ?\varepsilon(pe) \tag{3.25}$$

Dans le modèle d'exécution, nous considérons que l'ensemble des successeurs du noeud de structure va être activé en même temps. Cette activation simultanée implique des accès simultanés à la structure de donnée. Les contraintes d'accès seront considérées à l'implémentation du modèle sur l'architecture cible (lors de l'étape d'ordonnancement).

Règle de calcul des sorties

Lorsque le noeud de structure v_i est exécuté, la structure présente sur le port d'entrée *pe* noté $\nu(pe)$ est transmise sur le port de sortie *ps*, noté $\nu(ps)$. Cette règle de calcul des sorties est exprimée par l'équation 3.26.

$$\nu(ps) = \nu(pe) \tag{3.26}$$

Règle d'implémentation

Les noeuds de structure présents dans la modélisation fonctionnelle ont une implémentation de type mémoire dans l'architecture matérielle finale.

3.3.6 Le noeud opération hiérarchique

Sémantique du noeud

Nous avons présenté jusqu'à maintenant un ensemble de noeuds élémentaires permettant la modélisation des données et des opérations. L'ensemble de noeuds hiérarchiques noté $V_{hierarchique} \in V_{operations}$, autorise la modélisation de structures d'opérations complexes (boucles, séquences d'instructions, opérateurs séquentiels). Cette modélisation hiérarchique permet entre autre une réduction de la complexité du graphe. Les noeuds hiérarchiques sont constitués en interne d'un sous-graphe qui est perçu au niveau hiérarchique supérieur comme un noeud de type opération. Les noeuds hiérarchiques sont mis à profit afin de simplifier la modélisation et les traitements (optimisations, ordonnancement) du graphe en englobant un ensemble de traitements de manière hiérarchique. La complexité globale du processus de synthèse est ainsi réduite.

Les noeuds hiérarchiques sont créés à partir de la description algorithmique à modéliser. Un noeud hiérarchique est généré lorsque l'on rencontre au sein de la description algorithmique une des primitives algorithmiques suivantes :

- Un appel de fonction ou de procédure,
- Un coeur de boucle (en effet, chaque itération d'une boucle peut être modélisée sous la forme d'un noeud hiérarchique),
- Chacune des branches d'une même structure conditionnelle peut, elle aussi, être représentée sous forme d'un noeud hiérarchique.

Ces primitives algorithmiques sont, dans un premier temps, modélisées sous la forme de noeuds hiérarchiques dans notre modèle de représentation. Ces noeuds hiérarchiques pourront, dans un second temps, être mis à plat pour mettre en exergue le parallélisme global des opérations du graphe si cela est nécessaire.

Syntaxe

Un noeud hiérarchique (fig. 3.19), noté $v_h \in V_{hierarchique}$, est un noeud qui possède $\{1 \rightarrow n\}$ port(s) d'entrée et $\{1 \rightarrow p\}$ port(s) de sortie avec $n \geq 1$ et $p \geq 1$. Le noeud hiérarchique est composé d'un graphe interne de type *CSFG*. La somme des arcs entrant dans un noeud hiérarchique v_h est identique à la somme des arcs sortant du noeud v'_0 (noeud source du sous graphe interne). La somme des arcs sortant d'un noeud hiérarchique v_h est identique à la somme des arcs entrant dans le noeud v'_n (noeud puits du sous graphe interne).

Règle d'exécution

Les règles d'exécution du noeud hiérarchique sont basées sur les règles d'exécution des noeuds de type opération. Afin de pouvoir exécuter un noeud hiérarchique il est nécessaire que l'ensemble des ports d'entrée du noeud soit validé. Une fois les ports d'entrée validés, le noeud peut être exécuté. Après complétion, ses ports de sortie sont validés et valués. La différence entre un noeud hiérarchique et un

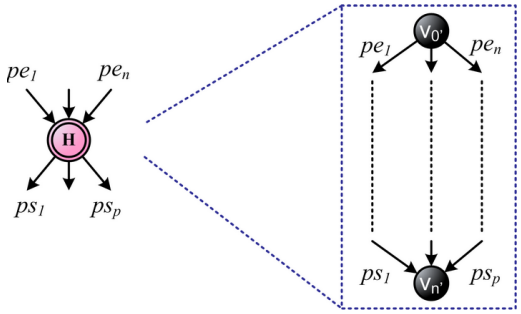

FIG. 3.19 – Modélisation d'un noeud hiérarchique.

noeud opération réside dans le *retard* que l'on peut associer aux ports d'entrée et de sortie du noeud. Les retards associés aux ports d'entrée sont notés $\delta(pe_i)$ et ceux associés aux sorties sont notés $\delta(ps_i)$. La règle d'exécution d'un noeud hiérarchique peut être traduite par l'équation 3.27.

$$!(\varepsilon(ps1)_{|\delta(ps_1)},\ldots,\varepsilon(ps_p)_{|\delta(ps_p)})_{|\lambda(v_i)} \leftarrow ?(\varepsilon(pe1)_{|\delta(pe_1)} \wedge \ldots \wedge \varepsilon(pe_n)_{|\delta(ps_n)}) \tag{3.27}$$

$$\forall j, \forall k : \lambda(v_i) - \delta(pe_j) + \delta(ps_k) \geq 0 \tag{3.28}$$

Pour toutes valeurs de j et k, les retards $\delta(pe_j)$ associés aux ports d'entrée et $\delta(ps_k)$ associés aux ports de sortie sont constants et dépendent de la constitution du noeud hiérarchique. Le délai $\delta(ps_k)$ représente le retard temporel d'exécution du port de sortie ps_k. Le délai $\delta(pe_j)$ représente le retard temporel d'activation du port d'entrée pe_j. Les retards associés aux différents ports du noeud v_i doivent respecter l'équation 3.28.

Règle de calcul des sorties

Lorsque le noeud hiérarchique v_i est exécuté, les valeurs des sorties, notées $\nu(ps_j)$, sont calculées en fonction des valeurs présentes sur les ports d'entrée du noeud, notées $\nu(pe_k)$. Dans le cas d'un noeud hiérarchique, chaque sortie du noeud est une combinaison d'opérations entre les différentes valeurs présentes sur les ports d'entrée. Cette propriété peut être exprimée par l'équation 3.29.

$$\nu(ps_j) = f_i(\nu(pe_1),\ldots\nu(pe_n)) \tag{3.29}$$

3.4 Généralisation des noeuds hiérarchiques

Nous venons de présenter une classe de noeuds nommés noeuds hiérarchiques. Les noeuds hiérarchiques permettent de modéliser un ensemble d'opérations complexes (fonctions, boucles, composants séquentiels, ...). Nous allons maintenant présenter une généralisation de l'utilisation des noeuds hiérarchiques afin d'avoir une représentation uniforme pour l'ensemble des noeuds du graphe.

Pour cela, nous allons présenter les règles de transformation d'un noeud du graphe en noeud hiérarchique. Cette partie vise à généraliser les attributs (retards) afin de simplifier les méthodes d'ordonnancement du graphe.

3.4.1 Modélisation des noeuds variables

Les noeuds variables présentés précédemment sont des noeuds qui possèdent une entrée et une sortie. Leur temps d'exécution noté $\lambda(v_i)$ représente le temps séparant la validation du port d'entrée *pe* de la validation du port de sortie *ps*. Un noeud variable peut donc être modélisé au sein du graphe par un noeud hiérarchique en posant les relations exprimées par les équations (3.30)(3.31).

$$\lambda(H) = \lambda(v_i) \tag{3.30}$$

$$\delta(pe) = \delta(ps) = 0 \tag{3.31}$$

3.4.2 Modélisation des noeuds opérations

Les noeuds de type opération sont des noeuds qui possèdent $\{1 \rightarrow n\}$ port(s) d'entrée et $\{1 \rightarrow p\}$ port(s) de sortie. Leur temps d'exécution, noté $\lambda(v_i)$ et représente le temps d'exécution maximum entre la validation de tous les ports d'entrée et la validation du dernier port de sortie. Un noeud opération peut donc être modélisé par un noeud hiérarchique en posant les relations exprimées par les équations (3.32) et (3.33).

$$\lambda(H) = \lambda(v_i) \tag{3.32}$$

$$\forall i, \forall j : \delta(pe_i) = \delta(ps_j) = 0 \tag{3.33}$$

3.4.3 Modélisation des noeuds de structure

Les noeuds de structure sont des noeuds qui possèdent un port d'entrée et un port de sortie. Leur temps d'exécution est noté $\lambda(v_i)$ et représente le temps d'exécution maximum entre la validation de l'entrée et la validation du port de sortie. Un noeud de structure peut donc être modélisé par un noeud hiérarchique en posant les relations exprimées par les équations (3.34) et (3.35).

$$\lambda(H) = \lambda(v_i) \tag{3.34}$$

$$\delta(pe) = \delta(ps) = 0 \tag{3.35}$$

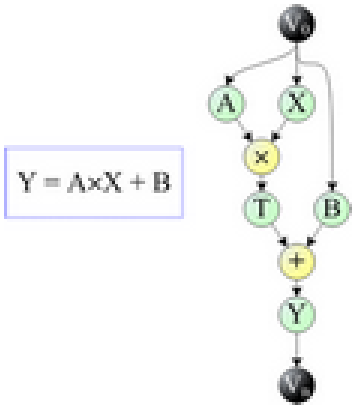

FIG. 3.20 – Exemple de modélisation d'une description "simple".

3.4.4 Modélisation des variables conditionnées

Les noeuds de type variable conditionnée sont des noeuds qui possèdent $\{1 \rightarrow n\}$ entrée(s) et une sortie. Leur temps d'exécution est noté $\lambda(v_i)$ et représente le temps d'exécution maximum entre la validation de tous les ports d'entrée et la validation du port de sortie. Un noeud variable conditionnée peut donc être modélisé au sein d'un noeud hiérarchique en posant les relations exprimées par équations (3.36)(3.37).

$$\lambda(H) = \lambda(v_i) \tag{3.36}$$

$$\forall k : \delta(pe_k) = \delta(ps) = 0 \tag{3.37}$$

3.5 Exemples de modélisations

Nous allons dans cette section présenter quelques exemples de modélisation réalisés à l'aide du modèle CSFG que nous avons développé.

3.5.1 Modélisation d'un flot de données

Dans ce premier exemple de modélisation, la figure 3.20 le graphe représentant un flot de donnée "simple". Ce graphe modélise le calcul de l'équation d'une droite $(Y = A \times X + B)$. Les noeuds v_0 et v_n sont respectivement les noeuds sources et les noeuds puits du graphe. L'ensemble des arcs liant les noeuds du graphe modélisent les dépendances de données qui vont contraindre l'ordre d'exécution des noeuds. Le début de l'exécution du graphe commence par la validation du noeud v_0, qui va valider l'ensemble de ses sorties. Une fois les successeurs immédiats de v_0 validés, ces derniers vont s'exécuter et valider leurs sorties en fonction de leurs règles d'exécution propres. La fin de l'exécution du graphe aura lieu lorsque l'ensemble des prédécesseurs immédiats de v_n sera arrivé à complétion.

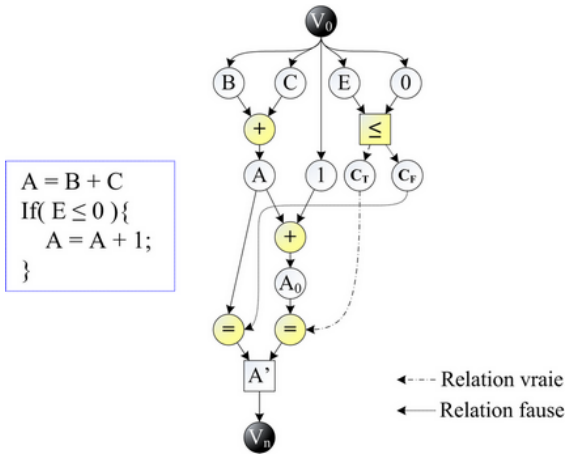

FIG. 3.21 – Exemple de modélisation d'une structure conditionnelle.

3.5.2 Modélisation d'une structure conditionnelle

Nous allons maintenant représenter une structure conditionnelle de type *if* (figure 3.21). Pour réaliser cette modélisation, un noeud de type *opération conditionnelle*, noté ($\leq$), ainsi qu'un noeud *variable conditionnée*, noté (*A'*), sont utilisés. Lors de l'exécution du graphe, après la validation des successeurs de v_0, les noeuds vont s'exécuter en fonction de leur règle d'exécution. En ce qui concerne *l'opération conditionnelle*, lors de son exécution, en fonction de la valeur du résultat de la condition, seul un de ses ports de sortie sera validé (celui menant à C_T correspond à la relation *vraie*, et celui menant à C_F correspond à la relation *fausse*). Cette validation sélective va avoir pour effet de ne permettre l'exécution que d'une opération parmi les 2 affectations possibles pour (*A'*). Au final, le noeud *variable conditionnée* A' a donc une et une seule entrée de validée, celle qui correspond à la branche conditionnelle exécutée.

Afin de trouver l'ensemble des opérations mutuellement exclusives, nous utiliserons des méthodes de remontée de chemins et de convergence de branches pour créer une liste contenant les opérations/variables qui appartiennent uniquement aux branches conditionnelles. Cela permet la modélisation de structures conditionnelles sans pour autant impliquer une exécution spéculative obligatoire de l'ensemble des noeuds comme cela se fait dans les représentations de type DFG.

3.5.3 Modélisation d'accès à des structures mémoires

La figure 3.22 représente un graphe modélisant une séquence d'accès déterministes et non déterministes à la mémoire. Ces accès à la structure de données, notée (T), et représentée par un noeud de structure dans notre exemple, sont réalisés par l'intermédiaire de noeuds de type opération. Ces noeuds dits d'adressage sont notés $@_r$ et $@_w$ respectivement pour la réalisation d'une lecture et d'une écriture dans la structure de données T.

Lors de la définition des noeuds de structure, nous avons précisé que nous procédions à un renommage de la structure pour chaque affectation (écriture), cela afin de lever/résoudre les problèmes de dépendance en appliquant sur les structures la méthode des affectations uniques. Dans notre exemple, lors de l'écriture

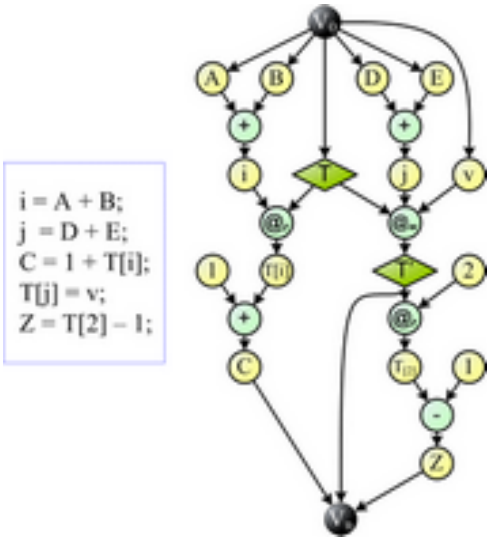

FIG. 3.22 – Exemple de graphe contenant des noeuds de structure.

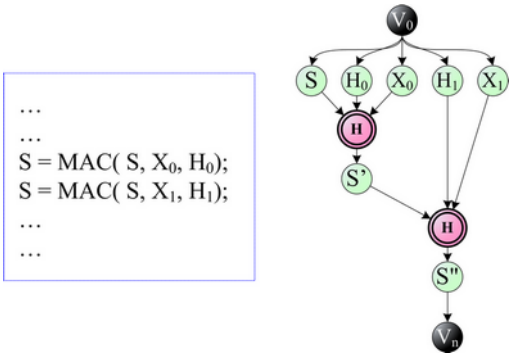

FIG. 3.23 – Exemple de graphe contenant des noeuds hiérarchiques.

dans la structure (T) à l'aide de l'opération *d'adressage en écriture* $@_w$, nous obtenons un nouveau noeud de structure nommé (T') qui contient les mêmes valeurs que (T) à $T[j]$ près.

3.5.4 Modélisation de composants "complexes"

La figure 3.23 présente l'utilisation de noeuds hiérarchiques à l'intérieur d'une représentation. Dans le cas présent nous considérons une partie d'un filtre de traitement du signal *FIR* dans lequel le concepteur a décidé d'utiliser des macro-blocs de type *MAC* pour réaliser les multiplications et accumulations effectuées sur les échantillons du filtre. Nous considérerons que l'opération MAC a été déclarée sous forme de fonction dans la description fonctionnelle du filtre. Cette fonction est représentée à l'aide du noeud hiérarchique H dans notre graphe. Ce noeud est modélisé comme un noeud de type opération nécessitant dans notre cas l'ensemble de ses entrées pour produire sa sortie.

Le sous-graphe interne modélisant les opérations réalisées dans le noeud hiérarchique H est présenté figure 3.24 (comportement de la fonction MAC). Il est intéressant de remarquer que l'entrée A du noeud

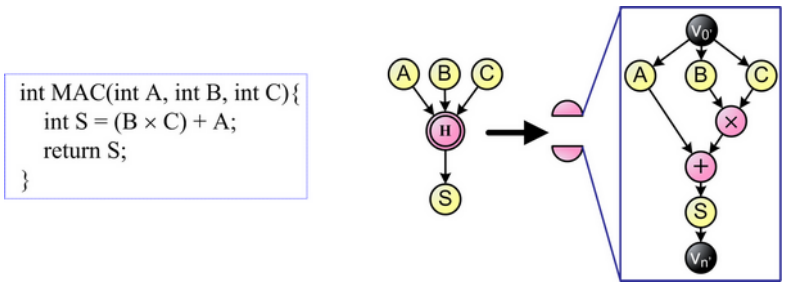

FIG. 3.24 – Contenu du noeud hiérarchique représentant une opération MAC.

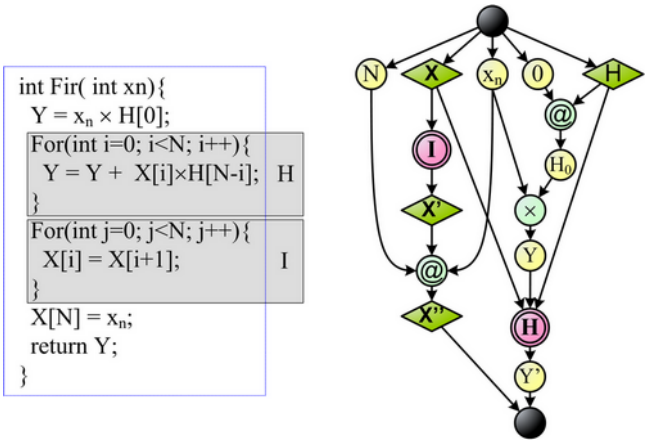

FIG. 3.25 – Modélisation d'une application contenant des boucles roulées.

hiérarchique ne peut être consommée au plus tôt qu'après la complétion de l'opération ($\times$). Ce délai entre le début de l'exécution du noeud hiérarchique (H) et la date d'exécution au plus tôt de (A) donne lieu à un offset de consommation entre A et le *noeud hiérarchique*. Dans le chapitre dédié aux expérimentations, cet exemple est repris et traité dans son intégralité afin de montrer l'intérêt de l'utilisation des offsets lors de la synthèse des noeuds hiérarchiques.

3.5.5 Modélisation d'une boucle non déroulée

La figure 3.25 présente un graphe spécifiant une application de type *FIR à N points*. Dans cet exemple, les noeuds hiérarchiques, noté (H) et (I), représentent des boucles non déroulées (ou partiellement déroulables). Ces dernières sont représentées sur fond gris dans la description comportementale associée. Dans ce cas, le graphe interne modélisant le comportement des noeuds hiérarchiques représente le déroulement d'une ou plusieurs itérations de la boucle (en cas de déroulage partiel).

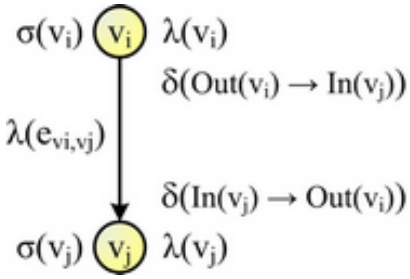

FIG. 3.26 – Ordonnancement d'un noeud possédant un seul prédécesseur.

3.6 Ordonnancement des noeuds du graphe

Afin de pouvoir projeter le modèle de représentation que nous venons de définir sur une architecture matérielle, il est nécessaire d'attribuer à chaque noeud une date d'exécution notée σ. Le problème d'ordonnancement d'un graphe G de type $CSFG$, est défini de la façon suivante :

Définition 3.6.1 (*Ordonnancement d'un graphe*)

L'ordonnancement d'un graphe de contrôle et de structure (CSFG) est un étiquetage entier $\sigma : V \rightarrow Z^+$ *d'un ensemble de noeuds* V *vers des entiers positifs* Z^+*, tel que* $\sigma(v_j) \geq \sigma(v_i) + \lambda(v_i \rightarrow v_j)$ *s'il existe un arc reliant* v_i *à* v_j *en considérant* $\lambda(v_i \rightarrow v_j)$ *comme étant la somme des poids. La somme des poids liant* v_i *à* v_j *est égale au temps d'exécution du noeud* v_i *associé aux offsets de production de* v_i *et de consommation de* v_j *(équation 3.38). Un ordonnancement minimum est un ordonnancement tel que* $(\sigma(v_i) - \sigma(v_0))$ *est minimum pour tous les* $v_i \in V$*.*

$$\lambda(v_i \rightarrow v_j) = \lambda(v_i) + \delta(Out(v_i)) + \lambda(e_{v_i,v_j}) + \delta(In(v_j)) \tag{3.38}$$

La définition ci-dessus est valable dans le cas où le noeud v_i est l'unique prédécesseur du noeud v_j (figure 3.26). Si nous considérons maintenant le cas où v_j possède plusieurs prédécesseurs (figure 3.39), et que $V_{pi} = Pred(v_j) \in V$ soit le sous-ensemble contenant l'ensemble des prédécesseurs du noeud v_j, la date d'exécution $\sigma(v_j)$ est alors :

$$\begin{aligned} \sigma(v_j) &\geq \max_{r \in Pred(v_j)} (\sigma(r) + \lambda(r \rightarrow v_j)) \\ avec\ \lambda(r \rightarrow v_j) &= \lambda(r) + \delta(Out(r)) + \lambda(e_{r,v_j}) + \delta(In(v_j)) \end{aligned} \tag{3.39}$$

La date d'exécution du noeud est fonction des dates d'exécution de l'ensemble des prédécesseurs de ce noeud. L'équation prend en considération les offsets de consommation et de production disponibles entre chaque couple *(noeud, prédécesseur)*.

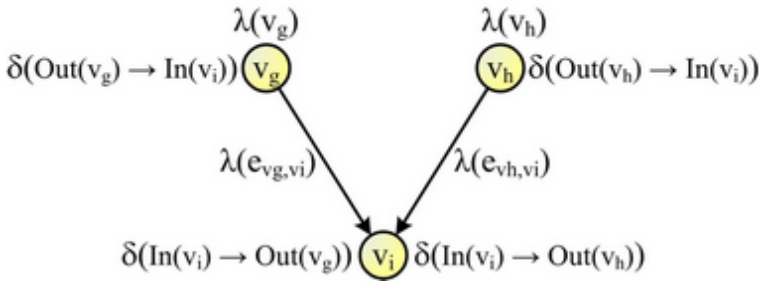

FIG. 3.27 – Ordonnancement d'un noeud possédant plusieurs prédécesseurs.

3.6.1 Datation des noeuds du graphe

Afin d'estimer la date d'exécution possible de chaque opération contenue dans le graphe, nous allons dater les opérations relativement les unes par rapport aux autres. Pour cela nous définirons pour chaque noeud sa date d'exécution au plus tôt (ASAP) notée $\sigma_{min}(v)$, et sa date d'exécution au plus tard (ALAP) qui sera notée $\sigma_{max}(v)$.

Date d'exécution au plus tôt (ASAP)

Si le noeud v_i est l'unique prédécesseur du noeud v_j, alors, on peut écrire que la date d'exécution au plus tôt (ASAP) du noeud v_j s'exprime tel que :

$$\sigma_{min}(v_j) \geq \sigma_{min}(v_i) + \lambda(v_i \rightarrow v_j) \tag{3.40}$$

Dans le cas ou v_j possède plusieurs prédécesseurs, la date d'exécution ASAP notée $\sigma_{min}(v_j)$ est alors fournie comme suit :

$$\sigma_{min}(v_j) \geq Max_{r \in Pred(v_j)}(\sigma_{min}(r) + \lambda(r \rightarrow v_j)) \tag{3.41}$$

Date d'exécution au plus tard (ALAP)

Si le noeud v_j est l'unique successeur du noeud v_i, alors on peut écrire que la date d'exécution au plus tard (ALAP) du noeud v_i s'exprime :

$$\sigma_{max}(v_i) \leq \sigma_{max}(v_j) - \lambda(v_i \rightarrow v_j) \tag{3.42}$$

Dans le cas où v_i possède plusieurs successeurs, la date d'exécution $\sigma_{max}(v_i)$ est alors fournie comme suit :

$$\sigma_{max}(v_i) \leq Min_{r \in Succ(v_i)}(\sigma_{max}(r) - \lambda(v_i \rightarrow r)) \tag{3.43}$$

3.6.2 Les offsets dans les noeuds hiérarchiques

Le noeud hiérarchique est constitué en interne d'un sous-graphe. Une fois daté, le noeud hiérarchique peut faire apparaître des mobilités sur ses entrées et ses sorties. Dans ce cas, il est intéressant de remonter cette information au niveau hiérarchique supérieur afin de pouvoir bénéficier du relâchement des contraintes ainsi généré lors du calcul des dates ASAP/ALAP des noeuds. Les offsets de production et de consommation sont des informations liées aux ports d'entrée et de sortie des noeuds hiérarchiques. Ces offsets permettent de prendre en considération la mobilité des entrées et des sorties au sein du sous-graphe interne au noeud hiérarchique. Nous considérons que le sous-graphe interne du noeud hiérarchique du noeud v_h est noté G'.

Définition 3.6.2 (*Offset de Consommation*)

Pour chaque entrée $In(v_h)$ du graphe G' il existe un offset $\delta(pe)$ tel que $\delta(pe)$ modélise le retard relatif acceptable pour l'entrée pe du noeud hiérarchique v_h en rapport à la date d'exécution de v_h.

Cet offset de consommation a un impact sur la date d'exécution au plus tôt du noeud v_h (ASAP) ainsi que sur la date d'exécution au plus tard (ALAP) des successeurs du noeud (H).

Définition 3.6.3 (*Offset de Production*)

Pour chaque port de sortie $Out(v_h)$ du graphe G' il existe un offset $\delta(ps)$ tel que $\delta(ps)$ modélise l'avance relative acceptable pour la sortie ps par rapport à la complétion du noeud v_h.

Cet offset de production a un impact sur la date minimum d'exécution des successeurs de v_h (ASAP) ainsi que sur la date d'exécution au plus tard du noeud v_h (ALAP).

Un exemple pédagogique qui met en application la méthode de calcul des offsets dans les noeuds hiérarchiques et montre l'intérêt de ces informations durant l'ordonnancement est présenté dans la partie "Exemples Pédagogiques" du chapitre 5.

3.7 Conclusion

Nous avons vu que, selon le type d'application à modéliser et les transformations à appliquer, les modèles de représentation utilisés diffèrent, en particulier, en ce qui concerne les applications de TDSI considérées dans nos travaux. L'usage de modèles différents pour représenter des boucles, des structures conditionnelles et des accès mémoires empêchent l'utilisation d'un flot unifié permettant de couvrir ces différentes primitives.

Pour remédier à cet inconvénient, nous avons défini dans ce chapitre un modèle de représentation nommé CSFG. Ce modèle permet de représenter des sémantiques algorithmiques de contrôle et de données : structures conditionnelles, boucles, hiérarchie ainsi qu'accès indéterministes à la mémoire. Le modèle permet la modélisation d'applications de TDSI complexes couramment utilisées.

Nous allons dans le prochain chapitre présenter le processus de synthèse qui consiste à transformer la spécification en une implémentation. Ce processus se base sur des modèles formels et des transformations prouvées. Nous présentons également le modèle d'implémentation générique ciblé par notre flot de

synthèse.

Chapitre 4

Modèle Architectural et Synthèse de Haut Niveau

4.1 Introduction

Si le processus de synthèse consiste à transformer une spécification en une implémentation, ce processus, pour être fiable, doit se baser sur des modèles formels et des transformations prouvées. Après avoir présenté le modèle de représentation que nous avons développé, nous allons dans ce chapitre présenter le modèle d'implémentation générique ciblé par notre flot de synthèse. L'architecture cible est constituée de 3 unités fonctionnelles distinctes : l'unité de traitement *(UT)*, l'unité de mémorisation *(UMem)* et l'unité de communication *(UCom)*. L'ensemble de ces unités forme, après synthèse, un composant de niveau RTL dérivé de la description algorithmique et respectant les contraintes d'intégration. Le flot de conception de ce composant s'appuie comme nous le verrons par la suite sur un ensemble de transformations successives qui raffinent le modèle algorithmique en une description structurelle. L'approche de la conception du composant virtuel repose sur trois étapes que sont la synthèse sous contraintes de l'unité de traitement, la synthèse sous contraintes de l'unité mémoire et la synthèse sous contraintes de l'unité de communication. Dans ce document, nous étudierons plus particulièrement les deux premiers points, la synthèse de l'unité de communication ayant déjà été déjà traitée dans la thèse de P. Coussy [Cous03]. Notons également que la synthèse particulière de l'UT et de l'UMem *sous contrainte de placement des données en mémoire* a été considérée dans la thèse de G. Corre [Corr05].

4.2 Etat initial de l'outil GAUT

GAUT (historiquement acronyme de Générateur Automatique d'Unité de Traitement) est un environnement de synthèse d'architecture matérielle, dédié aux algorithmes de Traitement du Signal et de l'Image (TDSI) sous contrainte de cadence d'itération. A partir de la spécification d'un algorithme C (ou VHDL), il génère une description structurelle VHDL de niveau RTL optimisée en surface et destinée aux outils de synthèse logique du marché tels ISE/Foundation de Xilinx, Quartus d'Altera, ou Design Compiler de

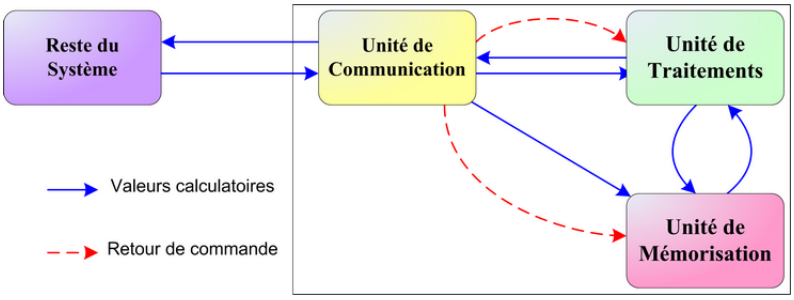

FIG. 4.1 – Spécification des communications entre les différentes unités.

Synopsys. Cet outil universitaire résulte de travaux de recherche commencés au LASTI dans les années 1990 et poursuivis au LESTER depuis 1994 [Mar93]. Nous allons tout d'abord présenter les différentes techniques employées dans l'outil afin de pouvoir ensuite évoquer les modifications que nous avons dues apporter pour pouvoir gérer les structures itératives, les structures conditionnelles ainsi que les accès indéterministes à la mémoire. Nous allons dans un premier temps aborder l'architecture générique ciblée par l'outil, puis nous présentons les contraintes supportées par le processus de synthèse avant de présenter les transformations raffinant automatiquement la description algorithmique.

4.2.1 Présentation de l'architecture cible

L'architecture des composants que nous considérons est constituée, comme indiqué sur la figure 4.1, de trois unités fonctionnelles : une unité de communication (UCom), une unité de mémorisation (UMem) et une unité de traitement (UT). Chaque unité possède son propre module de contrôle.

Les canaux de communication situés entre les différentes unités permettent le transfert de valeurs calculatoires entre ces dernières. Un canal de retour de commande unidirectionnel, orienté de l'UCom vers les autres unités, permet à cette dernière de stopper l'exécution des calculs si les E/S (reste du système) ne sont pas disponibles/consommées. L'architecture générique ciblée par l'outil est présentée en figure 4.2.

L'unité de traitement (UT)

L'unité de traitement implémente la partie calculatoire de l'algorithme en utilisant des cellules élémentaires qui réalisent les opérations arithmétiques ou logiques de la spécification algorithmique. Chaque cellule est composée d'un opérateur, d'un ensemble de registres et de glue logique (multiplexeurs, démultiplexeurs et buffers trois états). Les registres permettent le stockage temporaire des données et la synchronisation des transferts de données au sein de l'UT. Les multiplexeurs, démultiplexeurs et buffers trois états ont en charge l'aiguillage des données. Leur présence optionnelle dans une cellule résulte du partage temporel des registres et des opérateurs. Les cellules communiquent soit directement, soit au travers du réseau de bus qui interconnecte les différentes unités (figure 4.2). L'unité de contrôle qui pilote l'UT est constituée d'une machine d'états finis linéaire qui résulte de l'ordonnancement. L'étape d'ordonnancement affecte une date d'exécution, et de façon équivalente une étape de contrôle, à chacune

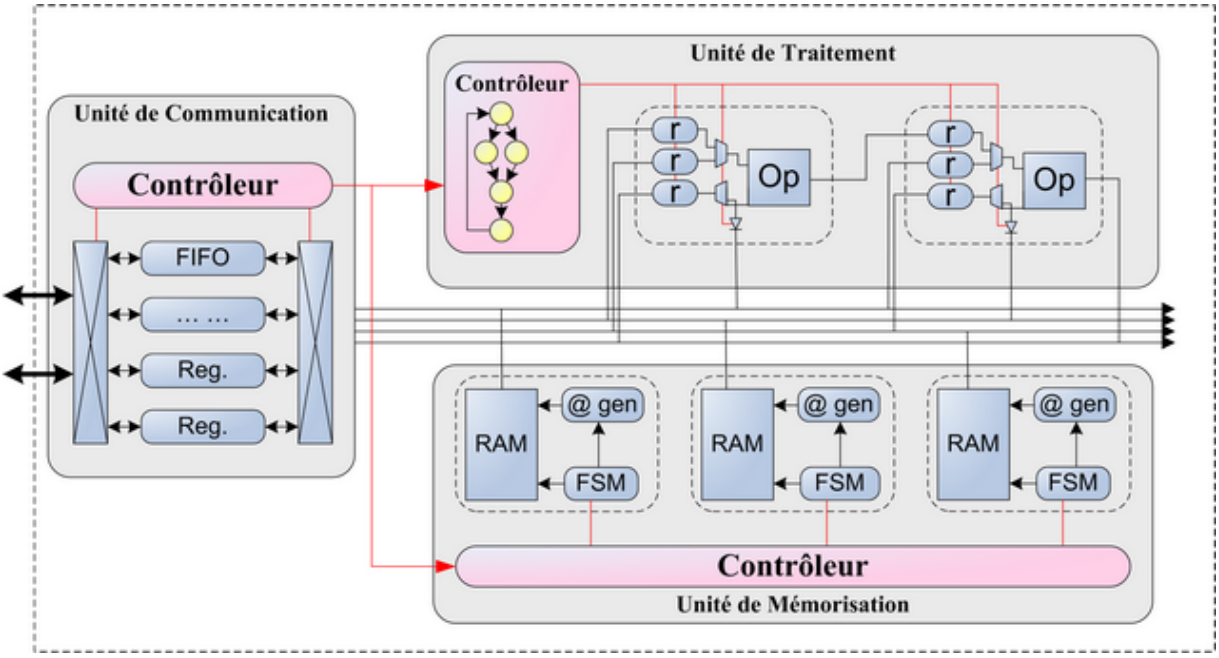

FIG. 4.2 – Modèle architectural ciblé par l'outil GAUT.

des opérations en tenant compte d'une part des dépendances de données et d'autre part des contraintes d'E/S et de placement mémoire (mapping).

L'unité de communication (UCom)

L'unité de communication contient des éléments mémorisants (FIFO, LIFO, registres), et leurs structures de contrôle associées, qui sont dédiées à chaque port d'E/S et ce en fonction des types des transferts (lecture ou écriture). L'ensemble des *FSMs* (une par port) implémente le protocole d'échange de données spécifié par le concepteur du *SoC*. Une autre machine à états finis réalise la synchronisation entre les unités de traitements et de mémorisation et l'unité de communication [Bome04] (l'architecture est globalement asynchrone et globalement synchrone (GALS)). Des travaux concernant l'UCom sont actuellement en cours au LESTER ; ils font partie des travaux de thèse de C. Chavet.

L'unité de mémorisation (UMem)

L'unité de mémorisation contient des éléments mémorisants (bancs mémoire) et leurs structures de contrôle associées (générateurs d'adresses). Les générateurs d'adresses associés à chacun des bancs mémoire ont pour but de générer les séquences d'adresses et de signaux de commande pour assurer les lectures et écritures nécessaire à l'UT de manière parfaitement synchrone [Corr05]. Le contrôleur de l'unité de mémorisation est constitué d'une machine à états finis linéaire qui résulte de l'ordonnancement des accès aux différents bancs mémoire.

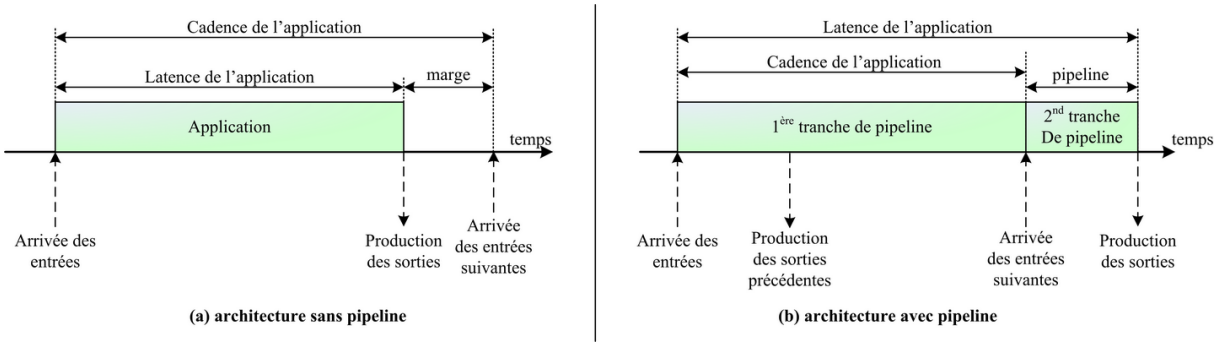

FIG. 4.3 – Explications relatives aux architectures pipeline.

Limitations

En l'absence de retour d'état au sein de l'UT, l'architecture présentée est dédiée à l'implémentation d'applications déterministes avant synthèse. Dans le cas de structures conditionnelles et de boucles possédant des conditions d'exécution/d'arrêt non déterministes a priori, il faut alors procéder à des exécutions spéculatives de l'ensemble des calculs. Cela implique par effet de bord la nécessité de fournir des données depuis l'unité de mémorisation, ce qui n'autorise pas un arrêt de certaines unités fonctionnelles permettant de réduire la consommation d'énergie de l'architecture quand cela est possible. Le modèle fonctionnel ciblé est synchrone pour les accès aux données et la réalisation des calculs, cela ne permet pas de réaliser des accès indéterministes à la mémoire.

4.2.2 Contraintes supportées par le processus de synthèse

L'outil de synthèse comportemental GAUT cible les applications TDSI. Les contraintes et les optimisations que supporte le flot de raffinement automatique sont de différentes natures : temporelle (cadence), mémoire (mapping), date d'arrivée/production des E/S, surface, consommation ...

Contrainte temporelle

L'outil GAUT cible les applications TDSI sous contrainte temps réel. Les applications sont exprimées sous contrainte de cadence d'itération. Cette valeur notée $T_{cadence}$ correspond à la plage temporelle qui sépare l'arrivée des données nécessaires à deux itérations successives de l'algorithme. Le temps d'exécution de l'algorithme avant implémentation correspond à la latence minimum de l'application (le chemin critique est noté $T_{Latence}$. En fonction du couple $(T_{Cadence}, T_{Latence})$, l'architecture générée après synthèse sera ou non de type pipeline en fonction de la relation liant ce couple de valeur.

Lorsque la contrainte de cadence liée à l'application est supérieure à la durée de la latence (avant et après synthèse), cela permet la génération d'une architecture où la complétion de l'application (production des sorties) est réalisée avant l'arrivé d'un nouveau jeu d'entrées (fig. 4.3a).

Lorsque la cadence de l'architecture est inférieure à la latence de l'architecture comme cela est représenté

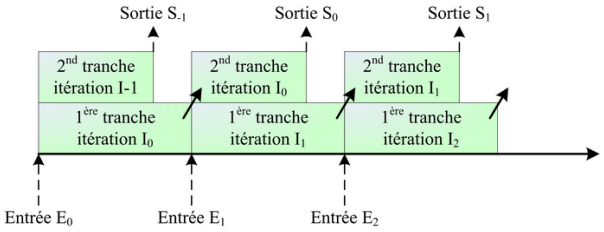

FIG. 4.4 – Explications relatives aux architectures pipeline, exécution des tranches.

dans la figure 4.3b, dans ce cas, l'architecture générée sera de type pipeline pour satisfaire la contrainte temporelle spécifiée par le concepteur. Cette architecture pipeline sera composée d'un ensemble d'unités de traitement dédié de manière exclusive à chacune des tranches. A la fin de chaque temps cadence, les données sont vieillies, passant de la tranche n à la tranche $n+1$ jusqu'à arriver à la dernière tranche p où les dernières sorties sont produites. Ces propriétés sont illustrées par la figure 4.4.

Le nombre de tranches de pipeline qui constituent une application peut se calculer à l'aide de l'équation (4.1).

$$Nbtranches = \left\lceil \frac{T_{Latence}}{T_{Cadence}} \right\rceil \tag{4.1}$$

Cette équation permet d'estimer le nombre de tranches que contiendra l'architecture avant la phase d'allocation et après la phase d'ordonnancement.

Contraintes d'entrées/sorties

Les problèmes d'ordonnancement sous contrainte temporelle sont généralement définis et résolus pour des graphes soumis à des dates d'arrivée (et de production) des données figées. Suite aux travaux présentés dans [Cous03] le flot de synthèse d'architecture intègre aussi la gestion des contraintes d'entrée/sortie étendue aux graphes soumis à des dates de production variables et bornées des E/S. La gestion des dates d'entrée/sortie de l'application permet à l'architecture générée d'être intégrée au sein du système sans nécessiter l'utilisation de wrappers de protocole et/ou d'entrelaceurs.

Les échanges de données entre l'environnement et le composant virtuel sont modélisés par un graphe de contrainte d'entrée/sortie. L'ensemble des noeuds représente des transferts de données et l'ensemble d'arcs représente le séquencement des transferts. Les poids associés aux arcs représentent les contraintes temporelles entre les transferts de données. L'indéterminisme d'une date de transfert est modélisé à l'aide de contraintes temporelles représentant la borne inférieure et supérieure entre les dates d'exécution de deux opérations de transfert. Ce type de contraintes permet aussi la représentation des caractéristiques des modes de communication : rafale, série ...

La modélisation des caractéristiques de l'architecture de communication (transfert série, parallèle...) est supportée par le graphe *IOCG* [Cous05b] dans lequel ces caractéristiques sont exprimées à l'aide de noeuds hiérarchiques. La figure 4.5 décrit le séquencement d'un transfert en rafale d'un vecteur de données A, et du transfert d'un scalaire b. Les caractéristiques des transferts tel que le nombre de cycles

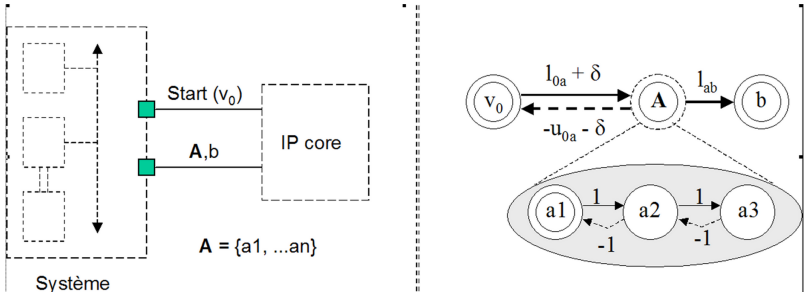

FIG. 4.5 – Exemple de modélisation d'un transfert rafale.

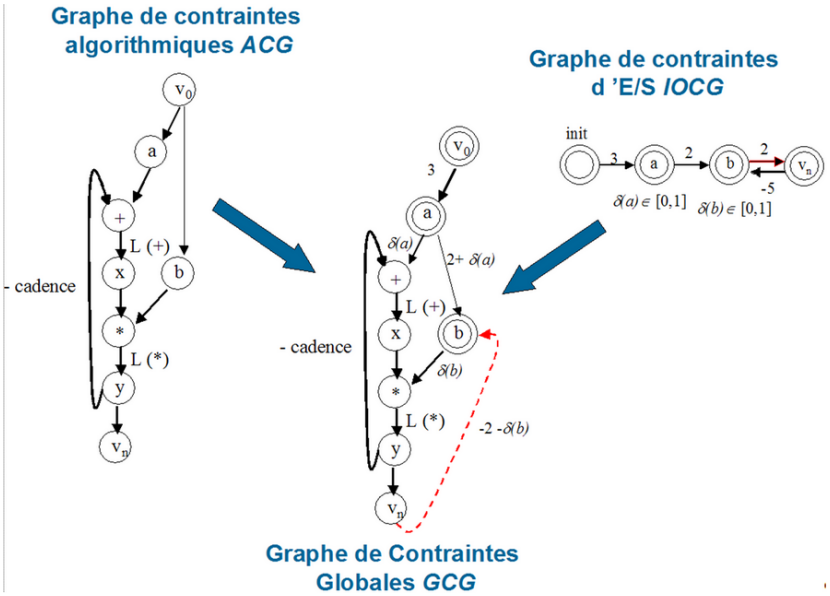

FIG. 4.6 – Fusion des graphes pour vérifier la cohérence des contraintes.

séparant deux données consécutives dans une rafale sont introduites dans les noeuds hiérarchiques à l'aide d'arcs avant et arrière entre les noeuds de données (a_1, a_2, a_3).

La figure 4.6 schématise l'obtention du graphe de contraintes global *GCG*. Le graphe *GCG* est composé à partir du modèle de représentation algorithmique de l'application et du graphe de contraintes des entrées/sorties

Après avoir vérifié la faisabilité de la synthèse du graphe de contraintes globales, l'outil de synthèse va contraindre le processus de synthèse afin de respecter les contraintes liées aux entrées/sorties. Ces contraintes vont provoquer la modification du processus de synthèse en passant d'une contrainte de cadence sur l'algorithme à une contrainte de cadence sur l'algorithme et de latence sur les E/S contraintes. De plus amples informations sur le mode de synthèse contraint par les E/S figurent dans [Cous03].

Contraintes de mémorisation

Une méthodologie de gestion des unités de mémorisation (placement des données dans les bancs mémoire) intégrée au flot de synthèse d'architecture est présentée dans [Corr05]. Le placement des données peut jouer un rôle important lors de la synthèse comportementale [Corr03b] ; en fonction du placement des données en mémoire, l'outil GAUT peut ou non exploiter le parallélisme d'accès aux données afin de réduire la latence de l'architecture et réduire par là même l'utilisation de coûteuses tranches de pipeline.

Ainsi, à partir des contraintes de placement mémoire, une approches a été développée afin d'améliorer la gestion de la mémorisation : d'une part, une analyse des données manipulées dans les applications de traitement du signal a permis de proposer un nouveau mécanisme de placement des données et de génération d'adresses ; d'autre part, une nouvelle gestion des accès mémoire basée sur le modèle de gestion de production kanban a été mise en oeuvre. Cette nouvelle gestion permet de répondre au problème lié à la violation des contraintes temporelles et fonctionnelles (limitation du nombre d'accès) qui peut être engendrée par l'introduction de contraintes mémoires en utilisant des mécanismes de lecture spéculative.

La synthèse sous contrainte mémoire repose sur une approche formelle de la spécification de contraintes d'accès aux données sur une optimisation temporelle de leur accès et sur la génération de l'unité de mémorisation. Un modèle de représentation formel est utilisé et permet de représenter les contraintes fonctionnelles d'accès à la mémoire à partir d'une spécification algorithmique et de ses caractéristiques technologiques. Pour cela, un graphe de contraintes mémoire est construit. Il représente les conflits d'accès aux données et exprime les possibilités d'ordonnancement des opérations dont les opérandes sont placés en mémoire. Un Graphe de Contraintes Mémoire *(MCG : Memory Constraint Graph)* est construit à partir du SFG, de la distribution de données en mémoire, et des contraintes technologiques de la mémoire. Ce graphe est utilisé durant la phase d'ordonnancement de la synthèse d'architecture.

L'approche proposée intègre la gestion des données vieillissantes pour une distribution des données sur plusieurs bancs mémoire. La gestion des vecteurs vieillissants lors de la synthèse repose sur un modèle de graphe permettant de modéliser les séquences d'accès aux données sur les itérations successives de l'algorithme. Une projection du modèle est ensuite effectuée sur une architecture mémoire générique (figure 4.7). La gestion du vieillissement permet la génération d'architectures mémoire génériques fonctionnelles intégrant le parallélisme d'accès aux données d'un vecteur vieillissant.

La gestion des accès à la mémoire à l'aide de la méthode kanban vise à réguler le flot d'accès mémoire en anticipant les besoins avant que l'ordonnancement des opérations de l'unité de traitement ne l'impose. La méthode est inspirée de la gestion de production par flux tiré [Corr05b]. Ce principe a été adapté de manière à gérer le stock de données lues et de données écrites et à contrôler le nombre de registres utilisés dans l'unité de traitement pour stocker les données traitées. L'anticipation des besoins permet également de conserver une distribution simple des données en mémoire pour des contraintes temporelles fortes.

Techniques d'optimisations

Durant le processus de synthèse architecturale, différentes techniques sont employées afin d'optimiser l'architecture générée. Il est par exemple possible de demander une optimisation visant à réduire la

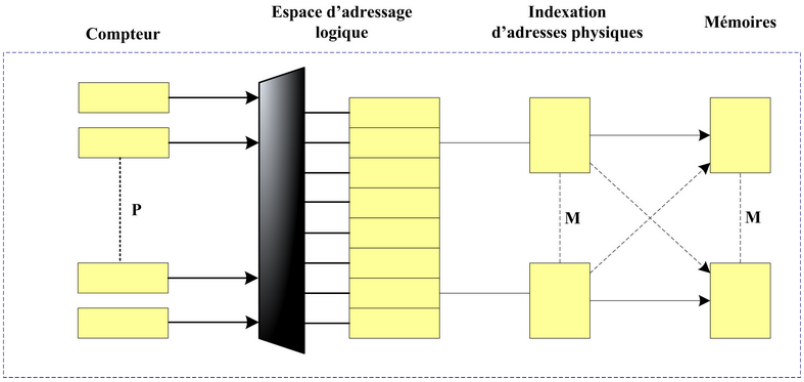

FIG. 4.7 – Modèle architectural générique de l'unité de mémorisation.

consommation globale de l'architecture, par utilisation de techniques *low-power* lors des étapes de sélection et d'ordonnancement/assignation des opérations (réduction des transitions [Gail98]). Suivant les contraintes du concepteur, une technique de réduction de la surface de l'architecture peut aussi être appliquée afin de fusionner les registres (partage global des registres (sur l'ensemble de l'architecture) ou local (interne aux cellules élémentaires de calcul)). L'optimisation du nombre de registres peut être complétée par l'optimisation du nombre de bus reliant les différentes unités de l'architecture. Un partage des bus est alors effectué moyennant l'utilisation de multiplexeurs et de démultiplexeurs. Parmi les autres optimisations possibles, on peut citer les travaux de C. Jego visant à limiter le coût des interconnections lors de la synthèse dans les technologies VDSM où le coût des interconnections devient prépondérants [Jego99].

4.2.3 Le processus de synthèse

Le flot de conception mis en oeuvre dans GAUT peut être dans son principe décrit par la figure 4.8. La description comportementale est décrite en langage de haut niveau (C ou VHDL). Cette description initiale est accompagnée d'un ensemble de contraintes (cadence, technologie, mémoire, E/S, ...). La phase de compilation effectue une analyse syntaxique, un contrôle sémantique et une parallèlisation du code. Cette étape réalise (1) la suppression des dépendances de contrôle (déroulage des boucles, mise en ligne des appels de procédures, parallèlisation des branchements conditionnels) et (2) la suppression des fausses dépendances de données. En raison de l'assignation unique, la parallèlisation du code se termine par un renommage des variables.

La compilation produit une représentation interne de l'algorithme sous la forme d'un graphe flot de signaux (SFG). Cette modélisation permet l'expression du parallélisme maximal de l'algorithme tel qu'il est décrit. Après l'étape de synthèse [Mar92] de l'unité de traitement dont les différentes étapes vont être décrites, l'unité de contrôle et de mémorisation sont générées. Afin de garantir la compatibilité avec le plus grand nombre d'outils de synthèse RTL du commerce, le VHDL RTL produit par GAUT est strictement conforme à la norme IEEE P1076 *(Standard for VHDL Register Transfer Level Synthesis)*.

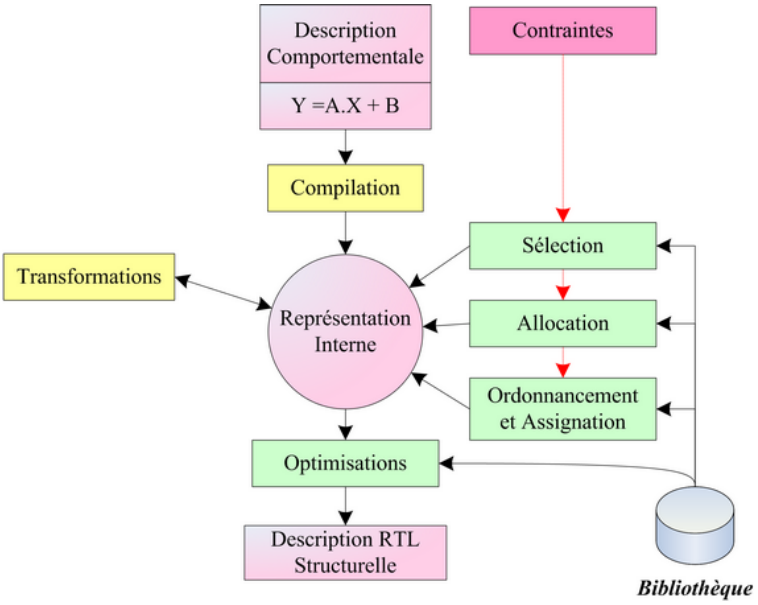

FIG. 4.8 – Flot de synthèse actuel de l'outil GAUT.

Sélection des Opérateurs

La synthèse de l'unité de traitement débute par la sélection des opérateurs. Cette phase permet d'associer des délais T_{fonc} aux opérations contenues dans la représentation interne. Ces délais établis à partir d'une bibliothèque de composants de niveau RTL préalablement caractérisée, sont ajustés pour être des multiples entiers de la fréquence d'horloge notée $phase_{min}$. La mobilité des opérations est ensuite calculée en utilisant les dates au plus tôt $(ASAP)$ et au plus tard $(ALAP)$ [Gajs92] de chaque noeud du graphe.

Allocation des Opérateurs

L'allocation dans GAUT consiste à allouer des opérateurs sur chaque tranche de pipeline : le nombre d'opérations de chaque type est donc comptabilisé par tranche. La complexité de la phase d'allocation est en $O(n)$, où n représente le nombre d'opérations à traiter. Le nombre de tranches de pipeline noté Tranche est défini par : $Tranche = T_{Latence}/T_{Cadence}$. Le nombre moyen d'opérateurs par tranche est calculé comme suit [Mar92] :

$$Nb\_opr(f,t) = \frac{Nb\_ops(f,t) * tps\_fonct(f)}{Tcadence} \forall \begin{array}{c} 0 < t < tranche - 1 \\ 1 < f < Nb\_fonc\_Max \end{array} \quad (4.2)$$

où $Nb_{opr}(f,t)$ est le nombre d'opérateurs de type f alloués à la tranche t, $tps_{fonct}(f)$ le temps de traversée des opérateurs réalisant la fonction f et $T_{cadence}$ la contrainte de cadence fixée par l'utilisateur. Si cette allocation est respectée durant l'ordonnancement, alors la meilleure solution en terme de surface est obtenue. La technique d'allocation, basée sur le nombre moyen d'opérateurs, est adaptée aux synthèses contraintes par les ressources matérielles. A la fin de l'étape d'allocation, l'outil de synthèse

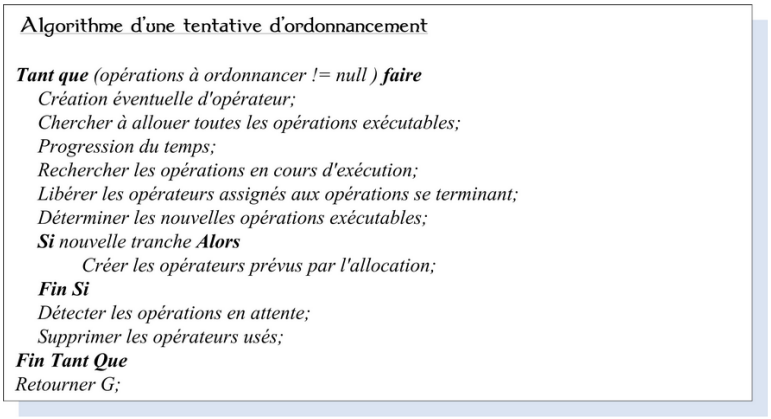

FIG. 4.9 – Algorithme d'une tentative d'ordonnancement.

tente d'optimiser les opérateurs alloués en regroupant certains d'entre eux afin d'allouer des opérateurs multifonctions en vue de minimiser la surface occupée.

Ordonnancement et Assignation

Un des objectifs de l'ordonnancement tel qu'il est implanté dans GAUT est de réguler le parallélisme du *Graphe Flot de Signaux* au parallélisme moyen de l'architecture (cf allocation) tout en limitant l'élongation du graphe à l'aide d'algorithmes de faibles complexités afin de pouvoir traiter des applications complexes. En effet, plus l'allongement est important plus le nombre de tranches, et donc le nombre d'opérateurs, est important. L'algorithme utilisé, de type *Static List Scheduling* [Jain91], est basé sur la gestion des priorités des opérations à exécuter et permet de maîtriser l'élongation du graphe. Un autre objectif de l'ordonnancement est d'optimiser l'utilisation des opérateurs. Pour cela, une priorité d'utilisation est associée aux opérateurs déjà alloués/utilisés qui sont ainsi réutilisés dès leur libération. Les données lues sur les ports d'entrées sont supposées être présentes aux dates assignées par l'ordonnancement, ou, lorsque la synthèse est réalisée sous contraintes d'E/S, aux dates fixées par le modèle de contraintes. L'algorithme (fig. 4.9) décrit une tentative d'ordonnancement. Il est important de noter qu'avant de commencer la phase d'assignation, il existe une phase de création éventuelle d'opérateurs n'ayant pas été prévus par l'allocation. En effet, certaines opérations ne peuvent être allouées dans la tranche de pipeline où elles étaient initialement prévues. On observe dans ce cas une élongation du graphe qui nécessite la création d'opérateurs supplémentaires pour traiter ce parallélisme dur.

Génération matérielle

Après l'étape d'ordonnancement/assignation, les techniques d'optimisation de l'architecture précédemment citées sont mises en oeuvre afin de réduire la surface nécessaire à l'architecture. Une fois cela réalisé, les différentes unités sont générées au niveau RTL.

4.2.4 Conclusion

Le flot de synthèse de l'outil GAUT dans son état initial permet la synthèse d'applications de TDSI sous contraintes temps réel. Les applications traitées avec cette version de l'outil sont nécessairement déterministes. Dans le cas de branches conditionnelles, des transformations sont appliquées afin de résoudre l'indéterminisme. Pour l'ensemble des autres applications ne pouvant se traduire sous la forme d'un graphe flot de données statique, la synthèse n'est pas possible.

4.3 Modifications nécessaires de l'architecture

Nous allons maintenant étudier le modèle d'implémentation associé afin de mettre en oeuvre l'ensemble des sémantiques du CSFG. Nous commencerons notre analyse par les branches conditionnelles, puis nous poursuivrons par les accès indéterministes à la mémoire et nous conclurons par les boucles roulées bornées ou non bornées qui tirent partie des 2 premières analyses.

4.3.1 Gestion des opérations conditionnelles

En ce qui concerne l'implémentation des opérations conditionnées, plusieurs méthodes sont envisageables. Actuellement, l'architecture ciblée par GAUT réalise une implémentation spéculative de l'ensemble des opérations présentes dans les structures conditionnelles. Pour cela, toutes les opérations et les transferts de données associées sont exécutées de manière spéculative. Les résultats adéquats sont ensuite sélectionnés grâce à un opérateur de sélection (EQMUX, ...). La technique actuelle réduit la latence de l'application, mais interdit l'utilisation du partage d'opérateurs entre branches mutuellement exclusives (en vue d'une réduction de la surface et de la consommation).

Nous allons distinguer 2 types de primitives architecturales en fonction du contenu des branches conditionnelles : les structures conditionnelles dites *locales* ont des répercussions se limitant à l'UT (tout les accès aux données sont déterministes) et les structures conditionnelles dites *globales* ont des répercussions impliquant l'UMem et/ou l'UCom (les accès aux données sont indéterministes). Une analyse de la localité d'implémentation des opérations et de données comprises dans les structures conditionnelles permet de déterminer la classe de cette dernière (si toutes les opérations et les données impliquées sont implémentées dans l'UT alors la structure conditionnelle est *locale* sinon elle est *globale*).

Solution locale - Afin d'implémenter des techniques permettant de partager les opérateurs entre plusieurs opérations mutuellement exclusives, il est nécessaire de posséder une liaison entre le chemin de données et le contrôleur du chemin de données. Cette liaison doit servir à transmettre les retours d'états qui ont été calculés dans le chemin de données vers le contrôleur afin que ce dernier gèle les opérations qui n'appartiennent pas aux branches conditionnelles exécutées. Le retour d'états entre le chemin de données et son contrôleur permet d'implémenter toutes les structures conditionnelles locales dont les opérandes et les résultats sont produits et consommés par l'unité de traitement.

Solution globale - Cette extension permet de gérer (geler/réaliser) les accès indéterministes aux données de/vers l'UMem et/ou l'UCom. Cette initiative permet une réduction substantielle du nombre de commutations sur les bus (ainsi que le nombre de bus nécessaires). Pour implémenter cette solution, il est nécessaire de posséder un système de communication inter-unités pour transmettre le résultat des conditions calculées dans l'UT vers l'ensemble des unités qui sont concernées par ces résultats.

Perspective

Afin de réaliser un système de communication inter-unités, il est possible d'utiliser le réseau de bus existant. L'utilisation des bus de données actuels pour le transfert de retour d'états est envisageable. Cette solution architecturale nécessite toutefois une connexion liant les différents contrôleurs aux bus de données susceptibles de leur transmettre des retours d'états.

4.3.2 Gestion des Accès Indéterministes

L'architecture initiale permet uniquement l'implémentation d'applications contenant des séquences d'accès déterministes à la mémoire. Afin prendre en considération les accès indéterministes à la mémoire (adressages dynamiques, accès conditionnels, nombre d'accès indéterminé), il est nécessaire d'ajouter au modèle d'architectural des séquenceurs un mécanisme permettant de transférer les adresses depuis les opérateurs où elles ont été calculées (dans l'UT) vers les générateurs d'adresses des bancs mémoire concernés par les accès. Ces transferts d'adresses peuvent être modélisés comme des transferts de données. De plus, les transferts de données (et les accès mémoire associés) doivent pouvoir être interrompus lorsque ces derniers sont rendus inutiles lors de la non exécution d'une structure conditionnelle (condition non validée) afin d'économiser de l'énergie (réduction des commutations sur les bus, des accès mémoire).

En résumé, nous devons apporter 2 changements majeurs dans les séquenceurs d'accès à la mémoire : (1) la prise en compte des adresses calculées dans l'UT à la place de celles générées par les générateurs d'adresses, (2) la possibilité de geler une partie des accès à la mémoire et des transferts de données si un retour d'états en donne l'ordre.

4.3.3 Gestion des boucles non déterministes

La gestion des boucles roulées déterministes ou non déterministes est fortement liée à l'implémentation des primitives fonctionnelles que nous venons d'aborder. Il est nécessaire afin d'implémenter des boucles roulées de pouvoir transférer des retours d'états vers le ou les contrôleurs nécessitant l'information. La différence par rapport à la gestion des branches conditionnelles réside dans l'impact du retour d'états sur les contrôleurs ciblés. De la même manière que pour les conditions, nous pouvons distinguer 2 types de boucles : les boucles *locales* (impliquant uniquement l'UT) et les boucles *globales* (qui impliquent aussi l'UMem et/ou l'UCom). Une analyse de la localité d'implémentation des opérations et de données comprises dans les boucles permet de déterminer la classe de cette dernière.

4.3.4 Bilan

Nous venons d'avancer trois modifications à apporter à l'architecture cible afin de permettre l'implémentation des primitives algorithmiques supportées par le modèle CSFG :

1. *Le réseau de communication intra-unité et inter-unités :* afin de permettre à l'UMem et/ou UCom de geler certains transferts devenus inutiles avec le contexte applicatif, il est nécessaire de posséder un réseau de communication entre les unités pour remonter les informations liées aux retours d'états (branches conditionnelles et boucles non déterministes). Ce réseau de communication doit aussi permettre de geler les opérations implémentées dans l'UT.
2. *Le type des contrôleurs :* actuellement les contrôleurs des différentes unités sont implémentés sous la forme de machines à états linéaires. Ces machines d'états synchrones ne permettent pas la prise de décisions en fonction de résultats conditionnels. Il faut donc les remplacer par des contrôleurs non linéaires permettant les prises de décisions conditionnelles (branches conditionnelles, boucles).
3. *Les capacités des séquenceurs mémoire :* ces derniers doivent être capables de réaliser des accès indéterministes à partir des adresses qui sont calculées dans l'UT. Ils doivent aussi permettre de stopper toutes les lectures/écritures si ces dernières sont conditionnées et que leurs conditions d'exécution ne sont pas validées.

4.3.5 Architecture cible proposée

La nouvelle architecture cible est décomposée comme précédemment sous forme de trois unités (fig. 4.11). Un schéma récapitulant les besoins en terme de communication inter-unités est présenté dans la figure 4.10.

Des canaux de communication relient maintenant les contrôleurs au chemin de données. Ces connexions permettent l'utilisation des résultats des opérations conditionnelles pour gérer les structures conditionnelles (partage d'opérateurs et/ou gèle des opérations inutiles) et les boucles non déterministes.

La présence de retours d'états vers les différents contrôleurs de notre architecture implique une modification du modèle des contrôleurs qui pour l'instant sont linaires (et donc inadéquats pour prendre des décisions en fonction des entrées). Les contrôleurs des différentes unités sont maintenant basés sur une implémentation de type *FSM de Moore* permettant la prise en considération des entrées (retours d'états provenant de l'UT) afin de définir la valeur du prochain état. Les différentes unités composant l'architecture restent localement synchrones et globalement synchrones car leurs communications sont "pilotées" par la même horloge. Nous allons maintenant étudier les avantages et les inconvénients à posséder n retours d'états possibles en parallèle.

Contrôleurs à 2 **états fils (**1 **retour par cycle) -** Dans le cas d'une implémentation d'un contrôleur de type *Moore* avec au maximum 2 fils par état, le besoin en terme de communication entre le chemin de données et le contrôleur est limité à 1 fil de 1 bit (le résultat étant de type booléen : *vrai* ou $faux$). La ressource à mettre en oeuvre afin de transmettre les résultats conditionnels est limitée, mais la latence de

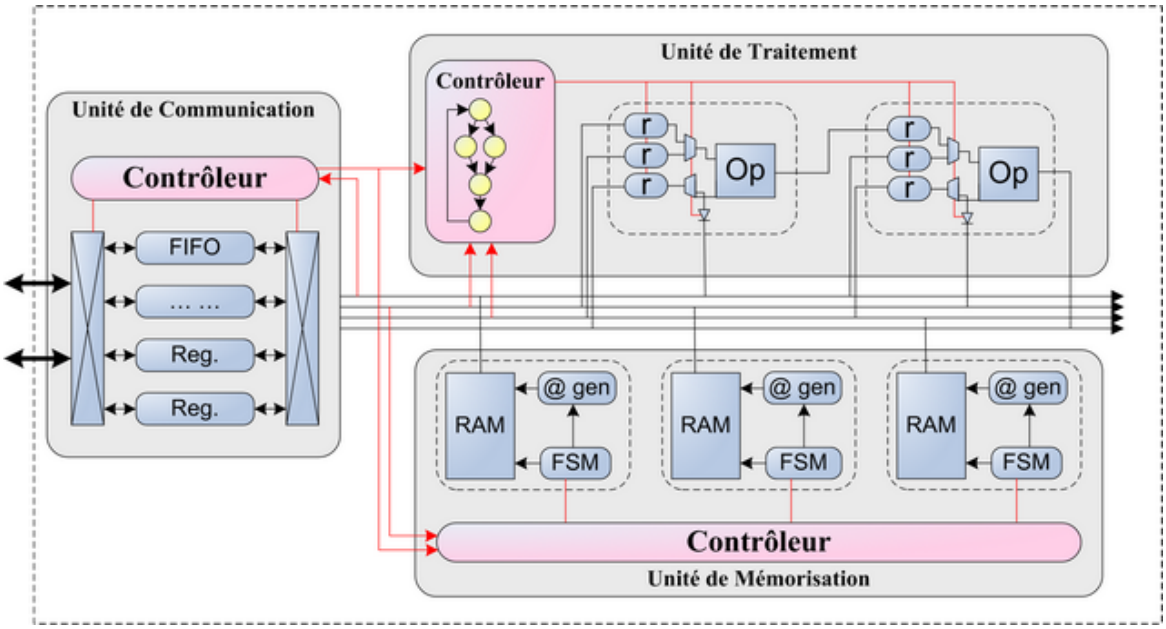

FIG. 4.10 – Architecture cible proposée.

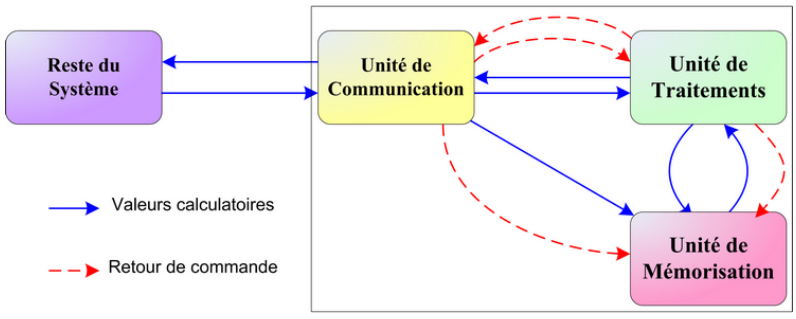

FIG. 4.11 – Besoin en terme de moyens de communication dans l'architecture cible proposée.

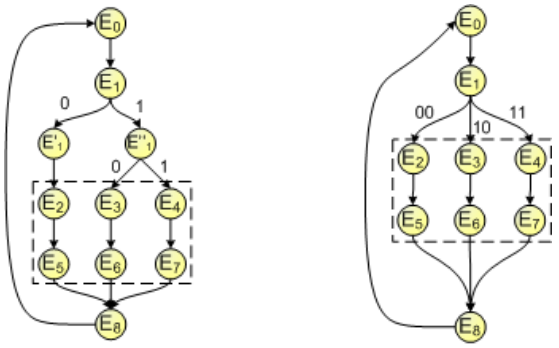

FIG. 4.12 – Implémentation d'une FSM de Moore.

sélection d'une branche conditionnelle d'une structure de type *case* à p branches correspond à n cycles avec $n = log_2(p) + 1$. Une implémentation de ce type est présentée dans la figure 4.12a.

Contrôleurs à n états fils (p retours en parallèle) - Dans le cas d'une implémentation d'un contrôleur de type *Moore* avec au maximum n fils, le besoin en terme de communication entre le chemin de données et le contrôleur est égal à $p = log_2(n) + 1$ fils de 1 bit. La ressource à mettre en oeuvre afin de transmettre les résultats conditionnels est importante, mais la latence de sélection d'une branche conditionnelle d'une structure de type *case* à 2^p branches est égale à 1 cycle. Une implémentation de ce type est présentée dans la figure 4.12b.

Choix réalisé - Les deux variantes possibles du contrôleur de *Moore* ont chacune leurs avantages et inconvénients en terme de compromis surface/latence. Notre flot de synthèse est dirigé par une contrainte en cadence. Nous avons précédemment expliqué que lorsque la latence devient trop importante, l'architecture devient pipeline ce qui entraîne un surcoût matériel (et donc un accroissement de la surface de l'architecture). Pour cette raison principale, nous avons choisi une implémentation de contrôleur pouvant posséder n retours d'états simultanés. L'ensemble des unités présentes dans l'architecture générique cible utilise donc des contrôleurs basés sur des FSM de Moore et possédant n entrées, ce qui permet de réduire la latence induite par les retours d'états.

Unité de traitement

Nous avons vu que des canaux de communications permettant de réaliser des transferts de données entre le chemin de données et le contrôleur ont été mis en place (schématisation figure 4.13). Pour réaliser le transfert des résultats conditionnels (retour des résultats des conditions vers les contrôleurs), nous utilisons le système des bus permettant aux unités de communiquer entre elles et cela pour 2 raisons : cela permet de réutiliser et de partager des bus dédiés au transfert des données, et cela permet également de réaliser une diffusion par *broadcasting* des informations lorsque ces dernières intéressent plusieurs unités (UMem/UCom).

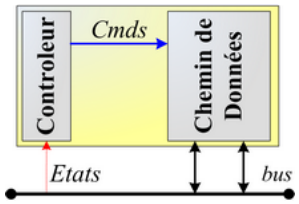

FIG. 4.13 – Unité de Traitement avec retour d'états.

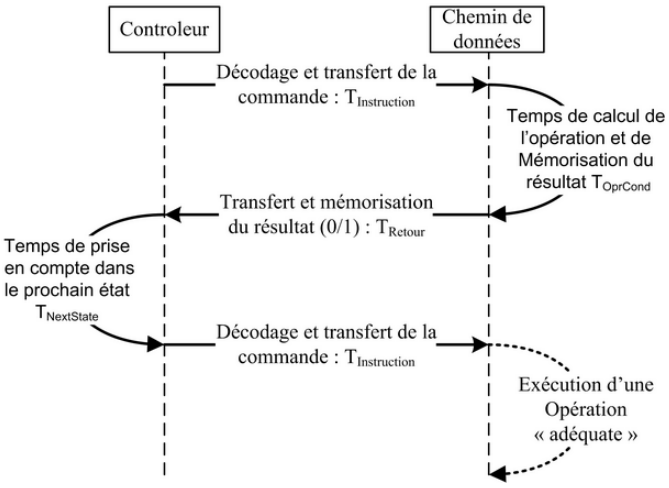

FIG. 4.14 – Diagramme de séquences modélisant un retour d'état.

L'architecture de l'unité de traitement est synchrone et pipeline. Elle nécessite plusieurs cycles notés $T_{Commandes}$ avant de transmettre une commande adaptée du contrôleur vers le chemin de données. De la même manière, un certain nombre de cycles noté T_{Retour} s'écoule entre la fin du calcul d'une condition et l'arrivée du résultat de cette condition au sein du contrôleur. Ces délais sont importants à connaître car ils permettent de savoir combien de cycles sont nécessaires entre le calcul d'une condition, son retour vers le contrôleur, le choix des opérations à exécuter et leur exécution dans le chemin de données. La figure 4.14 montre les étapes successives nécessaires entre le calcul d'une condition et le temps minimum nécessaire pour une répercussion au sein de l'unité de traitement.

Dans cet exemple nous avons considéré que le temps de traversée du contrôleur est d'un cycle d'horloge. Dans le cas de circuits complexes, il peut être nécessaire de pipeliner le contrôleur. Le temps de traversée du contrôleur devient donc fonction du nombre de tranches de pipeline qui le compose. Dans ce cas, la formule générale donnant le temps de réponse du contrôleur en fonction de la date d'exécution de la condition dans le chemin de données est :

$$T_{Controleur} = T_{NextState} + T_{Instruction} \tag{4.3}$$

Avec le temps de propagation dans le chemin de données égal à :

$$T_{Datapath} = T_{OprCond} + T_{Retour} \tag{4.4}$$

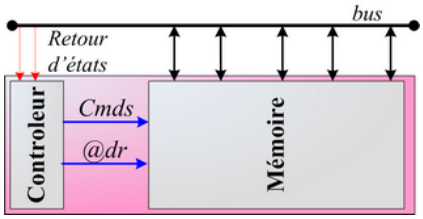

FIG. 4.15 – Modèle architectural de l'unité de mémorisation.

Le temps total de propagation d'une condition est exprimé comme étant :

$$T_{Total} = T_{Controleur} + T_{Datapath} \tag{4.5}$$

$$T_{Total} = (T_{NextState} + T_{Instruction}) + (T_{OprCond} + T_{Retour}) \tag{4.6}$$

Ces informations temporelles obtenues après analyse de la cible architecturale sont utiles durant la phase de projection architecturale. Elles sont utilisées lors de l'ordonnancement car elles contraignent le processus : une opération conditionnées ne peut être exécutée au plus tôt que T_{Total} cycles après que l'opération calculant la condition ait été ordonnancée dans le chemin de données. Ces temps de propagation intra-unité devront être extraits des architectures génériques ciblées afin de pouvoir justement contraindre la synthèse.

Unité de mémorisation

Pour implémenter les nouvelles primitives fonctionnelles, une nouvelle architecture de séquenceur d'accès à la mémoire est nécessaire afin d'accéder dynamiquement aux données (calculs d'adresses dynamiques/accès conditionnels). L'architecture présentée en figure 4.15 représente le modèle architectural de l'unité de mémorisation permettant l'accès à la mémoire (banc mémoire, séquenceur).

Le séquenceur détaillé en figure 4.16 contient deux nouveaux blocs fonctionnels destinés à fournir les services nécessaires à la gestion des accès dynamiques (mémorisation des adresses auxquelles on souhaite accéder et un contrôleur permettant de gérer les accès indéterministes). Cette nouvelle architecture de séquenceur mémoire permet par l'intermédiaire du chemin de données de l'UT d'accéder dynamiquement à des données : les adresses des données (ou leurs conditions d'accès) sont calculées au sein de l'unité de traitement et sont ensuite transférées au séquenceur par l'intermédiaires des bus de données. Une fois les données disponibles pour l'unité de mémorisation, le séquenceur prend les dispositions nécessaires afin de réaliser (ou geler) les accès en lecture/écriture (conditionnels ou dynamiques).

Implémentation des adressages (@dr logique => physique) - Dans les travaux de G. Corre [Corr05], l'intérêt du placement des différentes données d'un même vecteur dans une configuration multi-bancs est présenté. Ces configurations permettent d'exploiter lors de la synthèse le parallélisme d'accès aux mémoires afin d'améliorer les performances du circuit. Les techniques employées contraignent le processus

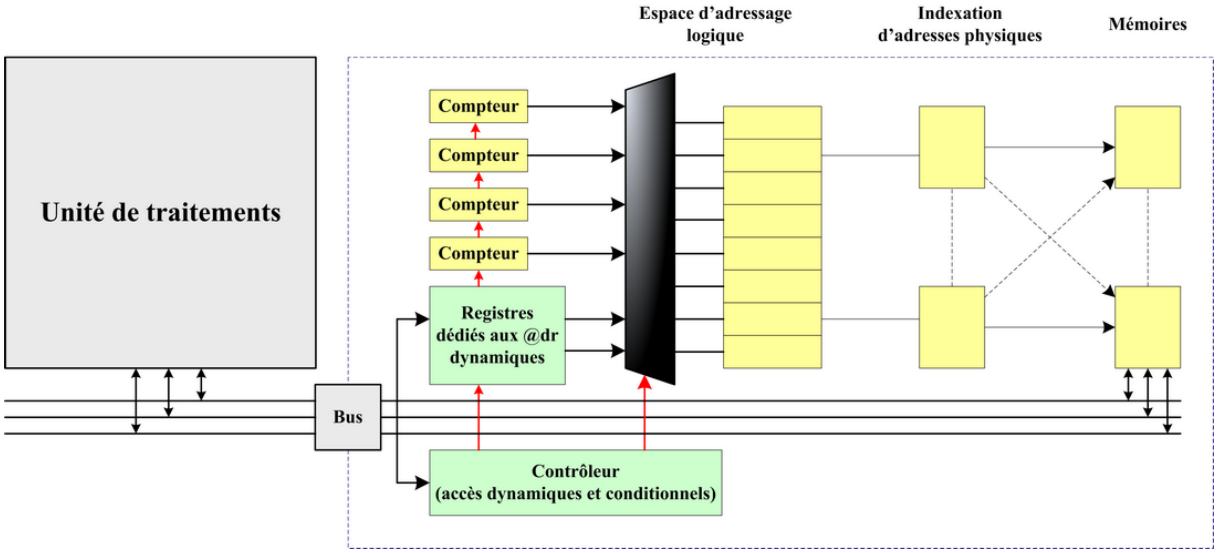

FIG. 4.16 – Nouvelle architecture du séquenceur mémoire pour les adressages indéterministes.

de synthèse par le mapping des données en mémoire, ainsi que par l'utilisation d'une table d'adressage dans l'architecture d'implémentation des séquenceurs. La table d'adressage permet d'associer à chaque adresse logique (donnée, tranche) un couple (adresse physique, banc mémoire). Dans notre méthodologie, nous pourrions employer cette technique afin de translater l'adresse dynamique calculée sous la forme d'un couple (adresse physique, banc mémoire). Deux possibilités s'offrent à nous :

1. Utilisation d'une table d'adressage prenant en entrée l'indice de l'élément à accéder et fournissant le couple (banc mémoire, adresse physique) de la donnée considérée,
2. Utilisation d'une table d'adressage prenant en entrée l'adresse logique de l'élément à accéder et fournissant le couple (banc mémoire, adresse physique) de la donnée considérée.

Suivant la méthode employée, les informations à transmettre à la table de translation sont différentes. La table de translation a ainsi suivant sa nature un impact direct sur les transformations à appliquer avant synthèse : doit-on calculer un indice de l'élément dans la structure ou bien son adresse logique au sein de la mémoire ?

Utilisation d'une table de translation d'indices - Dans le cadre des translations d'indices, on doit associer à chaque indice d'un vecteur un couple (banc mémoire, adresse physique). Afin de gérer une application réelle composée de plusieurs éléments structurés, il est nécessaire de posséder une table de translation d'indices pour chaque structure et de router les indices vers la bonne table (pointant vers la bonne structure) tout au long de l'exécution de l'application. Une telle architecture est présentée dans la figure 4.17a

Utilisation d'une table de translation d'adresses - Dans le cadre des translations d'indices, on associe à chaque donnée une adresse logique (banc mémoire, adresse physique). Afin de gérer une application réelle composée de plusieurs éléments structurés, chaque donnée, vecteur ou tableau possède une

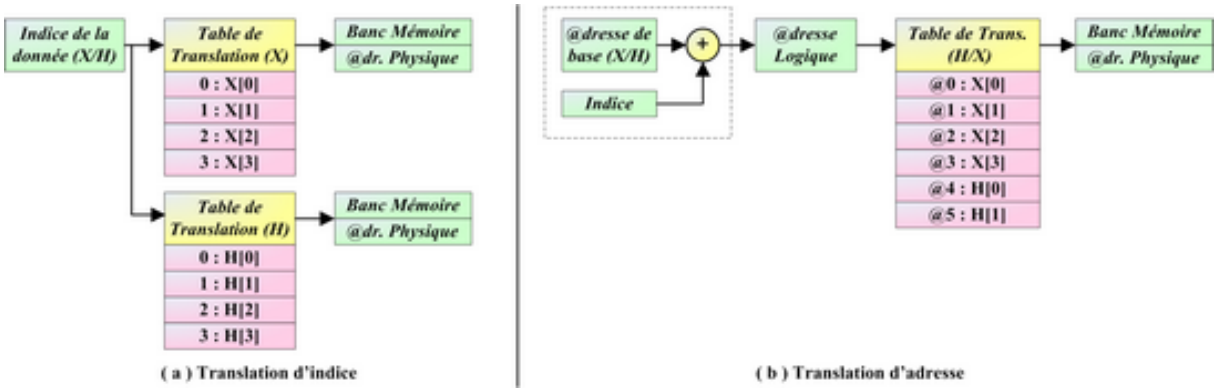

FIG. 4.17 – Exemple de tables de translation.

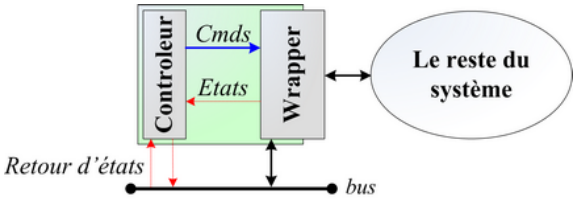

FIG. 4.18 – Modèle architectural de l'unité de communication.

adresse de base à partir de laquelle on vient calculer les adresses logiques des données *(adresse logique = adresse de base de la structure + indice)*. Grâce à cela, une seule table de translation d'adresses est nécessaire pour l'ensemble de l'application. La contrepartie de cette approche réside dans le calcul obligatoire de l'adresse logique à partir de l'adresse de base. Cette technique est, par exemple, employée dans les compilateurs logiciels. La figure 4.17b montre le principe de cette méthode.

Bilan - Afin de simplifier l'architecture cible, nous incluons au sein du séquenceur mémoire une table de translation basée sur l'adresse des données auxquelles on souhaite accéder. Dans le graphe modélisant l'algorithme à implémenter il est nécessaire de transformer les calculs d'indices en calculs d'adresses *(adresse logique = adresse de base + indice)*. La prise en considération de la translation des adressages logiques vers des adressages physiques au sein de l'unité de mémorisation autorise ainsi une répartition de manière non contiguë des données au sein de la mémoire afin d'augmenter le parallélisme d'accès aux structures.

Unité de communication

L'unité de communication est également modifiée afin de supporter la gestion des structures conditionnelles. Cette gestion des retours d'états est assurée par une connexion entre les bus de données et le contrôleur de l'unité de communication (figure 4.18).

Cette transformation permettra de geler les transferts inutiles. Cette technique peut être étendue pour éviter la mémorisation, à l'intérieur de l'UCom, d'E/S devenues inutiles lors de l'exécution. Des travaux

relatifs à l'UCom sont actuellement en cours au LESTER dans le cadre de la thèse de C. Chavet.

4.3.6 Implémentation architecturale des primitives fonctionnelles

Nous venons de présenter les modèles fonctionnels et structurels que nous ciblons. Nous allons maintenant considérer les règles d'implémentation des primitives algorithmiques sur l'architecture ainsi que les délais induits par ces projections.

Implémentation des calculs d'adresses dynamiques

Les adressages dynamiques (calculs d'adresses dynamiques ou accès conditionnels) ne peuvent être résolus avant/pendant la synthèse contrairement aux adressages statiques. Ils sont évalués durant l'exécution de l'application. Une fois ces calculs d'adresses réalisés, il revient à l'unité de mémorisation de délivrer les données adressées. Les adresses sont calculées au sein de l'UT. Elles sont alors transmises en direction du séquenceur mémoire par l'intermédiaire des bus de données. Les délais temporels induits par un accès indéterministe (adressage dynamique uniquement) sont présentés dans la figure 4.19. Lire une donnée à partir d'une adresse dynamiquement calculée nécessite par exemple un certain nombre d'opérations :

1. L'adresse doit être transférée de l'UT vers l'UMem, ce temps sera noté $T_{Transfert}$.
2. Le séquenceur mémoire doit mémoriser l'adresse afin de réaliser un accès synchrone à la mémoire. Le temps nécessaire à cette mémorisation est noté $T_{Memorisation}$ (NB : dans le cas d'une écriture la donnée peut arriver dans un cycle différent de celui où arrive l'adresse).
3. Le séquenceur doit translater l'adresse logique sous la forme d'un couple (banc mémoire, adresse physique) et piloter le banc pour effectuer la lecture ou l'écriture : en fonction de l'opération, le temps nécessaire sera noté respectivement $T_{Lecture}$ ou $T_{Ecriture}$.
4. Dans le cas où l'opération s'avérerait être une lecture, la donnée a besoin d'être transférée vers l'unité de traitement via les bus de données (nécessitant un délai de $T_{Transfert}$).

Les délais nécessaires à la complétion des lectures et écritures dynamiques au sein de l'unité de mémorisation sont exprimés dans les équations suivantes :

$$T_{Lecture-dynamique} = T_{Transfert-adresse} + T_{memorisation} + T_{Lecture} + T_{Transfert-donnee} \tag{4.7}$$

$$T_{Ecriture-dynamique} = Max(T_{Transfert-adresse}, T_{Transfert-donnee}) + T_{Memorisation} + T_{Ecriture} \tag{4.8}$$

Les délais exprimés dans les équations (4.8, 4.8) sont fonction de l'architecture d'implémentation et des différents composants qui ont été alloués.

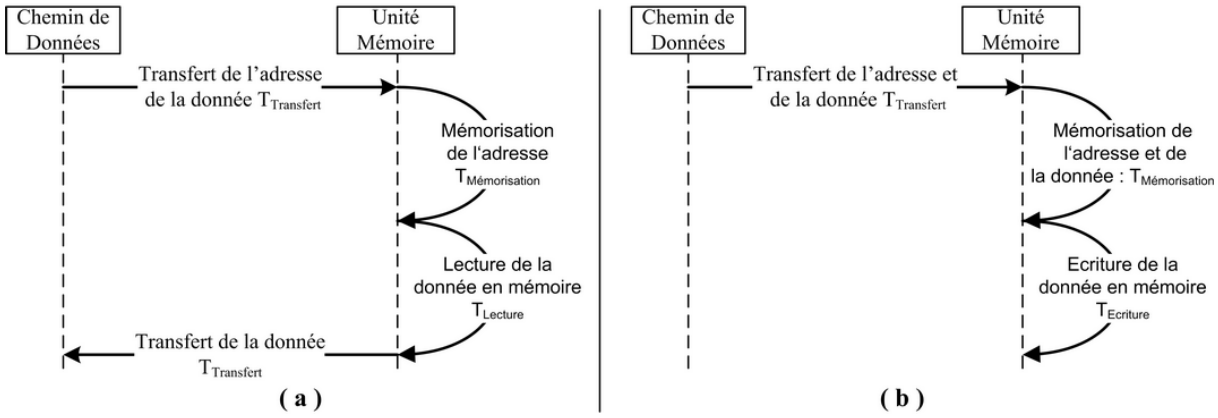

FIG. 4.19 – Diagramme de séquences pour une lecture/écriture dynamique.

Implémentation des structures conditionnelles

Le nombre de cycles de pénalité varie en fonction du type de la structure conditionnelle (locale, globale). Afin de réaliser une étude plus fine des différentes pénalités que nous serons amenés à prendre en compte, nous allons étudier les deux cas envisageables :

– *Branche conditionnelle locale :* dans ce cas la portée du retour d'états se limite à l'unité de traitement. La condition calculée dans l'unité de traitement est renvoyée à l'aide de bus vers le contrôleur de cette même unité.
– *Branche conditionnelle globale :* dans ce cas de figure, les autres unités de l'architecture nécessitent le retour d'états (lecture en mémoire conditionnée par exemple). La condition calculée dans le chemin de données de l'UT doit être transmise vers le contrôleur de l'UT, mais aussi vers celui/ceux de l'UMem et/ou de l'UCom. Une fois le retour d'états reçu, l'UMem et/ou l'UCom prendront les décisions appropriées afin de transmettre les données adéquates. Dans ce cas de figure, les pénalités supportées sont plus importantes comme le montre la figure 4.20 (lecture de donnée conditionnelle).

La différence entre la pénalité imposée par un retour d'états local et un retour d'états global peut être importante. Ces pénalités dépendent essentiellement de l'architecture d'implémentation ciblée (vitesse des différentes unités, du réseau de communication, etc.). Ces informations devront toutefois être prises en considération lors de la synthèse (ordonnancement) de l'application. Cette prise en considération se fait par une annotation du CSFG.

4.4 Projection architecturale

Nous venons de présenter l'architecture générique que cible notre flot de synthèse. Nous avons extrait de ce modèle les délais associés à l'implémentation de diverses primitives fonctionnelles. Ces informations vont être mises à profit pour annoter le modèle de spécification afin de guider la synthèse (figure 4.21). Le point de départ est le CSFG qui modélise l'application. La première étape nommée *projection architecturale* va utiliser les informations intrinsèques de l'application afin d'ajouter une localité à chacun des

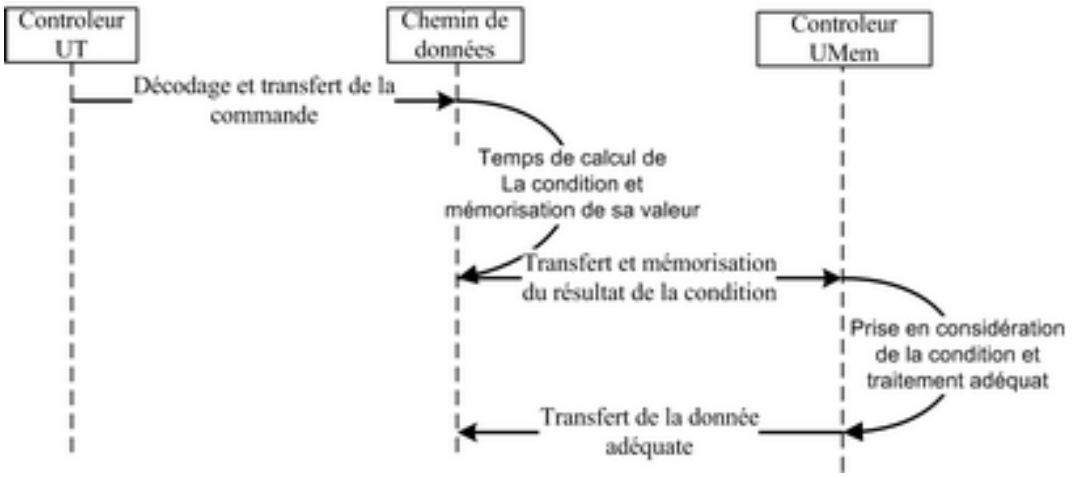

FIG. 4.20 – Exemple de branche conditionnelle globale avec implication de l'UMem pour une lecture conditionnelle.

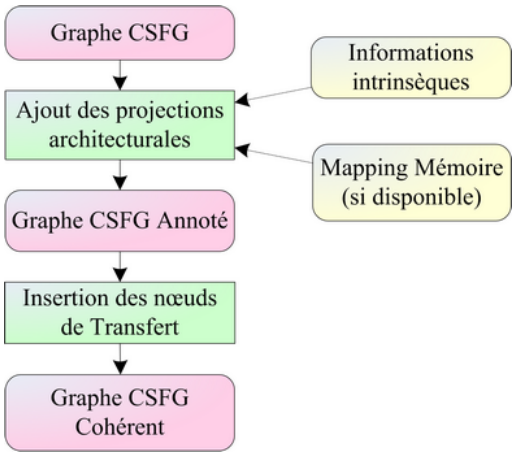

FIG. 4.21 – Flot de projection architecturale.

noeuds ainsi que les pénalités dues aux retours d'états sur les arcs. Cette étape d'annotation peut être guidée si le concepteur fournit un *mapping* mémoire, mais cela n'est pas obligatoire. Dans un second temps une modélisation explicite des transferts entre les différentes unités va être insérée dans le modèle de représentation. Ces 2 étapes sont détaillées dans les prochaines parties.

4.4.1 Annotation du graphe par les localités

L'architecture ciblée post synthèse est composée comme nous l'avons déjà précisé de 3 unités distinctes : l'unité de traitement (UT), unité de mémorisation (UMem) et l'unité de communication (UCom). L'ensemble des opérations contenues dans la représentation fonctionnelle ainsi que les données sont donc implémentées/mémorisées dans l'une de ces unités après synthèse.

Définition 4.4.1 (*Localisation des noeuds du graphe*)

L'opération de localisation des opérations et des variables consiste à ajouter une annotation aux

```
Algorithme d'annotation des variables

Graphe AnnotationDesVariables( Graphe G, Mapping m )
ListeVar = RécupererToutesLesVariablesDuGraphe( G );
Pour tous les variables contenues dans ListeVar
    Selon Que
        Type(variable) = E/S :
                loc(variable) ← UCom;
                break;
        Type(variable) = Structure :
                loc(variable) ← UMem;
                break;
        Type(variable) = Données vieillissante:
                loc(variable) ← UMem;
                break;
        Type(variable) = Constante :
                loc(variable) ← UMem;
                break;
        Défaut :
                Si ( m = null ) Alors
                        loc(variable) ← UT;
                Sinon
                        loc(variable) ← RechercherMapping( variable, m);
                Fin Si
    Fin Selon Que
Fin pour
Retourner G;
```

FIG. 4.22 – Algorithme d'annotation des variables par leur localité.

noeuds spécifiant où est réalisé/mémorisé le noeud du graphe. Afin de réaliser cette annotation, nous définissons la fonction Loc : $Localite \leftarrow noeud$ la fonction qui permet de lire ou de modifier la localité du noeud considéré avec $Localite \in \{UT, UMem, UCom\}$.

Les variables contenues dans graphe vont tout d'abord être annotées à l'aide d'informations intrinsèques telles que :

- Si la variable est une entrée/sortie, alors elle est localisée dans l'UCom,
- Si la variable est une structure, une constante ou un scalaire vieillissant, alors elle est localisée dans l'UMem,
- Le reste des variables du graphe sont des variables temporaires utilisées dans les calculs ; ces dernières sont typiquement implémentées dans l'UT.

Cette annotation peut être complétée par le mapping mémoire réalisé par le concepteur du circuit.

L'annotation du graphe à l'aide des informations intrinsèques est réalisée suivant la méthode présentée dans l'algorithme 4.22. L'exemple proposé dans la figure 4.23 illustre l'étape d'annotation des variables comprises dans le graphe en fonction des informations intrinsèques associées.

Une fois les variables du graphe annotées, il faut accomplir de même pour les opérations. La localité des opérations est telle que :

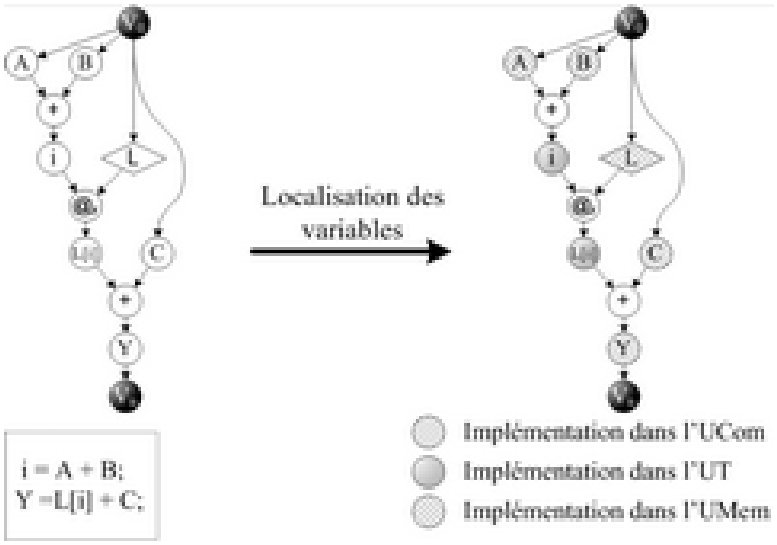

FIG. 4.23 – Exemple d'annotation des variables par leur localité.

– l'ensemble des opérations arithmétiques et logiques est réalisé dans l'unité de traitement,
– seules les lectures/écritures mémoire (les noeuds $@r$ et $@w$) sont implémentées dans l'unité de mémorisation,
– aucune opération n'est implantée au sein de l'unité de mémorisation.

L'algorithme permettant de réaliser cette tache est présenté dans la figure 4.24.

Si nous reprenons notre exemple et que nous lui appliquons cette seconde étape d'annotation, nous obtenons un graphe entièrement annoté par la projection architecturale de la localité (figure 4.25).

Le graphe ainsi obtenu renseigne sur la localisation a priori des données qui vont être utilisées pour les calculs réalisés dans l'UT et met donc en évidence des transferts à opérer. Pour l'instant, l'ensemble des transferts de données (données, adresses, retours d'états) est intrinsèque aux changements de localité : par exemple les entrées A et B avant de pouvoir être utilisées par l'opération $(+)$ devront être transférées de l'unité de communication vers l'unité de traitement.

4.4.2 Cohérence du graphe

Afin d'assurer une cohérence entre la mémorisation des données et leur(s) consommation(s), nous définissons la notion de cohérence d'un graphe. Cette notion de cohérence permet de vérifier la correspondance des localités des données et des opérations liées. La cohérence du graphe permet de vérifier que toutes les données sont transférées, si nécessaire de manière explicite, à l'aide de noeuds de transfert entre les différentes unités où elles seront produites, mémorisées et consommées.

Définition 4.4.2 (*Cohérence d'une séquence de noeuds*)

Nous définissons la cohérence d'une séquence de noeuds S comme étant une séquence pour laquelle la relation suivante est vérifiée : $\forall v \in S \rightarrow loc(v) = loc(Pred(v)) = loc(Succ(v))$. *Dans le cas où la relation n'est pas vérifiée, alors la séquence de noeuds S n'est pas cohérente. Un graphe sera dit*

Algorithme d'annotation des opérateurs

```
Graphe AnnotationDesOpérateurs( Graphe G )
ListeOpr = RécupererToutesLesOpérationsDuGraphe( G );
Pour tous les opérations contenues dans ListeOpr
    Selon Que
        Type(opération) = @r :
                loc(opération) ← UMem;
                break;
        Type(opération) = @w :
                loc(opération) ← UMem;
                break;
        Défaut :
                loc(opération) ← UT;
                break;
    Fin Selon Que
Fin pour
Retourner G;
```

FIG. 4.24 – Algorithme d'annotation des opérations par leur localité.

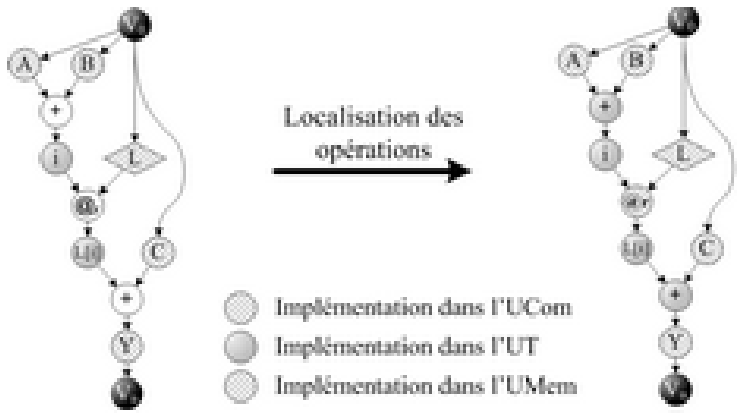

FIG. 4.25 – Exemple d'annotation des opérations par leur localité.

cohérent si l'ensemble des séquences de noeuds compris dans ce graphe est cohérent.

4.4.3 Ajout de noeuds de transfert dans le graphe annoté

Nous venons de montrer que l'on peut exprimer de manière intrinsèque les transferts de données entre unités différentes. A l'aide de noeuds spécifiques de type opération, nous allons maintenant symboliser explicitement ces transferts afin de rendre le graphe cohérent.

Définition 4.4.3 (*Les noeuds de transfert*)

Les noeuds de transfert sont des noeuds de type opération qui permettent la modélisation d'un transfert de données inter-unités. Ces transferts de données inter-unités sont réalisés à l'aide des bus interconnectant ces dernières. Une opération de type transfert possède une double localité, celle du noeud source d'où part le transfert et celle du noeud destination où arrive la donnée. Le temps d'exécution d'un noeud de transfert v_i noté $\lambda(v_i) = T_{Transfert}$ équivaut au temps nécessaire pour réaliser le transfert de la donnée depuis la source vers la destination.

Ce passage de la déclaration implicite des transferts à une déclaration explicite permet une modélisation précise des temps de transfert entre les différentes unités de l'architecture. Cela peut être utile lorsque, par exemple, on utilise plusieurs bancs mémoire possédant des caractéristiques différentes (certaines plus lentes pour limiter la consommation du circuit) ou bien lorsque le nombre de ressources de communication est limité. La technique d'insertion des noeuds de transfert au sein du graphe afin de le rendre cohérent est décrite par l'algorithme (figure 4.26).

Si nous considérons l'exemple pris précédemment afin d'illustrer les annotations de localité, nous pouvons rendre ce graphe cohérent en ajoutant des noeuds de transferts. Une fois la transformation appliquée, nous obtenons la figure 4.27.

Il ressort de cette figure une "clusteurisation" des opérations et des mémorisations effectuées dans chacune des unités. Ces clusters sont toujours séparés à l'aide de noeuds de transfert (un cluster est une séquence d'opérations comprise entre deux noeuds de transfert). Cette figure permet aussi d'observer le nombre de transferts inter-unités nécessaires afin d'implémenter l'application. Dans l'exemple présenté ce nombre est égal à 6.

Perspective

La modélisation explicite des transferts de données via des noeuds opérations peut permettre de contraindre la synthèse sous contrainte de canaux de communication (sélection et allocation d'un nombre limité de bus entre les différentes unités). La limitation du nombre de connexions entre les unités pourrait faire partie d'une optimisation basée sur le principe d'une allocation moyenne (comme pour les opérateurs). Cette perspective ne sera pas développée dans ce mémoire car elle pourrait impacter négativement sur l'implémentation des structures conditionnelles et des accès dynamiques à la mémoire. Dans le cas d'applications déterministes, cette solution serait par contre certainement bénéfique.

Maintenant que nous avons annoté le modèle de représentation, nous allons détailler l'ensemble des méthodes qui va être mis en oeuvre afin de projeter le modèle de représentation de l'application sur

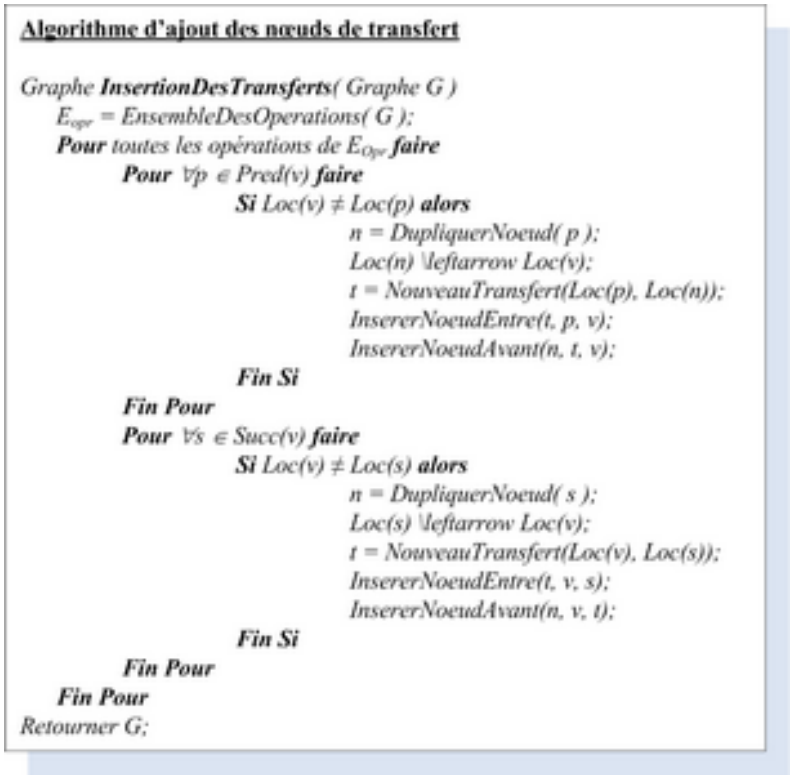

FIG. 4.26 – Insertion des noeuds de transfert.

l'architecture cible.

4.4.4 Sélection et allocation des opérateurs

La méthode de sélection des opérateurs nécessaires à l'implémentation matérielle des opérations reste équivalente à celle utilisée dans l'outil GAUT. La méthode d'allocation des opérateurs, pour permettre l'exploitation du parallélisme contenu dans le modèle de représentation, est aussi identique car il n'y a a priori aucune connaissance de l'efficacité du partage des opérateurs entre les opérations mutuellement exclusives. Une nouvelle méthode d'allocation des opérateurs pourrait néanmoins être développée dans de futurs travaux afin de prendre en considération le partage possible dans le cas des branches conditionnelles et des boucles non déroulées.

4.4.5 Ordonnancement et Assignation

L'objectif de notre technique d'ordonnancement/assignation est de minimiser le temps nécessaire à l'ordonnancement des opérations afin d'obtenir rapidement une solution. La solution consiste à utiliser un algorithme d'ordonnancement de faible complexité permettant de prendre en compte le partage des opérations mutuellement exclusives ainsi que les adressages dynamiques. Nous allons réemployer un algorithme d'ordonnancement de type *Static List-Scheduling* comme employé initialement dans l'outil GAUT. Durant la phase d'ordonnancement, certaines opérations supplémentaires vont être néanmoins

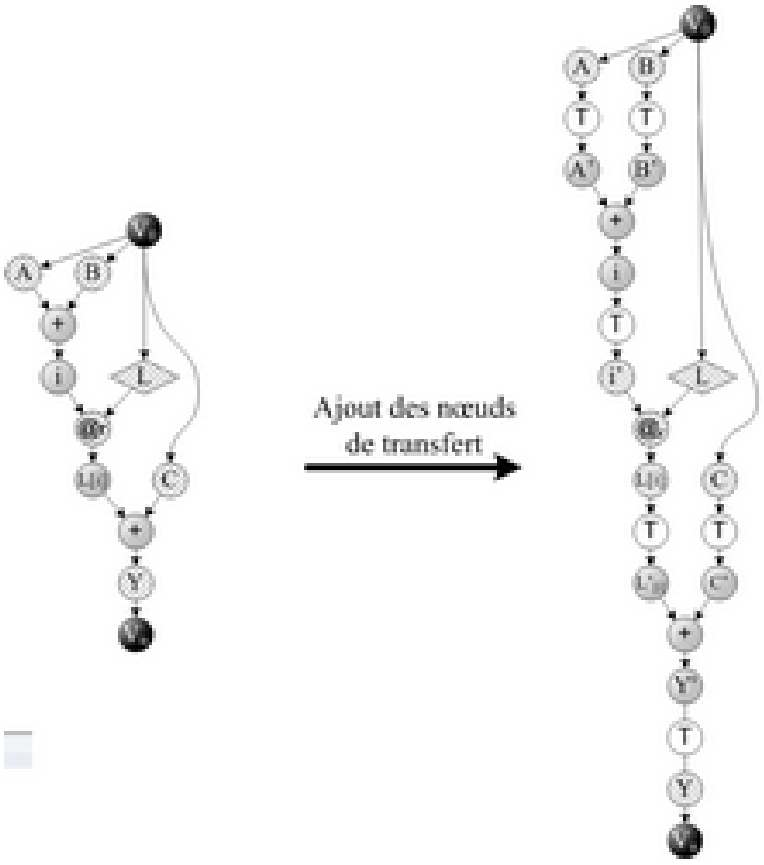

FIG. 4.27 – Exemple d'insertion des noeuds de transfert.

réalisées :

- *Gestion des opérations conditionnées :* lors de l'étape d'ordonnancement/assignation d'une opération conditionnée, il est important de mémoriser la condition qui permettra ou non l'exécution de l'opération afin de générer le contrôleur adéquat post-synthèse.
- *Gestion des opérations mutuellement exclusives :* lors de l'assignation d'une opération sur un opérateur, il est nécessaire de chercher dans la liste des opérations exécutables si une opération est mutuellement exclusive avec celle traitée afin de partager l'opérateur durant le T_{cycle}.
- *Gestion des transferts de données :* la gestion des transferts de données intra et inter-unités étant maintenant explicite, il faut assigner à ces transferts des moyens physiques. Pour cela, une file de bus inter-unités sera mise en place.
- *Adressages dynamiques :* lors de l'ordonnancement des accès dynamiques à la mémoire, il est nécessaire de verrouiller l'accès à l'ensemble des bancs mémoire qui contiennent au moins un élément de la structure adressée.
- *Les noeuds hiérarchiques :* la gestion des noeuds hiérarchiques doit être réalisée de manière équivalente à l'ensemble des autres noeuds. Ces derniers sont ordonnancés une fois lors de la phase de préordonnancement du graphe. De plus amples informations seront trouvées dans l'exemple pédagogique consacré aux noeuds hiérarchiques (section 5.2.1).

Des techniques d'ordonnancement ont été développées pour répondre à chacun des points qui viennent d'être énoncé. Nous présentons ces méthodes dans la partie dédiée aux exemples pédagogiques du chapitre "Expérimentations". L'ensemble de ces algorithmes d'ordonnancement et d'assignation sera fusionné dans les travaux de C. Andriami qui réalise actuellement une thèse au laboratoire LESTER.

4.4.6 Conclusion

Nous avons présenté un nouveau flot de synthèse du graphe qui prend en considération les structures conditionnelles avec partage des opérateurs ainsi que l'accès dynamique aux données. Notre méthode permet également l'implémentation de noeuds hiérarchiques qui ont pour but de réduire la complexité de l'ordonnancement. La technique d'ordonnancement/assignation de tels noeuds est adaptée afin de permettre une exploitation du parallélisme de l'application

4.5 Optimisation des calculs d'adresses dynamiques

4.5.1 Motivations

Dans les précédantes parties de ce manuscrit, nous avons pu observer que les adressages dynamiques impliquent le transfert des adresses calculées dans l'UT vers le séquenceur de l'UMem. Ces transferts d'adresses induisent certains effets néfastes sur les performances du circuit : la nécessité de réaliser un transfert entraîne la commutation des bus et engendre donc une consommation d'énergie. De plus, il est nécessaire de posséder un nombre de bus et de registres conséquent (pour mémoriser les adresses dans l'UT et dans le séquenceur) afin de permettre le transfert des données et des adresses en parallèle quand

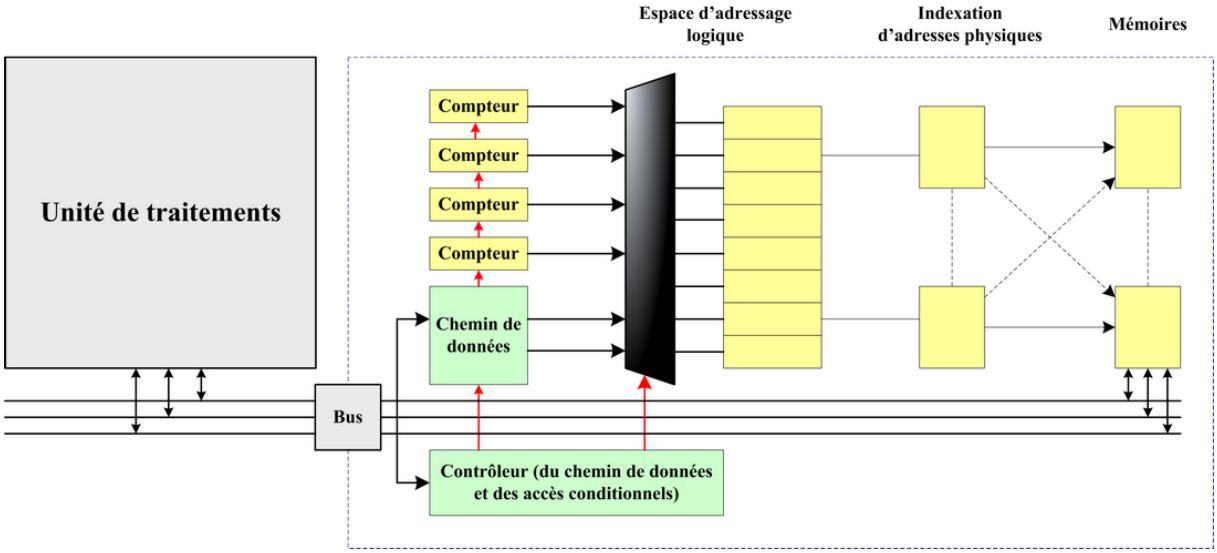

FIG. 4.28 – Séquenceur d'accès mémoire avec chemin de données pour les calculs d'adresses dynamiques.

cela est nécessaire. Ces contraintes provoquent une augmentation de la surface et de la consommation de l'architecture finale. Un exemple expliquant nos motivations dans le transfert des calculs d'adresses de l'UT vers le séquenceur mémoire est détaillé dans la section 5.2.3. Dans cet exemple nous montrons qu'il est intéressant, dans certains cas, de réaffecter les calculs d'adresses dans le séquenceur mémoire afin de réduire le nombre de transferts inter-unités à réaliser. Nous allons aborder maintenant les évolutions à apporter à l'architecture cible ainsi qu'au flot de synthèse pour bénéficier des gains apportés par la relocalisation des calculs d'adresses (de l'UT vers l'UMem).

4.5.2 Modification de l'architecture cible

L'exemple présenté dans les "exemples pédagogiques" nous permet d'apprehender au sein de l'unité de mémorisation un chemin de données dédié aux calculs d'adresses dynamiques. Ce chemin de données spécifique doit se situer en amont de la table de translation d'adresses afin de reproduire le comportement initial (lorsque les calculs d'adresses étaient réalisés dans l'UT). L'architecture obtenue est présentée dans la figure 4.28.

Le chemin de données implanté dans le séquenceur est construit de manière identique à celui de l'unité de traitement. Il se compose d'opérateurs arithmétiques et logiques effectuant les opérations nécessaires aux calculs d'adresses (figure 4.29).

L'ensemble des éléments présents (opérateurs et registres) dans le chemin de données interne au séquenceur est dimensionné de manière adéquate par rapport à la dynamique des adresses à calculer. En effet, dans les applications TDSI, la largeur en nombre de bits des adresses est souvent inférieure à la largeur des données présentes en mémoire (par exemple : le bus d'adresse est sur 16 bits alors que les données mémorisées sont codées sur 32 bits). Cette réduction du nombre de bits du chemin de données permet de

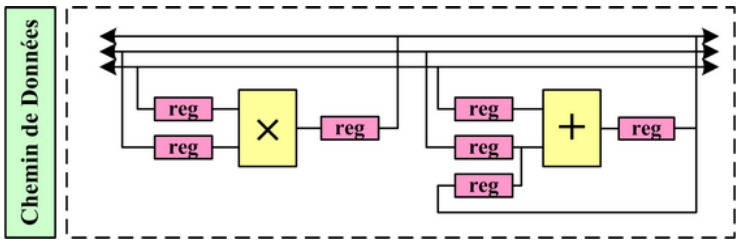

FIG. 4.29 – Chemin de données interne au séquenceur.

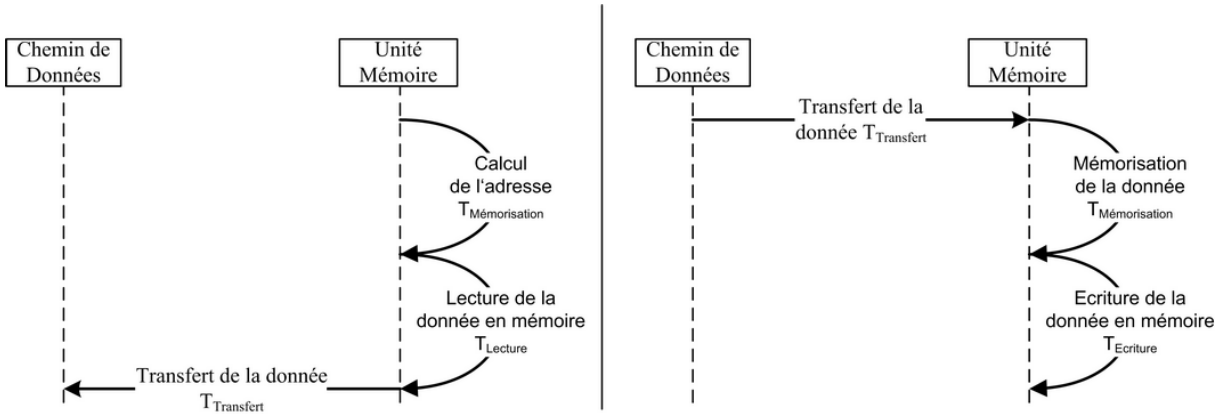

FIG. 4.30 – Diagramme de séquences pour les accès dynamiques à la mémoire.

réduire la surface nécessaire et le nombre de commutations relatives aux calculs d'adresses dynamiques (par rapport à une solution où ces calculs sont opérés dans l'unité de traitement).

Remarque

> Il est nécessaire de tempérer la précédente remarque car si le nombre de calculs d'adresses dynamiques est excessivement faible, ces derniers auraient probablement avantage à être implémentés dans l'unité de traitement sur un opérateur "dédié" aux calculs sur les données. Dans ce cas précis, le gain en surface produit par notre méthode peut devenir négatif.

Le diagramme de séquences présenté dans la figure 4.30 détaille les différentes étapes nécessaires pour réaliser une lecture/écriture mémoire lorsque les calculs d'adresses dynamiques sont implémentés directement dans le séquenceur. Pour cette figure, nous partons du principe que les données nécessaires au calcul d'adresse sont déjà disponibles en mémoire. Par rapport aux diagrammes de séquences présentés dans la partie 4.3, nous pouvons remarquer que l'introduction du chemin de données dédié dans l'UMem permet de réduire le nombre de transferts nécessaire pour réaliser des accès dynamiques (fig. 4.30).

4.5.3 Optimisation du graphe

Nous allons intégrer dans notre flot de synthèse une étape d'optimisation afin de permettre le transfert de certains calculs d'adresses dynamiques présents dans l'UT vers l'UMem. Cette étape de transfert (des calculs d'adresses de l'UT vers le chemin de données de l'UMem) va être effectuée entre l'étape

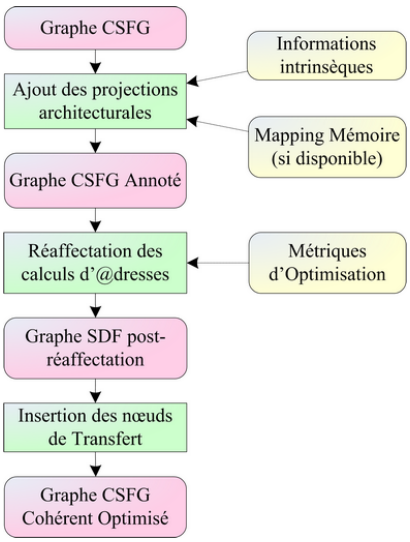

FIG. 4.31 – Flot d'optimisation des calculs d'adresses dynamiques présynthèse.

d'annotation du graphe et celle d'insertion des noeuds de transfert. Le flot d'optimisation présynthèse modifié est présenté dans la figure 4.31.

A partir des annotations du graphe, il est possible de déterminer les gains positifs/négatifs apportés par le changement du lieu d'implémentation de l'opération.

Métriques d'optimisation

Comme nous l'avons montré dans l'exemple introductif, les transferts de calculs d'adresses peuvent également aider à réduire la latence de l'application. Ces transformations impliquent la mise en place d'un chemin de données dans les séquenceurs de l'UMem donnant lieu à une augmentation de la surface de cette dernière (registres, opérateurs, contrôleur du chemin de données). Afin d'obtenir une architecture globale optimale, des métriques sont utilisées pour guider l'étape de transfert des calculs d'adresses. Le critère de décision est une somme pondérée des différentes métriques suivantes :

1. *Le nombre de transferts* nécessaire pour effectuer le calcul d'adresse dynamique au sein de chacune des unités (UT/UMem). Les transferts de données et d'adresses provoquent des commutations sur les bus. La réduction du trafic sur les bus peut aussi induire une réduction de leur nombre.
2. *L'augmentation ou la réduction de la latence* du graphe. Le transfert d'une opération d'une unité à l'autre peut permettre de réduire le nombre global de transferts nécessaire à l'application, mais cela peut aussi provoquer l'allongement du chemin auquel appartient l'opération. Si le chemin rallongé est un chemin critique, cette transformation augmente la latence maximum de l'application après synthèse.
3. *La dynamique des opérateurs et des registres* dédiés aux calculs d'adresses par rapport à celle des opérateurs utilisés dans l'UT. Dans les applications TDSI, la dynamique des données est souvent plus importante que la dynamique nécessaire pour les adresses. Il peut être judicieux d'allouer un

opérateur et ses registres avec une dynamique de 16 bits dans l'UMem plutôt que d'allouer les mêmes composants sur 32 bits dans l'UT.

4. *Le taux d'utilisation des opérateurs* au sein de chaque chemin de données (celui de l'unité de traitement et celui du séquenceur). Après avoir transféré les calculs d'adresses d'une unité vers l'autre, il est intéressant d'observer les taux d'utilisation des opérateurs a posteriori afin de vérifier si le chemin de données de l'UMem est rentabilisé (a posteriori).

L'ensemble des métriques élémentaires va nous permettre de composer une fonction de coût locale à chaque calcul d'adresse ainsi qu'une métrique globale permettant d'évaluer la pertinence de la solution du graphe après optimisation complète. La fonction de coût permettant de décider si un calcul d'adresse doit être transféré de l'UT vers l'UMem, est fonction des métriques suivantes : le nombre de transferts (1) et la réduction/augmentation de la latence de l'application (2). La fonction de coût globale est fonction du gain sur la dynamique des opérateurs (3) et de leur taux d'utilisation après optimisation (4) qui permet de calculer les gains obtenus après optimisation.

$$Metrique\ locale = p_0 \times (gain\ nbre\ transfert) + p_1 \times (reduction\ latence) \tag{4.9}$$

$$Metrique\ globale = p_2 \times (optimisation\ dynamique) + p_3 \times (augmentation\ rendement) \tag{4.10}$$

Les équations (4.9, 4.10) montrent que les métriques sont pondérées au sein des fonctions de coût. Le poids associé à chacune des métriques permet de décider de son impact dans la fonction de coût utilisée pour l'optimisation courante du graphe. Les changements de pondération des métriques permettent d'orienter les effets souhaités par le transfert des calculs d'adresses : on peut décider d'orienter l'optimisation afin de réduire la latence, la consommation ou réaliser un compromis entre les deux.

Algorithme d'optimisation

La méthode de réaffectation des calculs d'adresses entre le chemin de données et le séquenceur mémoire est appliquée de manière statique. Cette technique (algorithme 4.32) est employée avant synthèse.

La méthode d'optimisation agit de manière locale afin de limiter sa complexité calculatoire. Cette décision est prise à l'aide des fonctions de coût présentée précédemment : si la métrique indique un gain possible, alors la localité de l'opération et de son résultat est modifiée afin de représenter sa nouvelle affectation. Cette procédure est répétée à l'identique pour tous les calculs d'adresses présents dans le graphe.

4.5.4 Synthèse avec l'unité de calcul d'adresses

Le processus de synthèse de l'outil GAUT n'est pas adapté à la réalisation d'une synthèse commune de deux chemins de données indépendants (UT et UMem). Le flot doit donc être adapté afin de répondre aux modifications que nous avons apportées au modèle de spécification et au modèle architectural cible.

Algorithme de réaffectation des calculs d'adresses

```
Graphe OptimisationParRéaffectation( Graphe G )
ListeOpr = RécupererToutesLesOpérationsDuGraphe( G );
Pour toutes les opérations contenues dans ListeOpr
    Si Type(opération ) = Calcul d'adresse alors
        gainL = CalculGainLatence( opération )
        gainT = CalculGainTransfert( opération )
        Métrique = (PoidsL * gainL) + (PoidsT * gainT)
        Si Métrique ≥ Seuil de transfert Alors
            ChangerLocalitéOpération( opération );
            ChangerLocalitéOpérandes( opération );
        Fin Si
    Fin Si
Fin pour
Retourner G;
```

FIG. 4.32 – Algorithme de réaffectation des calculs d'adresses en fonction de la valeur du métrique.

Les algorithmes de sélection, allocation, ordonnancement et assignation doivent être modifiés en conséquence afin de garantir la génération d'une architecture correcte dans un processus de synthèse commun. Ces modifications, présentées dans les prochaines parties, visent à prendre en compte la localité des calculs qui peuvent être déportés au sein du séquenceur.

4.6 Synthèse de l'architecture

Le flot de synthèse architectural (fig. 4.33) consiste en étapes de sélection et d'allocation indépendantes pour chacun des différents chemins de données tandis que les étapes d'ordonnancement et d'assignation sont communes pour les deux chemins de données.

4.6.1 Sélection des opérateurs dans la nouvelle approche

L'étape d'assignation qui se déroule de manière conjointe à celle d'ordonnancement réalise l'assignation des opérations ordonnancées sur les opérateurs matériels disponibles. Cette étape est réalisée en adéquation avec la localité des opérations et des opérateurs disponibles. Les opérateurs sélectionnés ne sont plus destinés uniquement au chemin de données de l'unité de traitement, mais aussi à celui de l'unité de mémorisation. Afin de simplifier l'opération de sélection, nous allons considérer de manières indépendantes les 2 chemins de données. A partir de maintenant nous utiliserons donc 2 listes pour modéliser :

1. les opérations nécessaires à l'unité de traitement $(Liste_{fonctions-UT})$,
2. les opérations réalisées dans le séquenceur $(Liste_{fonctions_UMem})$,

Chacune de ces listes contient l'ensemble des opérations différentes (ainsi que leur nombre) du graphe et qui doivent être implémentées.

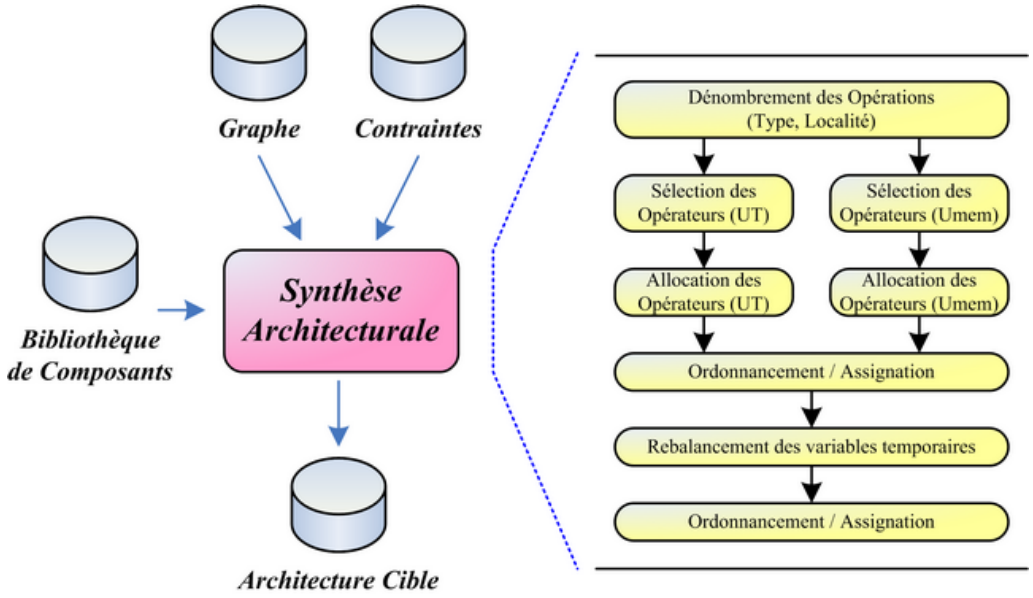

FIG. 4.33 – Nouveau flot de synthèse d'architecture.

Les opérations d'accès à la mémoire et les transferts de données - Les noeuds opérations de type transfert (T) qui ont été ajoutés au graphe afin de modéliser les contraintes architecturales (noeuds de type adressage dynamique, transfert de données inter-unités) ne donnent pas lieu à la sélection d'opérateurs. En effet, ces opérations sont réalisées implicitement par l'architecture cible (bus inter-unités et utilisation des décodeurs d'adresses des bancs mémoire). En fonction de l'architecture cible (technologie cible, paramètres des bancs mémoire, ...), les informations caractérisant ces opérations (latence, surface, consommation) sont extraites des bibliothèques afin de caractériser les noeuds correspondants dans le graphe. La dynamique des opérateurs dédiés à l'unité de mémorisation est dimensionnée en fonction de la dynamique maximum des calculs d'adresses qui sont amenés à y être calculés. La dynamique des calculs d'adresses peut être extraite d'une analyse des structures des données de l'application ou fournies par le concepteur du circuit.

Dans notre flot de conception, nous considérons par la suite que nous disposons d'un nombre de bus illimité permettant de transmettre des E/S, d'accéder aux bancs mémoire, de transférer des adresses vers le séquenceur et de retourner les retours d'états vers les contrôleurs. Cela permet d'exploiter pleinement le parallélisme d'accès aux données présentes dans les différents bancs mémoire (synthèse sous contrainte temporelle prioritaire).

Perspective

> Il est possible de réaliser une sélection des bus que l'on souhaite utiliser (largeur, débit, latence) de manière à contraindre la synthèse (plus particulièrement l'ordonnancement) des communications inter-unités. Cela permettrait de limiter les coûts de communication en réalisant un lissage des accès sur les ressources disponibles grâce à l'utilisation de la méthode kanban [Corr05].

Après la sélection des opérateurs, l'allocation va opérer de manière indépendante sur chacune des listes d'opérations à réaliser générant respectivement $Liste_{Operateurs-UT}$ et $Liste_{Operateurs-UMem}$ qui sont les

listes contenant l'ensemble des opérateurs.

4.6.2 Allocation des opérateurs

La quantité d'opérateurs matériels nécessaire à la génération du chemin de données de l'UT sous contrainte de cadence est basée sur le nombre moyen d'opérateurs par tranche de pipeline pour l'unité de traitement. Pour ne pas limiter le parallélisme au sein de l'UT, il est en effet nécessaire de pouvoir disposer des données au plus tôt comme cela a été démontré dans la thèse [Corr05] traitant de la synthèse sous contraintes de mémorisation. Afin de respecter cette exigence, deux approches peuvent être mises en oeuvre afin d'allouer des ressources dans le chemin de données de l'UMem :

Allocation statique préordonnancement - La méthode d'allocation statique des opérateurs pour le séquenceur mémoire est basée sur l'idée qu'il est nécessaire de réaliser au plus tôt les accès aux bancs mémoire afin de ne pas réduire le parallélisme exploitable. Pour permettre au chemin de données du séquenceur mémoire d'exploiter de manière optimum le parallélisme des calculs d'adresses, nous allons allouer pour chaque fonction f le nombre d'opérateurs égal au nombre maximum de calculs de type f en parallèle. L'extraction du parallélisme maximum pour chaque fonction f implémentée dans le séquenceur mémoire est réalisée à l'aide d'un ordonnancement de type *" Force Directed Scheduling "* où, pour chaque cycle, une analyse des opérations exécutables est réalisée en fonction de leur mobilité. Le nombre d'opérateurs à allouer pour chaque fonction peut être exprimé comme mentionné par l'équation 4.11.

$$Nb_{opr-UMem}(f,t) = Max(Nb_{opr}(f,t)) \tag{4.11}$$

Allocation dynamique durant l'ordonnancement - La méthode d'allocation dynamique durant la phase d'ordonnancement, contrairement à la méthode précédente, est basée sur l'idée qu'il peut être plus bénéfique d'allouer des opérateurs au sein de l'UMem lorsque le besoin s'en fait ressentir. L'allocation dynamique d'opérateurs va allouer de nouveaux opérateurs implémentant la fonction f lorsque le nombre d'opérations ordonnançable de type f au cycle courant est supérieur au nombre de ressources matérielles disponibles. Dans ce cas, on crée dynamiquement n opérateurs pour répondre aux besoins immédiats avec n égal au nombre d'opérateurs manquant pour ordonnancer l'ensemble des calculs d'adresses éligibles et en attente. On commencera la phase d'ordonnancement avec un nombre d'opérateur nul au sein du séquenceur mémoire.

C'est donc pendant la phase d'ordonnancement et d'assignation du graphe que l'allocation des opérateurs par type de fonction sera réalisée. Suivant les contraintes appliquées (E/S, mapping mémoire, cadence, allocation des opérateurs dans l'UT, ...), le parallélisme réellement exploitable entre les calculs d'adresses peut être de loin inférieur aux prévisions. Cette technique permet d'adapter le nombre de ressources disponibles lors de l'ordonnancement.

Choix de l'approche - L'allocation statique a pour avantage de réduire la complexité de l'ordonnancement mais peut entraîner une réduction de l'exploitation réelle du parallélisme entre les calculs d'adresses

en les lissant dans le temps. L'allocation dynamique permet quant à elle de s'adapter dynamiquement aux besoins (unités de calculs) lors de l'ordonnancement des opérations mais implique une augmentation de la complexité de ce dernier. Nous avons décidé d'implémenter la seconde méthode dans notre flot de synthèse afin de permettre une meilleure exploitation du parallélisme entre les calculs d'adresses.

4.6.3 Ordonnancement et assignation

Plusieurs modifications ont été nécessaires pour rendre l'algorithme d'ordonnancement par liste compatible avec la prise en compte du chemin de données dans le séquenceur. Comme nous l'avons vu, GAUT dans sa version initiale procède tel que tous les calculs sont implémentés dans un unique chemin de données localisé dans l'UT. Dans le cas présent, nous possédons 2 chemins de données localisés dans différentes unités (UT et UMem) communicant à l'aide des bus de données. La progression du temps est synchrone pour les 2 chemins de données. Les modifications apportées concernent :

1. *La gestion de 2 listes d'opérateurs disjointes :* en fonction de la localité des opérations, l'étape d'assignation cherchera un opérateur libre implémentant la fonction f dans le chemin de données correspondant à sa localité d'implémentation. Aucun changement de localité ne sera effectué durant l'étape d'ordonnancement afin de ne pas augmenter la complexité.
2. *Gestion des conflits d'accès pendant les adressages dynamiques :* lors d'un accès dynamique à la mémoire (lecture/écriture), il est indispensable de geler tous les accès concurrents aux bancs potentiellement concernés. En effet, comme l'accès est dynamique, on ne peut pas prédire a priori quel élément va être ciblé (et donc le banc mémoire cible, si le vecteur est placé de manière non contiguë et multi-bancs). Il faut donc réserver l'ensemble des bancs mémoire qui contiennent au moins 1 élément du vecteur.
3. *Gestion du vieillissement des opérateurs dans les séquenceurs :* a priori avec les applications considérées, le nombre de calculs d'adresses au sein de l'UMem est inférieur au nombre de calculs dans l'UT. Il en va de même pour le nombre d'opérateurs matériels alloués. En contre partie d'une augmentation de la complexité de l'étape d'assignation et de la technique de vieillissement des opérateurs présents dans l'UMem, nous allons augmenter leur taux d'utilisation. L'augmentation du taux d'utilisation de ces derniers est réalisée en remplaçant la technique de vieillissement des opérateurs. Dans le chemin de données de l'UT un opérateur est utilisable/assignable sur le créneau $[T_{Debut}, T_{Debut} + T_{Cadence}[$. Au bout de cette durée, il est déclaré comme étant usé et devient non réutilisable. Nous avons remplacé cette méthode par une utilisation complète des créneaux temporels des opérateurs. Cette modification entraîne une augmentation de la complexité de la recherche d'un opérateur libre durant l'étape d'assignation (par $t\%T_{Cadence}$), mais favorise, en contre partie une meilleure exploitation des ressources matérielles de l'UMem dans le cas d'architectures pipeline.
4. *Allocation dynamique des opérateurs du séquenceur mémoire :* pour chaque étape d'assignation, il faut vérifier qu'aucune opération de calcul d'adresses n'est éligible et dans un même temps non assigné par manque d'opérateurs libres. Si c'est le cas, alors, il est nécessaire d'allouer un nouvel opérateur et de le mettre en "service" afin d'assurer l'exécution au plus tôt de tous les calculs d'adresses.

Algorithme d'une tentative d'ordonnancement

```
Graphe TentativeOrdonnancement( Graphe G )
Tant qu'il reste des opérations à exécuter faire
    ListeOpr = RechercherLesOperationsExecutables( G );
    Pour toutes les opérations de ListeOpr faire
        opr = RechercherOpérateurLibre( opération );
        Si opr = nul alors
            Si loc( opération ) = Mémoire alors
                opr = AllouerNouvelOpérateur( opération );
            Sinon Si loc( opération ) = Mémoire alors
                Passer à l'opération suivante;
            Fin Si
        Fin Si
        AssignerOpérationSurOpérateur( opération, opr );
        RetirerOperation( ListeOpr );
    Fin Pour
    Progression du temps;
    Libérer et faire vieillir les opérateurs de l'UT;
    Libérer et faire vieillir les opérateurs de l'UMem;
    Supprimer les opérateurs usés;
Fin Tant Que
Retourner G;
```

FIG. 4.34 – Algorithme d'une phase d'ordonnancement sous contraintes.

La figure 4.34 décrit le nouvel algorithme d'ordonnancement/assignation qui permet une gestion conjointe des chemins de données de l'UT et de l'UMem.

La gestion des conflits lors des adressages dynamiques est réalisée dans les fonctions de "recherche des opérations exécutables" et "d'assignation des opérations sur un opérateur". La première fonction va vérifier que les unités de mémorisation ciblées lors d'un adressage dynamique sont disponibles. La seconde fonction va, pour chaque opération réalisant un accès à la mémoire (statique ou dynamique), verrouiller les bancs pour le cycle courant de manière à ne pas avoir d'accès concurrent au cours du même cycle.

4.6.4 Machine d'état

Nous avons spécifié que l'ensemble des unités composant notre architecture cible possédait un contrôleur implémenté par une machine à états finis de type Moore, ce qui permet l'exploitation des retours d'états. Dans le cas du chemin de données dédié au séquenceur mémoire, un contrôleur de même nature est ajouté afin d'assurer la coordination des calculs d'adresses. Son implémentation et son fonctionnement sont identiques à ceux des autres contrôleurs des autres unités.

4.6.5 Bilan

Nous venons de présenter dans cette partie une approche innovante consistant à déplacer les calculs d'adresses dynamiques de l'unité de traitement à l'unité de mémorisation. Nous avons présenté une tech-

nique permettant de décider quels sont les calculs d'adresses intéressants à réaffecter afin de minimiser certains paramètres de l'architecture finale. Le flot de synthèse permettant de projeter le modèle de représentation du graphe vers l'architecture matérielle ciblée a également été présenté.

4.7 Conclusions et Perspectives

Dans une première partie de ce chapitre, nous avons présenté l'outil de synthèse architecturale GAUT. Dans sa version initiale, cet outil permet une synthèse automatique des unités de traitement, de mémorisation et de communication sous la contrainte d'une cadence d'itération de l'algorithme. Nous avons ensuite détaillé trois primitives fonctionnelles et leurs implémentations. Dans ce contexte, nous avons défini un ensemble de nouvelles primitives architecturales que nous avons ensuite intégrées à l'architecture générique ciblée par le flot de synthèse. Nous avons détaillé l'implémentation des structures conditionnelles (*if* et *case*) et des accès dynamiques à la mémoire au sein de notre flot de synthèse. Les techniques mises en oeuvre permettent l'implémentation de descriptions algorithmiques contenant de l'indéterminisme à l'aide d'un processus de synthèse basé sur l'exploitation maximum du parallélisme, propriété réservé jusqu'alors aux applications purement déterministes. Une étude menée sur les répercussions du choix du type de contrôleur (Moore/Mealy) et de ses caractéristiques a mis en évidence la nécessité d'un choix d'implémentation réfléchi afin de permettre l'implémentation d'architectures efficaces. *La définition d'une méthode d'ordonnancement des structures itératives en vue de leur implémentation matérielle sous contrainte de cadence sera réalisée dans des travaux de recherche à venir*. Finalement, nous avons proposé une architecture permettant l'implémentation des primitives algorithmiques ainsi que le flot de synthèse associé qui permettra, à terme, la synthèse de l'ensemble des unités (UT/UMem/UCom) avec leurs nouvelles capacités.

Nous avons donc au final proposé un modèle de représentation, une nouvelle architecture ainsi que le flot de synthèse associé permettant d'assurer la gestion de nouvelles primitives fonctionnelles. Ces différents points restent à coder dans l'outil GAUT. Cela sera réalisé après aboutissement des travaux sur l'implémentation des boucles non bornées dont une partie de l'étude a été menée ici. Cette répartition des travaux s'explique par la quantité de développements nécessaires afin de modifier le modèle de représentation interne de l'outil, les méthodes implémentées dans le coeur de synthèse ainsi que la génération de l'architecture décrite en VHDL-RTL.

Dans la deuxième partie de ce chapitre, nous avons présenté un flot d'optimisation pour les calculs d'adresses dynamiques. Avec la première solution proposée, ces dernières sont nécessairement implémentés dans l'UT. Nous avons montré que leur implémentation au sein du séquenceur mémoire pouvait être bénéfique. Afin de permettre l'implémentation des calculs d'adresses dynamiques dans l'UMem, une nouvelle architecture de séquenceur mémoire a été proposée. Ces travaux ont abouti à l'expression d'un nouveau flot de synthèse permettant, dans un premier temps, une transformation du modèle de représentation afin de déterminer les calculs d'adresses devant bénéficier de cette relocalisation. Dans un second temps, nous avons détaillé les différentes étapes permettant la synthèse des deux chemins de données de manière synchrone au sein d'un unique processus. Ce nouveau processus a abouti à la création de nouvelles méthodes de sélection/allocation dédiées au chemin de données du séquenceur mémoire.

Il a également été nécessaire d'adapter les techniques d'ordonnancement et d'assignation utilisées afin d'unifier la gestion des deux chemins de données.

Chapitre 5

Expérimentations Menées sur des Applications TDSI

Ce chapitre est composé de trois expériences réalisées pour deux d'entre elles sur des applications du domaine du traitement du Signal, de l'image et des Télécommunications. La première expérience mettra en évidence l'intérêt de notre approche avec trois exemples pédagogiques. Ces trois exemples présenteront les techniques employées pour gérer les noeuds hiérarchiques puis les structures conditionnelles et pour conclure le transfert des calculs d'adresses dynamiques dans le séquenceur. La seconde expérience montre l'intérêt, en terme d'optimisation du nombre de registres, d'opérateurs et de bus, de la prise en compte du transfert des calculs d'adresses dynamiques dans le séquenceur mémoire dans un flot de conception basé sur la synthèse comportementale. La troisième expérience est un exemple d'utilisation de la synthèse comportementale pour la conception d'une chaîne de transmission numérique composée de plusieurs composants virtuels comportementaux complexes dans un contexte applicatif temps réel.

5.1 Choix des applications

Dans ce chapitre, nous exposons les résultats de synthèses d'unités de traitement réalisées sous contraintes temporelles. Le choix des applications à synthétiser a été guidé par l'expérience acquise au LESTER dans les domaines du traitement du signal, de l'image et des télécommunications.

Nous débutons ce chapitre par la présentation de 3 exemples pédagogiques permettant d'aborder individuellement les points abordés dans cette thèse. Nous commençons par étudier le traitement et l'impact de la synthèse des noeuds hiérarchiques avec l'utilisation des offsets. Nous poursuivons par la gestion des structures conditionnelles pour la réduction des calculs exécutés dans une optique *low-power* sous contrainte temps réel. Enfin nous concluons cette partie consacrée aux exemples pédagogiques par un exemple dédié au transfert des calculs d'adresses dynamiques et à leurs impacts sur l'architecture générée.

La deuxième expérience compare les résultats obtenus par une méthode dite "classique" d'intégration d'application contenant des calculs d'adresses dynamiques (sans séquenceur) avec les résultats obtenus

par la méthode proposée dans ce manuscrit (gestion des adressages dynamiques à l'aide d'un séquenceur, puis avec transfert des calculs d'adresses). Cette expérimentation est menée avec des algorithmes d'estimation de mouvements employés dans les méthodes de compression vidéo telles que *MPEG-2/4*.

Nous terminons enfin ce chapitre avec une application orientée télécommunications. L'application choisie est un MODEM basé sur la norme *DVB-DSNG (Digital Video Broadcasting Digital Satellite News Gatherings)*. Cette norme intervient dans le cadre des transmissions numériques via satellite. Par exemple dans le domaine de la télévision, cette norme fournit, pour les acquisitions en temps réel d'événements, une réponse appropriée à un moindre coût pour établir des liaisons rapides entre des véhicules et des studios. Cette expérience montre l'intérêt de notre approche en terme de synthèse de composants virtuels de niveau algorithmique, flexibles et paramétrables. Cette expérience illustre la capacité de notre méthode et de l'outil GAUT à traiter des problèmes industriels de forte complexité.

5.2 Exemples pédagogiques

Dans cette partie, nous allons décrire le processus de synthèse de trois architectures simples. L'objectif est de montrer l'obtention d'une solution architecturale lorsque la description fonctionnelle contient des noeuds hiérarchiques, des structures conditionnelles et des calculs d'adresses dynamiques. Un bilan permettra de dégager les caractéristiques principales des différentes architectures obtenues.

5.2.1 Synthèse de système à l'aide de noeuds hiérarchiques

Présentation

Comme nous l'avons vu précédemment dans le chapitre 3, les noeuds hiérarchiques sont constitués d'un sous-graphe en interne. Ces sous-graphes une fois datés peuvent faire apparaître des mobilités sur leurs entrées et leurs sorties. Dans ce cas, il est intéressant de remonter cette information au niveau hiérarchique supérieur afin de pouvoir bénéficier du relâchement des contraintes générées lors du calcul des dates ASAP/ALAP des noeuds. Nous allons montrer l'intérêt des offsets lors du calcul des dates d'exécution des noeuds au travers d'un exemple modélisant un filtre FIR 4 points présenté dans la figure 5.1.

La modélisation du filtre FIR présentée en figure 5.1 est réalisée à l'aide de noeuds hiérarchiques. Ces noeuds hiérarchiques ont pour rôle de modéliser/encapsuler les opérations réalisées par la fonction MAC utilisée dans la description fonctionnelle. L'utilisation des noeuds hiérarchiques dans la modélisation permet de réduire la complexité de la modélisation. Le sous-graphe contenu dans les noeuds hiérarchiques (opérations MAC) est détaillé dans la figure 5.2.

Le sous graphe contenu dans le noeud hiérarchique consiste en deux opérations qui sont exécutées de manière séquentielle.

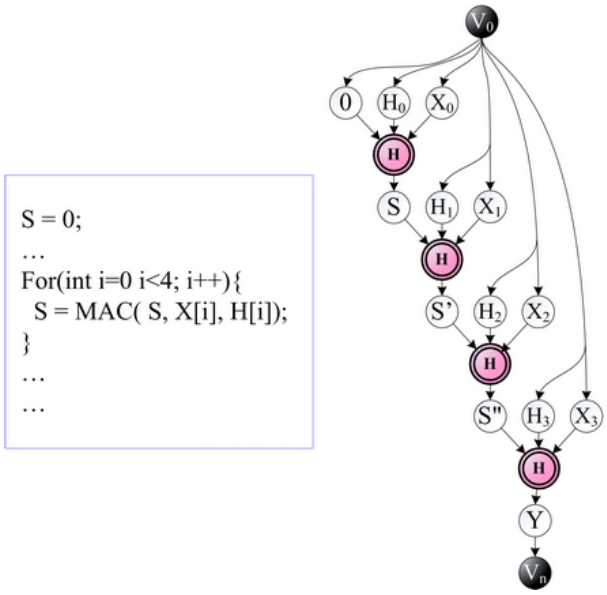

FIG. 5.1 – Modélisation d'un filtre FIR 4 points à l'aide de noeuds hiérarchiques.

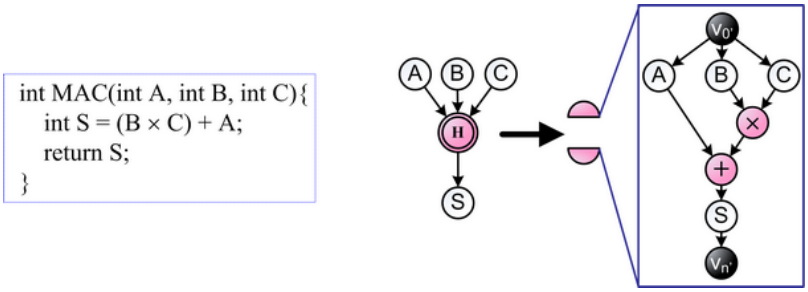

FIG. 5.2 – Composition d'un noeud hiérarchique réalisant une opération MAC.

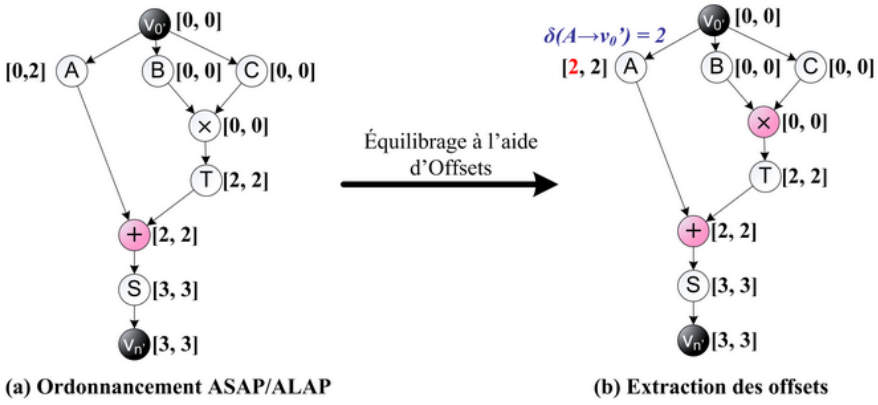

FIG. 5.3 – Ordonnancement du sous graphe et extraction des offsets.

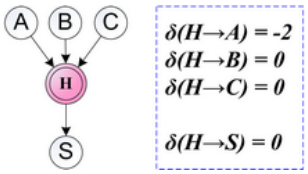

FIG. 5.4 – Expression des offsets de l'opérateur MAC au niveau hiérarchique supérieur.

Mise en oeuvre

Nous allons tout d'abord commencer par calculer les offsets de consommation et de production avant de répercuter ces informations au niveau hiérarchique supérieur. Afin d'extraire les offsets des noeuds hiérarchiques de type *MAC*, il va falloir dans un premier temps ordonnancer le sous-graphe. Pour pouvoir réaliser cette opération nous allons faire l'hypothèse que nous possédons dans l'architecture d'implémentation un additionneur et un multiplieur. Nous considérerons que l'additionneur possède un temps d'exécution de 1 unité de temps et le multiplieur de 2 unités. La première étape va être l'ordonnancement *ASAP*/*ALAP* du sous-graphe, comme présenté dans la figure 5.3a.

L'ordonnancement du sous graphe est réalisé à l'aide de tous les opérateurs disponibles au sein de l'architecture. Cette technique est employée afin de réduire la complexité de la synthèse du graphe. En effet, trouver le nombre d'opérateurs nécessaire pour ordonnancer efficacement un noeud hiérarchique n'est pas une opération triviale. Ensuite, une fois l'ordonnancement réalisé, on *réordonnance* l'ensemble des entrées et des sorties en fonction respectivement de la date au plus tôt de leurs successeurs et de la date au plus tard de leurs prédécesseurs. Les résultats de cette opération sont présentés dans la figure 5.3b. Nous pouvons observer sur l'arc reliant le noeud source v_0 à l'entrée A un offset de consommation. Ce dernier modélise le délai minimum entre le début de l'exécution du noeud et le besoin de l'entrée A pour l'exécution d'un ou plusieurs calculs. Il est maintenant nécessaire de répercuter notre connaissance des offsets dans le sous graphe au niveau hiérarchique supérieur. Les équations permettant ce transfert ont été présentées dans le chapitre 3. La figure 5.4 représente le sous graphe au niveau hiérarchique supérieur avec les offsets adéquats.

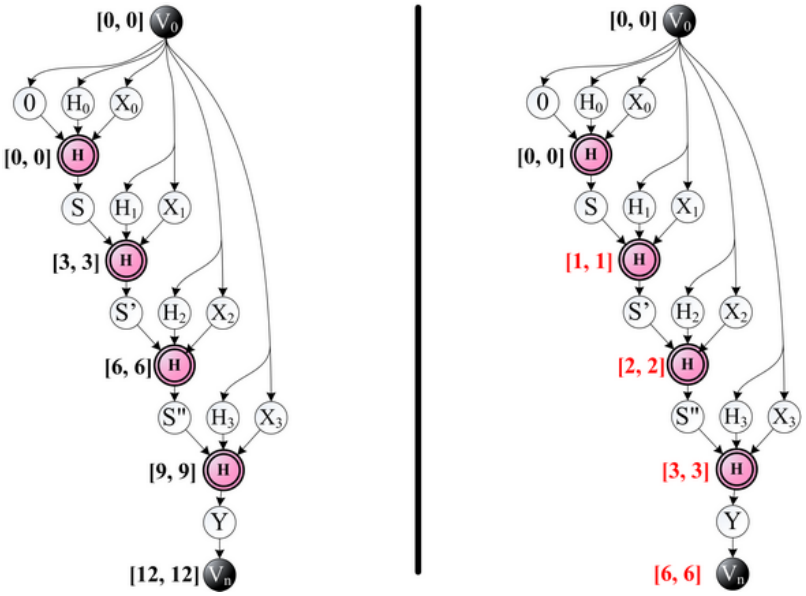

FIG. 5.5 – Ordonnancement du graphe (a) sans utilisation des offsets (b) avec utilisation des offsets.

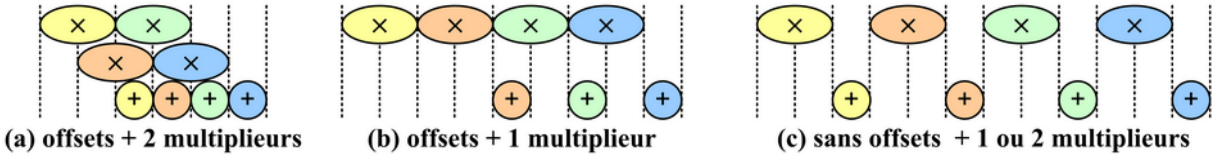

FIG. 5.6 – Comparaison des différents ordonnancements (avec et sans utilisation des offsets).

Nous allons maintenant considérer le graphe modélisant le FIR 4 points. L'ordonnancent est réalisé à l'aide des techniques *ASAP*/*ALAP*. Dans un premier temps, nous présentons les résultats de l'ordonnancement réalisé sans utilisation des offsets de consommation/production dans la figure 5.5a. Le même ordonnancement réalisé en tenant compte des offsets extraites du noeud hiérarchique est présenté dans la figure 5.5b.

Résultats

Le graphe ordonnancé avec prise en considération des offsets viole dans son état actuel les contraintes matérielles posées au départ. En effet, l'exécution des noeuds hiérarchiques est partiellement recouvrante et, dans l'état actuel de l'ordonnancement, 2 multiplieurs seraient nécessaires. Nous présentons dans la figure 5.5 les ordonnancements obtenus avec et sans utilisation des offsets, avec un ou deux multiplieurs disponibles. Les groupes de couleurs utilisés permettent la modélisation des opérations appartenant à un même noeud hiérarchique.

Nous pouvons remarquer la différence qui existe entre les différentes techniques d'ordonnancement en fonction du nombre de cycles d'horloge nécessaire pour exécuter le graphe. Les résultats de cette expérience sont consignés dans le tableau présenté en figure 5.6.

	Ressources matérielles		Nombre de cycles	Gain
	Additionneur	Multiplieur		
Sans utilisation des offsets	1	1	12	
	1	2	12	0%
Avec utilisation des offsets	1	1	9	33%
	1	2	6	50%

FIG. 5.7 – Résultats de l'ordonnancement des noeuds hiérarchiques.

Les résultats de cette expérience montrent clairement que l'ordonnancement d'un graphe en utilisant les offsets (de consommation dans notre exemple), induits par les noeuds hiérarchiques, permet une réduction de la latence grâce à une meilleure utilisation des ressources matérielles.

Conclusion

Cette expérience montre que la prise en compte des offsets de consommation et de production, induits par les noeuds hiérarchiques, est bénéfique lors de la phase d'ordonnancement. Dans notre exemple, nous avons mis en application uniquement les offsets de consommation car la complexité du modèle qu'il aurait fallu employer pour mettre en oeuvre des offsets de consommation et de production aurait nui à la lisibilité graphique. Malgré cela, les gains obtenus sur la latence de l'implémentation et la meilleure utilisation des ressources matérielles grâce à leur utilisation sont bien établis. Cette mise en oeuvre des noeuds hiérarchiques permet de limiter la complexité de la modélisation et de la synthèse de composants virtuels algorithmiques complexes qui ne pourraient être facilement implémentés autrement.

5.2.2 Synthèse optimisée d'une application contenant des branches conditionnelles

Présentation

L'exemple que nous allons maintenant aborder porte sur la gestion des structures conditionnelles afin de réduire le nombre d'opérations effectivement exécutées dans une synthèse sous contrainte temps réel. Comme nous l'avons vu précédemment dans le chapitre 3, les structures conditionnelles sont modélisées dans notre modèle CSFG par des noeuds nommés : *opérations conditionnelles* et *variables conditionnées*. Les opérations conditionnelles possèdent $\{2 \rightarrow n\}$ successeurs dont un seul sera validé en fonction de la condition évaluée. Ces successeurs seront utilisés lors de l'étape de synthèse pour déterminer quels sont les noeuds structurellement mutuellement exclusifs. Nous allons montrer l'intérêt de la gestion des opérations mutuellement exclusives dans le processus de synthèse architecturale. La description comportementale de l'exemple et le *CSFG* correspondant sont présentés dans la figure 5.8.

Dans l'exemple présenté nous avons volontairement effectué certaines simplifications dans la représentation afin d'en accroître la lisibilité graphique : les noeuds source et puits ont été omis.

```
int function(int A, int E){
    If( E ≤ 0 ){
     A = A + 1;
     Y = T[A] × 5;
    }else{
     A = A + 2;
     Y = 3 × T[A];
    }
    return Y;
}
```

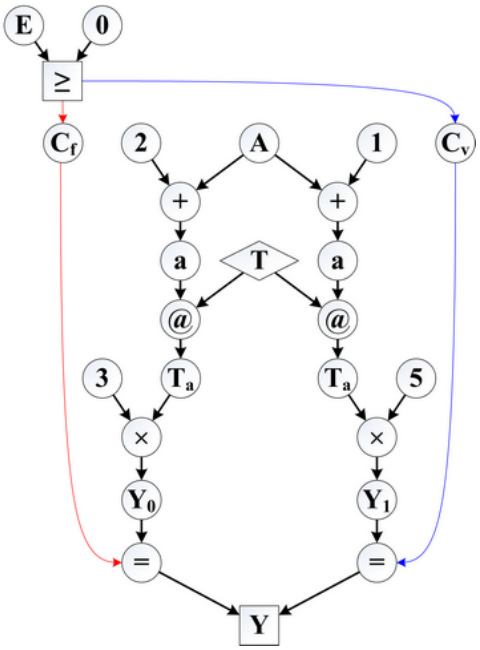

FIG. 5.8 – Exemple pédagogique modélisant une structure conditionnelle.

Afin de simplifier la comparaison des solutions architecturales obtenues, nous avons représenté le SFG actuellement mis en oeuvre par *GAUT* correspondant à notre exemple (figure 5.9a, exécution spéculative de toutes les opérations contenues dans les branches). Avec notre approche, la première étape à effectuer, en vue de l'implémentation matérielle, est de projeter sur le graphe les annotations architecturales nécessaires à la prise en compte des retours d'états et des transferts de données associés. Le résultat de la transformation du graphe est présenté dans la figure 5.9b.

Les dépendances de contrôle (traits en pointillés dans la figure 5.9) ne font pas partie des sémantiques du modèle CSFG. Les arcs ont été ajoutés afin de modéliser les informations obtenues par remontées de chemins afin d'obtenir les opérations mutuellement exclusives. Dans la "réalité" nous obtenons suite aux remontées des chemins entreprises à partir des noeuds d'affectation conditionnée $(=)$ *la liste de toutes les opérations mutuellement exclusives (appartenant à des branches différentes d'un même noeud opération conditionnelle).*

Mise en oeuvre

La phase de projection architecturale appliquée à notre exemple a permis de mettre en exergue les transferts de données entre les unités et pour certains leurs conditionnements au résultat du calcul conditionnel. De plus, nous pouvons observer l'apparition des noeuds (C_f, C_v) qui sont localisés au sein de l'UMem. L'étape de projection architecturale a aussi réalisé l'annotation des arcs reliant les noeuds (C_v, C_f) à leurs successeurs d'un poids équivalent à la pénalité du retour d'état (dans notre exemple $\lambda = 3$). Ces noeuds modélisent le retour d'état qui doit être transmis au contrôleur du séquenceur mémoire pour que ce dernier soit capable de décider des lectures/transferts de données qu'il devra effectuer. Nous avons donc dans notre modèle deux retours d'états distincts qui ont pour destination respective : le

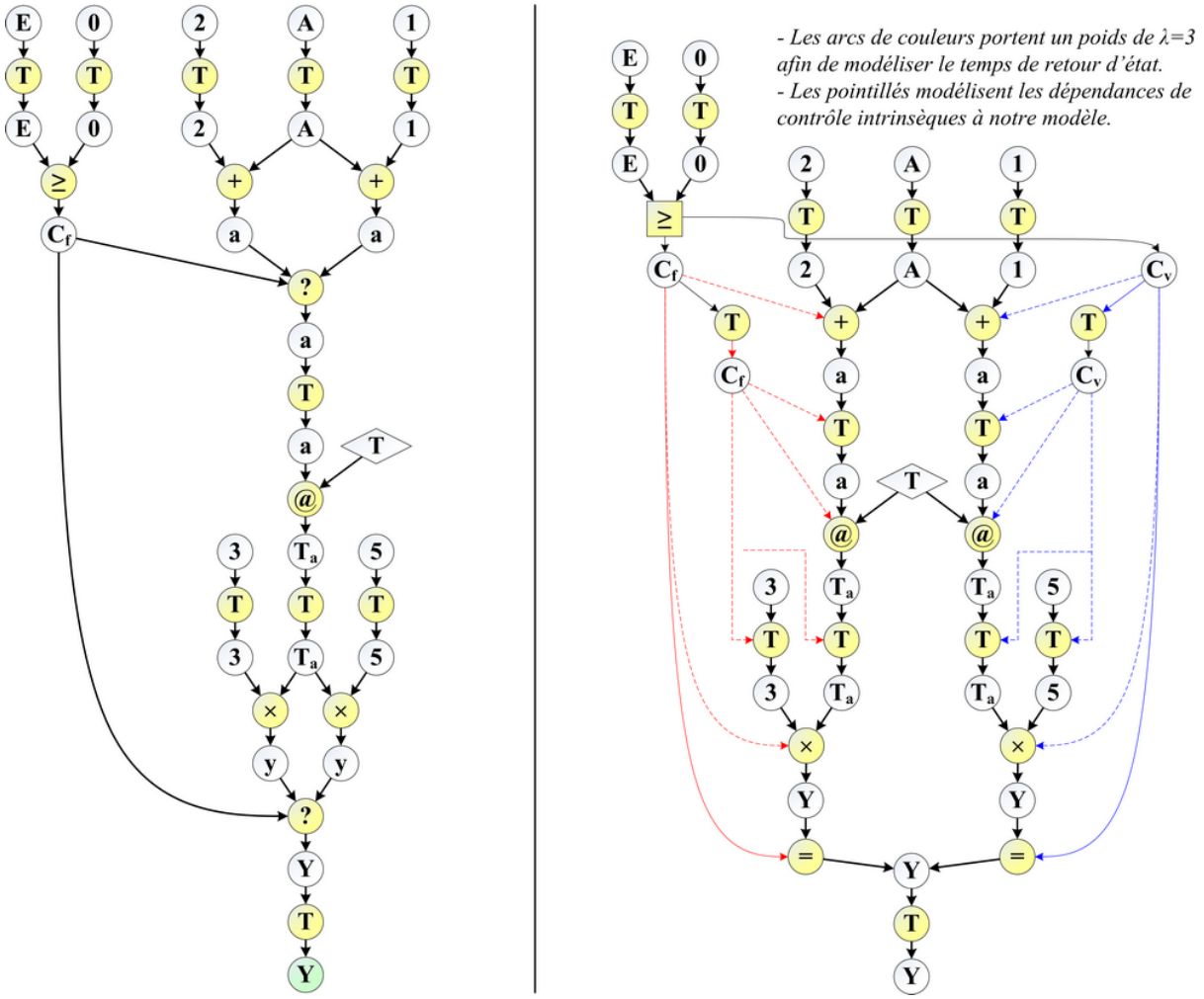

FIG. 5.9 – Modélisation de l'exemple conditionnel (a) avec traitement des conditions dans le chemin de données (b) à l'aide des techniques de retour d'états vers les controleur de l'UT/UMem.

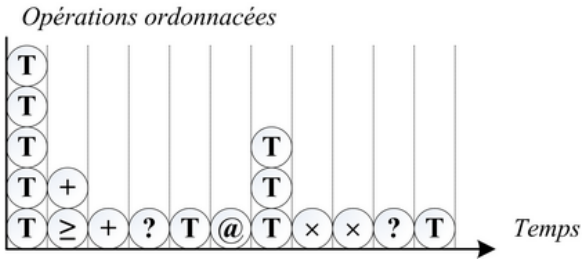

FIG. 5.10 – Ordonnancement actuellement réalisé par l'outil GAUT.

contrôleur de l'UT et le contrôleur du séquenceur mémoire.

Dans le cadre de notre exemple nous allons supposer que la phase de sélection et la phase d'allocation ont été réalisées et qu'elles ont conduit à allouer un additionneur et un multiplieur à l'architecture. Dans l'état actuel des choses, notre graphe *CSDF* peut être ordonnancé de manière quasi-équivalente à un graphe flot de données classique à condition de ne pas oublier les temps de retour d'état pondérant les arcs. Nous allons ordonnancer les graphes présentés dans les figures 5.9a et 5.9b. L'ordonnancement du graphe 5.9 sous contraintes matérielles a été effectué et les résultats sont présentés dans la figure 5.10.

En ce qui concerne l'ordonnancement sous contraintes matérielles du graphe 5.9b, nous avons pris en compte durant l'ordonnancement des opérations les pénalités dues aux retours d'états qui étaient modélisées à l'aide de poids sur les arcs. Ensuite, lors de l'assignation des opérations sur les opérateurs matériels, nous avons cherché dans la liste des opérations exécutables, des opérations mutuellement exclusives exécutables sur le même opérateur. Les résultats obtenus sont présentés dans la figure 5.11. La première rangée d'opérations modélise l'exécution des opérations non conditionnelles dans le temps. La seconde et la troisième rangée modélisent respectivement les opérations qui seront exécutées selon que le résultat de la condition sera *vrai/faux*. L'exécution de ces 2 branches ne peut pas être réalisée lors d'un même scénario d'exécution. Les opérations colorées en gris modélisent l'ensemble des opérations qui partage un même opérateur (opérateur arithmétique ou bus de communication). Il faut noter qu'un seul transfert est réalisé vers la mémoire pour le couple de noeuds (C_f, C_v) car il s'agit physiquement d'un simple bit de retour d'état.

Nous observons que l'ensemble des opérations comprises dans les deux branches conditionnelles partage le même matériel (dans notre exemple). Les machines d'état gérant l'architecture dans le cas de l'ordonnancement proposé avec partage des opérateurs devront, en fonction du résultat de la condition, choisir la branche conditionnelle à effectuer. Cette sélection se fera de manière synchrone entre le contrôleur de l'UT et de l'UMem mais avec n cycles de retard pour cette dernière (n modélisant le temps de transfert d'une donnée de l'UT vers l'UMem).

Résultats

La synthèse de ces deux modèles ciblant des architectures différentes a été réalisée. Les résultats comparatifs entre ces deux approches sont résumés dans la figure 5.12.

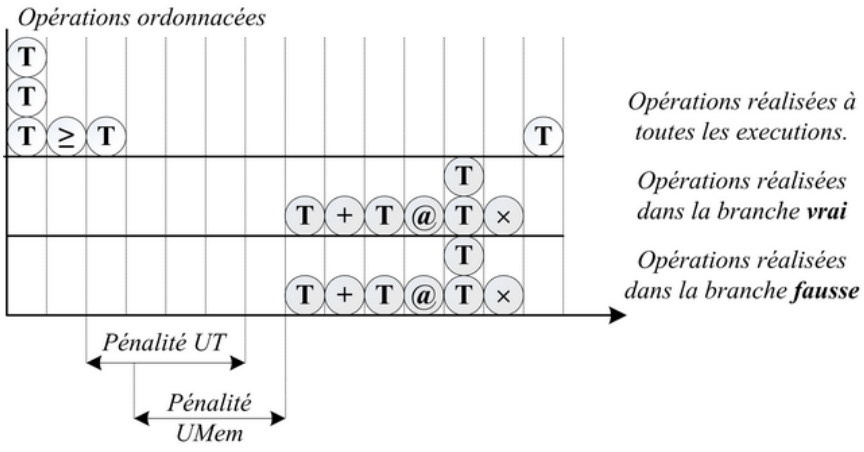

FIG. 5.11 – Ordonnancement réalisé afin d'utiliser les propriétés des opérations ME.

	Latence (cycles)	Nombre de traitements exécutés			Opérateurs mis en oeuvre		
		transferts	opérations	mémorisations	add.	mult.	eqmux
Implémentation parallèle	11	10	8	16	1	1	1
Implémentation avec partage	14	8	4	12	1	1	0

FIG. 5.12 – Comparaison des architectures synthétisées sans et avec partage des opérateurs.

Des différences importantes apparaissent entre les deux architectures qui ont été synthétisées. Notre méthode permettant de partager les opérateurs entre opérations mutuellement exclusives permet de réduire le nombre des opérations à exécuter, impactant directement sur la consommation énergétique du circuit. De plus cette technique permet de s'abstenir de l'allocation de composants de type *eqmux* qui permettent dans le cas d'une implémentation sans partage, de traiter les structures conditionnelles dans le chemin de données. La croissance de la complexité des contrôleurs de différentes unités est compensée totalement ou partiellement par les gains réalisés sur les besoins en registres (réduction des mémorisations à effectuer), en transferts simultanés (réduction du nombre de bus). En contrepartie cette solution engendre une augmentation de la latence de l'application à cause du temps de retour d'état dans l'architecture pipeline.

Conclusion

Cette expérience a montré que la prise en compte du partage des opérateurs lors de la synthèse permet d'optimiser le nombre d'opérateurs, de registres et de bus à mettre en oeuvre dans l'unité de traitement. De plus, cette approche permet une réduction de l'activité de l'unité de traitement en n'exécutant que les opérations dont la condition d'exécution est validée. La conséquence est un accroissement de la latence de l'application dans notre exemple dû au temps de retour d'états. L'augmentation de la latence est un paramètre spécifique à chaque application synthétisée. Dans certains cas, cette augmentation peut être nulle (si, par exemple, on possède des calculs non conditionnés à exécuter durant le retour d'état). Cette méthode de partage des opérateurs permettraient également de réaliser la synthèse d'applications composées de plusieurs scénarios d'exécution au sein de la même description *(si condition alors réaliser une fft64 sinon réaliser une fft32)* sans besoin de matériel supplémentaire. Cette approche n'est toutefois pas considérée dans ces travaux car elle nécessite des modifications dans la structure/synthèse de l'unité de communication.

5.2.3 Optimisation par projection architecturale des calculs d'adresses

Présentation

Afin de bien comprendre l'intérêt des transferts d'adresses de l'UT vers le séquenceur de l'UMem nous allons dans cet exemple pédagogique mettre en évidence ses avantages. Pour illustrer notre flot d'optimisation et de synthèse, nous allons procéder à une transformation étape par étape de l'exemple présenté dans la figure 5.13. Le graphe modélisant la description algorithmique a été annoté par la localité des différents noeuds (opération, mémorisation).

Dans l'exemple présenté ci-dessous nous avons effectué certaines simplifications dans la représentation du graphe afin d'en accroître la lisibilité : les noeuds source et puits ont été omis et les noeuds d'adressage dynamiques en lecture sont représentés par des opérations notées @ au lieu de @r.

Le graphe (figure 5.13) modélise une application où l'indice i permettant l'accès à $T[i]$ est calculé dynamiquement. Cet indice est ensuite réutilisé pour calculer $j = i+1$ afin d'accéder à la donnée $T[i+1]$. Dans notre exemple nous considérons que la structure de donnée T contient 256 éléments qui sont codés sur 32 bits. Les traitements réalisés au sein de l'UT sont donc opérés avec des opérateurs 32 bits et

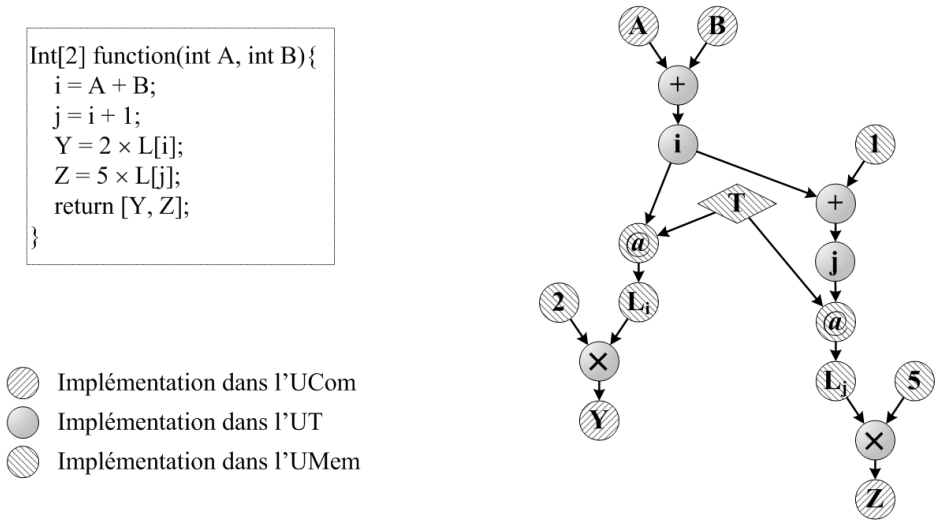

FIG. 5.13 – Description fonctionnelle et modèle de représentation associée (adressage dynamique).

l'adressage de la structure T requiert 8 bits.

Mise en oeuvre

Dans un premier temps nous ajoutons des noeuds de transferts au graphe afin de le rendre cohérent (cf. chapitre 4). Cette transformation va permettre d'obtenir un graphe où l'ensemble des transferts sera modélisé de manière explicite (figure 5.14).

La figure 5.14 représente la description comportementale une fois les projections architecturales appliquées et les opérations de transfert ajoutées. Dans ce graphe nous pouvons dénombrer 11 transferts de données nécessaires entre les différentes unités. Si toutes les opérations $(+, @r, \times)$ sont annotées d'une durée égale à 1 unité temporelle, nous obtenons une latence de 8 unités avant de pouvoir transmettre les résultats $[Y, Z]$ vers le reste du système.

Dans un second temps nous transformons le modèle de représentation afin de réaffecter les calculs d'adresses dynamiques de l'UT vers le séquenceur de l'UMem. Les calculs ainsi transférés seront effectués dans le chemin de données interne au séquenceur mémoire comme cela a été détaillé dans le chapitre 4. Cette transformation vise à réduire le nombre des transferts entre les différentes unités de l'architecture, induisant une réduction possible du nombre de bus/registres à mettre en oeuvre.

Le graphe (figure 5.14) est modifié afin de transférer le calcul de l'indice j (qui joue ici le rôle d'adresse car nous avons une unique structure) au sein de l'UMem (figure 5.15a). Nous nous intéressons dans un premier temps à j car l'indice i est nécessairement transféré en mémoire pour réaliser la lecture de $T[i]$. Ce changement de localité est validé par notre métrique de décision car ce déplacement va réduire le nombre de transferts d'une unité. De plus la taille de l'opérateur à mettre en oeuvre dans le séquenceur est de largeur bien inférieure (8 bits contre 32 bits dans l'UT). Cette transformation est modélisée par le changement de localisation de la seconde opération $(+)$ et de ses arguments. Cette modification de la description comportementale permet de réduire le nombre de transferts nécessaires à 9. La contrepartie

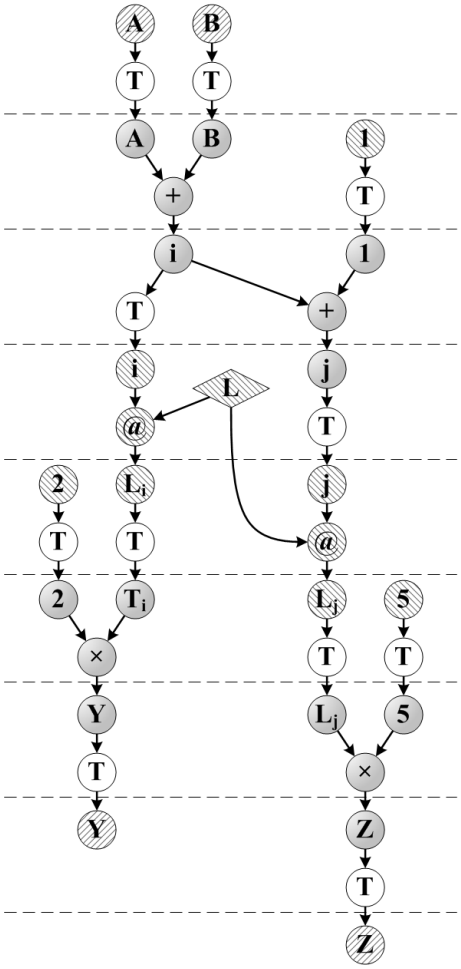

FIG. 5.14 – Représentation après projection architecturale (insertion des noeuds de transferts) en considérant la durée d'un transfert (T) à 1 cycle.

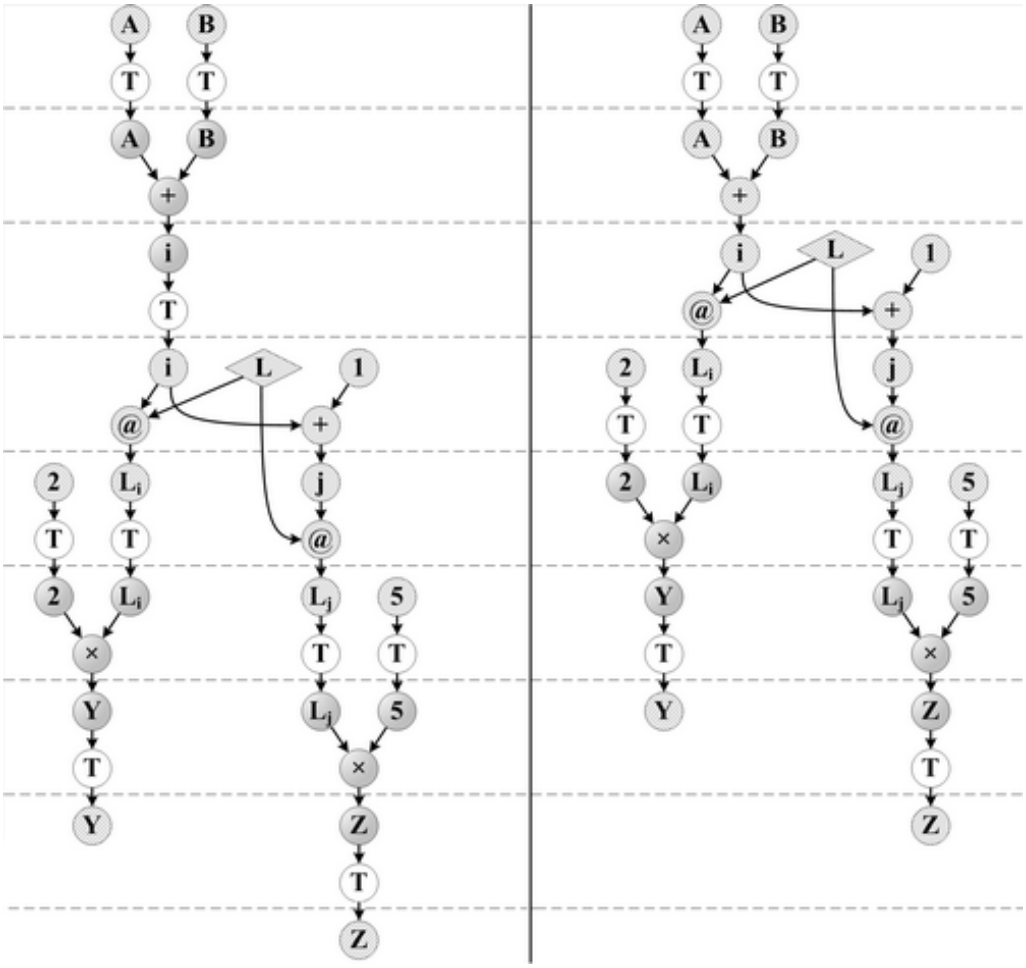

FIG. 5.15 – Calculs d'adresses dynamiques ($T = 1cycle$) (a) calculs d'adresses partiellement transférés (b) calculs d'adresses totalement transférés.

issue de cette optimisation est la génération d'un chemin de données interne à l'unité de mémorisation.

Le graphe obtenu en figure 5.15a peut être encore optimisé déplaçant le calcul de l'indice i au sein du séquenceur mémoire. Les résultats obtenus après transfert sont détaillés dans la figure 5.15b. Cela permet de regrouper l'ensemble des calculs d'adresses au sein de l'UMem, afin de réduire le nombre de transferts de données/adresses entre les différentes unités. A l'aide de cette seconde modification nous avons réduit le nombre de transferts inter-unités à 8. Le déplacement du calcul d'adresse et la réduction du nombre de transferts à réaliser impacte sur la latence du chemin critique qui vaut maintenant 7 unités (contre 8 précédemment).

Il faut, au final, pour implémenter notre modèle sur l'architecture cible, un multiplieur dans l'UT et un additionneur dans le séquenceur de l'UMem.

	Nombre opérateurs			Nombre mémorisations		Nombre transferts	Latence		
	Add (32b) UT	Mult (32b) UT	Add (8b) UMem	UT	UMem		*T=1*	*T=2*	*T=5*
Solution UT (totale)	1	1		11	4	12	9	13	25
Solution UMem (partielle)	1	1	1	9	4	9	9	13	25
Solution UMem (totale)		1	1	6	6	8	8	11	20

FIG. 5.16 – Comparaison des architectures synthétisées sans et avec partage des opérateurs.

Résultats

Le tableau figure 5.16 regroupe les caractéristiques des différentes transformations que le modèle de représentation a subies. Si nous observons les résultats obtenus nous pouvons nous rendre compte que la solution intermédiaire, consistant à ne déplacer qu'un seul des 2 calculs d'adresses dans le séquenceur mémoire, n'est pas intéressante : cette dernière permet bien de réduire le nombre de transferts/mémorisation de données mais implique la double allocation d'un additionneur (un dans l'UT et un dans l'UMem). La solution consistant à transférer l'intégralité des calculs d'adresses dans l'UMem, est par contre, bénéfique dans ce cas précis car cela permet de se passer de l'allocation d'un additionneur de la largeur du chemin de données dans l'UT. Cette allocation est remplacée par celle de l'additionneur 8 bits dans l'UMem qui convient pleinement aux besoins des calculs d'adresses. Cette transformation permet de réduire la taille de l'additionneur utilisé pour réaliser les calculs ainsi que la taille des registres qui lui sont associés. Les analyses des graphes, après leur transformation par réaffectation des calculs d'adresses, sont détaillées dans la figure 5.16.

Si nous observons attentivement l'évolution du nombre de transferts de données / mémorisations nécessaires, nous remarquons que les besoins diminuent lorsque les calculs d'adresses sont transférés de l'UT vers le séquenceur de l'UMem. Il en est de même pour le temps de traversée du graphe (latence) qui est réduite lorsque les calculs sont effectués dans l'UMem. Il est à remarquer que la réduction de la latence est d'autant plus importante que les connexions entre les unités sont lentes (bus pipeline, basse fréquence pour de la faible consommation, etc.). En fonction du parallélisme existant entre les transferts et les mémorisations présentes dans le graphe, cette réduction peut se matérialiser dans l'architecture ciblée sous contrainte de cadence par une réduction des ressources matérielles à mettre en oeuvre. Cette analyse sera plus amplement détaillée dans l'expérimentation de notre approche appliquée aux méthodes d'estimation de mouvements.

Conclusion

Au travers de cet exemple pédagogique nous venons de démontrer l'intérêt que peut présenter le transfert des calculs d'adresses au sein du séquenceur mémoire pour : réduire le nombre de transferts inter-unités, optimiser la taille des opérateurs employés pour calculer dynamiquement les adresses et diminuer le nombre de mémorisation à réaliser dans l'architecture (impliquant une réduction du nombre de commandes provoquant une baisse des commutations des contrôleurs). De plus, cette relocalisation des calculs est bénéfique car les constantes utilisées dans les calculs d'adresses (adresse de base, pas d'incrémentation, ...) sont stockées en mémoire évitant des transferts inutiles vers l'UT. Ces résultats permettent de valider notre motivation dans le domaine de l'optimisation des calculs d'adresses dynamiques par réaffectation de ces derniers de l'UT vers l'UMem. Cette motivation est d'autant plus grande que l'hypothèse selon laquelle les transferts de données possèdent une durée de 1 cycle n'est pas toujours vérifiée. Comme nous venons de le montrer les bénéfices générés par le transfert des calculs d'adresses peuvent, selon les applications traitées, se révéler être importants.

5.3 "Block Matching" pour la compression vidéo

Cette expérience se propose de mettre en avant l'intérêt de la relocalisation des calculs d'adresses dynamiques dans un flot de synthèse de haut niveau sous contrainte temps réel. Nous allons optimiser à l'aide de notre méthodologie différentes implémentations d'algorithmes d'estimation de mouvements basés sur la comparaison de blocs utilisée dans les normes MPEGx [Agha03] et h26x. Nous avons retenu les méthodes suivantes : la recherche en trois itérations (3SS [Koga81]), la recherche orthogonale [Puri87] et la recherche en croix (CSA [Ghar90]). Nous rappelons dans un premier temps les principes sur lesquels reposent les méthodes d'estimation de mouvement. Nous décrivons l'étape d'optimisation présynthèse (réaffectation des calculs d'adresses dynamiques) qui permettra de transférer les calculs d'adresses de l'UT vers le séquenceur de l'UMem. Puis, dans un second temps, nous présentons les résultats de synthèse que nous avons obtenus.

5.3.1 Présentation

L'augmentation des capacités de la qualité des flux multimédias implique une augmentation du débit nécessaire aux transferts de données entre les systèmes nomades. Il est évident que ces débits sont peu compatibles avec les espaces de stockage des applications embarquées. En effet une image d'une vidéo non compressée occupe une taille d'environ 1 Mo. Afin d'obtenir une vidéo fluide il est nécessaire de posséder des débits de l'ordre de 25 images par seconde. Ces transferts nécessitent un flux de données d'environ 30 Mo/s. Afin de réduire les quantités d'informations constituant les flux vidéo des travaux ont été menés. Pour cela, l'objectif est d'exploiter les redondances temporelles, fréquentielles et spatiales présentes dans les séquences d'images composant les flux vidéo.

Une des techniques consiste à exploiter la corrélation temporelle inter-image. La redondance temporelle présente dans une séquence est réduite par estimation du mouvement de l'image courante par rapport à une image précédente et/ou suivante. Cette prédiction de mouvement est réalisée sur des sous-ensembles

de l'image de taille 16×16 pixels appelés macroblocs (ou sur des blocs de 8×8 pixels optionnellement) et consiste à chercher, dans une fenêtre de recherche de l'image de référence, le macrobloc le plus ressemblant par calculs de distorsion. La partie d'estimation de mouvements à l'intérieur d'un codeur *MPEGx* ou *H26x* est de loin la fonction la plus complexe. Cette dernière influe de plus sur les taux et la qualité de la compression. Parmi les techniques d'estimation de mouvement, les méthodes par appariement de blocs *(Block-Matching Algorithm : BMA)* sont de loin les méthodes d'estimation de mouvements les plus utilisées par les standards vidéo. Ce choix est dû à leur facilité de mise en oeuvre et d'un bon compromis entre complexité et efficacité de codage. On ne s'intéressera donc ici qu'à ces dernières méthodes. L'opérateur de distorsion le plus souvent utilisé est basé sur la somme des valeurs absolues des différences :

$$\Delta = \sum_{y=0}^{m-1} \sum_{x=0}^{n-1} |I_1(y,x) - I_2(y,x)| \times \alpha(y,x)$$

L'algorithme de *bloc-matching* le plus simple est la recherche exhaustive ; il sélectionne l'optimum parmi tous les vecteurs possibles à l'intérieur de la fenêtre de recherche. Le nombre élevé d'opérations qu'il nécessite rend cependant cette approche inadaptée pour la plupart des applications temps réel. De nombreux algorithmes de recherche rapides ont été proposés. Ils transforment l'algorithme qui était déterministe en approches contenants de l'indéterminisme. Parmi les plus connus on trouve, la recherche en trois itérations (3SS [Koga81]), la recherche logarithmique 2D (2DLOG [Jain81]), la recherche orthogonale (OSA [Puri87]), la recherche en croix (CSA [Ghar90]), et d'autres plus récemment développées [Furh96] [Jamk02] [Lam03] [Wu05], etc. Ces méthodes ont des performances qui dépendent du type de mouvement contenu dans la scène (risque de convergence vers un minimum local). De plus, elles réduisent fortement la régularité de l'EM. En revanche, elles permettent une forte réduction de la complexité de l'EM. Les algorithmes d'estimation de mouvements que nous analyserons/synthétiserons sont présentés dans la figure 5.17.

Suivant la norme ciblée (MPEG-x ou h26x), la technique d'estimation de mouvement varie en terme de taille de la fenêtre de recherche, de dimension du bloc élémentaire et de dynamique des données. L'EM travaille sur des blocs de luminance qui, par défaut, sont de taille 16×16 pixels (appelés macroblocs), et dont chaque pixel est codé sur 8 ou 12 bits.

5.3.2 Optimisation et intégration des différents algorithmes

Dans cette section, nous comparons les résultats obtenus par une méthode dite "classique" (sans utilisation de séquenceur mémoire) avec la méthode proposée dans ce manuscrit (utilisation de séquenceurs mémoire) dans des applications contenant de l'indéterminisme. Cette dernière méthode sera déclinée sous deux types d'architectures différentes : une première où l'ensemble des calculs est implémenté dans l'UT et la seconde où nous allouons un chemin de données dans le séquenceur mémoire de l'UMem. Nous observerons dans un premier temps les résultats obtenus après la première phase d'optimisation par relocalisation des calculs d'adresses, puis dans un second temps nous nous intéresserons aux résultats de la synthèse architecturale des solutions à base de séquenceurs.

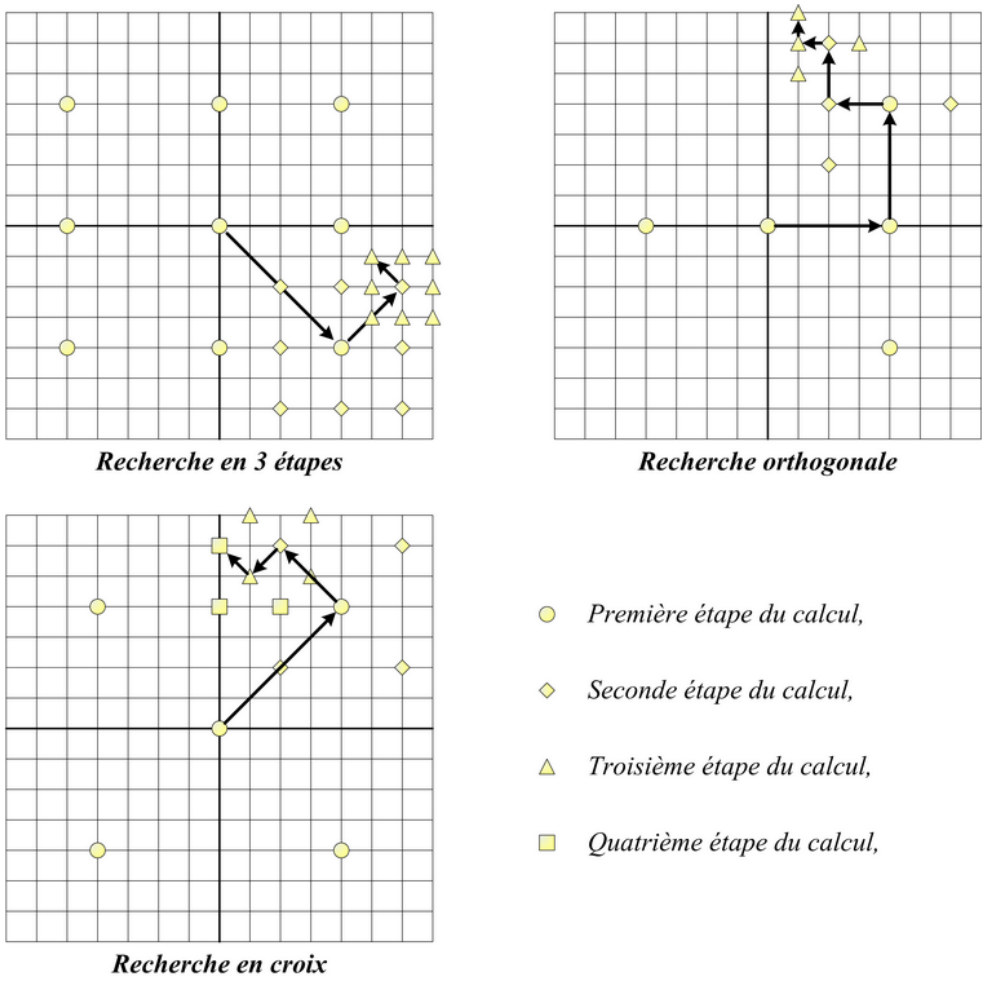

FIG. 5.17 – Exemples de recherches effectuées par les algorithmes d'estimation de mouvement que nous avons sélectionné.

5.3.3 Mise en oeuvre

Nous supposons dans cette première partie d'expérience que l'ensemble des algorithmes d'estimation de mouvements est décrit au niveau comportemental. A partir de ces descriptions nous allons dans un premier temps générer les modèles CSFG correspondants afin de pouvoir les traiter à l'aide de notre flot de synthèse. Avant cela nous allons tout d'abord étudier la dynamique des calculs en fonction de la dynamique du codage de la luminance. Pour cela, nous allons considérer 4 cas différents :

1. *Macrobloc de taille 8x8 avec une luminance codée sur 8 bits* - Dans ce cas, chaque calcul de différence absolue entre 2 pixels (image de référence, image suivante) possède une dynamique maximale de 8 bits. La somme des valeurs absolues des différences sur l'ensemble du macrobloc nécessite 64 sommations des résultats. Dans le pire cas, le résultat de cette somme doit être codé sur 14 bits. L'architecture que nous ciblons possède un chemin de données de taille constante ; cela impose que l'ensemble des opérateurs/registres aura une dynamique de 14 bits. Le codage des adresses pourra quant à lui être réalisé sur 6 bits (64 données à adresser).
2. *Macrobloc de taille 8x8 avec une luminance codée sur 12 bits* - Dans le pire cas, le résultat de la somme des différences absolue doit être codé sur 16 bits. L'architecture que nous ciblons possède un chemin de données de taille constante, cela impose que l'ensemble des opérateurs/registres aura une dynamique de 16 bits. Le codage des adresses pourra quant à lui être réalisé sur 6 bits (64 données à adresser).
3. *Macrobloc de taille 16x16 avec une luminance codée sur 8 bits* - Dans le pire cas, le résultat de la somme des différences absolue doit être codé sur 16 bits. L'ensemble des opérateurs/registres du chemin de données de l'UT aura une dynamique de 16 bits. Le codage des adresses pourra quant à lui être réalisé sur 8 bits (256 données à adresser).
4. *Macrobloc de taille 16x16 avec une luminance codée sur 12 bits* - Dans le pire cas le résultat de la somme des différences absolue doit être codé sur 20 bits. L'ensemble des opérateurs/registres du chemin de données de l'UT aura une dynamique de 20 bits et le codage des adresses pourra être réalisé sur 8 bits (256 données à adresser).

L'étude de la dynamique va nous permettre dans la suite de notre flot de synthèse d'utiliser des bibliothèques de composants adaptées (nombre de bits) lors de la synthèse. Grâce à ces informations, nous pourrons comparer efficacement les 2 approches (séquenceur avec ou sans chemin de données). Nous allons maintenant passer à la génération des modèles de représentation et à leur optimisation/synthèse. *Il est important de noter que dans la philosophie de l'outil GAUT la largeur du chemin de données est fixe. Cette hypothèse impose que tous les opérateurs alloués dans l'UT possèdent la même largeur (maximum de la dynamique des opérations). Nous avons étudier la dynamique des données afin de pouvoir instancier la cas échéant un chemin de données adapté à la dynamique des adresses dans l'UMem.*

Une fois les graphes CSFG extraits des descriptions algorithmiques nous allons, avant de réaliser la synthèse d'architecture proprement dite, optimiser ces graphes par réaffectation des calculs d'adresses dans le chemin de données du séquenceur lorsque ces mouvements sont bénéfiques. De ces transformations, nous pouvons extraire des statistiques sur les graphes ainsi obtenus. Afin de comparer nos résultats, nous étudions 3 types d'implémentations :

Architecture	MB	Opérations			Nombre de transferts	Mémorisations		
		UT	UMem	Total		UT	Umem	Total
Contrôle	8x8	11310		11310	14350	20886		20886
Séq. + UT		7493	0	7493	2560	9248	2049	11297
Séq. + Umem		6469	1024	7493	1308	7712	2333	10045
Contrôle	16x16	44910		44910	38430	64187		64187
Séq. + UT		29765	0	29765	9013	34679	9000	43679
Séq. + Umem		25669	4096	29765	4921	30571	9016	39587

FIG. 5.18 – Résultats après réaffectation pour la méthode "Recherche en 3 étapes".

1. L'architecture de type *contrôle* dans laquelle nous n'utilisons pas de séquenceur mémoire. L'ensemble des accès aux données est réalisé par le dépôt de l'adresse sur le bus d'adresse et la récupération de la donnée *n* cycles plus tard. Dans cette architecture tous les calculs d'adresses sont réalisés par les unités de traitement (accès statiques et dynamiques).
2. Dans la seconde implémentation nommée *séquenceur + UT* nous utilisons la première approche à base de séquenceur où tous les calculs d'adresses sont réalisés dans l'UT. Le séquenceur a alors pour rôle de router les calculs d'adressages dynamiques vers les bancs mémoire puis de transférer les données vers l'UT (cas d'une lecture) ou de router la donnée reçue ainsi que le calcul d'adresse vers le bon banc mémoire (cas d'une écriture).
3. Avec la troisième implémentation nommée *séquenceur + UMem* nous mettons en oeuvre l'architecture du séquenceur permettant d'effectuer les calculs d'adresses dynamiques au sein d'un chemin de données optimisé pour eux.

L'ensemble des optimisations présynthèse est réalisé à l'aide d'un outil entièrement automatique qui analyse le graphe et qui en fonction des contraintes spécifiées par le concepteur, optimise les calculs d'adresses dynamiques en conséquence. L'outil fournit aussi un certain nombre de statistiques permettant de comparer les graphes obtenus. Ces informations sont détaillées dans la prochaine partie.

5.3.4 Résultats d'optimisation et de synthèse

Les résultats post-optimisations sont présentés dans les figures 5.18, 5.19 et 5.20 en fonction de l'algorithme d'estimation considéré.

Le tableau 5.18 présente les résultats obtenus post-optimisation des calculs d'adresses. Ces résultats permettent d'observer dans un premier temps la réduction du nombre de calculs d'adresses à réaliser dans les approches à base de séquenceur mémoire, en comparaison avec l'approche de type contrôle qui ne les emploie pas : les approches à base de séquenceurs permettent de réduire le nombre de calculs (adresses) à réaliser ; dans notre cas les gains sont de 33%. Dans un second temps, nous pouvons observer les différences entre les 2 solutions à base de séquenceur (calculs d'adresses dynamiques dans l'UT versus dans l'UMem). La solution possédant un chemin de données dédié au sein du séquenceur de l'UMem réduit de manière significative une fois encore le nombre de transferts nécessaire entre les deux unités,

Architecture	MB	Opérations			Nombre de transferts	Mémorisations		
		UT	UMem	Total		UT	Umem	Total
Contrôle	8x8	5886	0	5886	7462	10869	0	10869
Séq. + UT		4006	0	4006	1586	5175	1267	6442
Séq. + Umem		3380	626	4006	774	4230	1400	5630
Contrôle	16x16	23358		23358	19989	33386		33386
Séq. + UT		15910	0	15910	5918	19265	5916	25181
Séq. + Umem		13380	2530	15910	3373	16740	5896	22636

FIG. 5.19 – Résultats après réaffectation pour la méthode "Recherche en orthogonale".

Architecture	MB	Opérations			Nombre de transferts	Mémorisations		
		UT	UMem	Total		UT	Umem	Total
Contrôle	8x8	7693	0	7693	9758	14204	0	14204
Séq. + UT		5168	0	5168	1920	6533	1537	8070
Séq. + Umem		4400	768	5168	846	5381	1615	6996
Contrôle	16x16	30541		30541	26131	43647		43647
Séq. + UT		20528	0	20528	6944	24397	6942	31339
Séq. + Umem		17456	3072	20528	3872	21313	6954	28267

FIG. 5.20 – Résultats après réaffectation pour la méthode "Recherche en croix".

de 49% ici. Cette réduction du nombre de transferts s'accompagne d'une diminution non négligeable des mémorisations réalisées dans l'UT. Cette réduction du nombre de mémorisations dans l'UT s'explique par la disparition de toute ou partie des calculs d'adresses (mémorisation des opérandes et des résultats) ; le gain ainsi obtenu est de 13%.

Les gains relevés après l'étape d'optimisation présynthèse permettent d'affirmer que l'approche proposée permet de réduire le nombre de calculs et transferts d'adresses par rapport à une solution de type contrôle. De plus, le transfert des calculs d'adresses dynamiques dans le séquenceur permet une réduction du nombre de mémorisations et de transferts de données. Cette réduction influera dans les prochaines étapes sur : la consommation de l'architecture, le nombre de ressources matérielles à mettre en oeuvre (lorsque les contraintes temporelles sont importantes) et la taille/complexité du contrôleur nécessaire pour piloter l'UT. La contrepartie de ces gains est la génération d'un chemin de données et de la glue logique adéquate dans le séquenceur de l'UMem (sans préjuger à ce stade d'un gain global ou non après synthèse de l'UT et de l'UMem).

Les figures 5.19 et 5.20 présentent les résultats obtenus post-optimisation sur les 2 autres algorithmes d'estimation de mouvements.

Les résultats obtenus dans le cas de la recherche orthogonale et de la recherche en croix confirment les points que nous avons fait ressortir de notre premier exemple : baisse du nombre de transferts et de calculs comparé à une architecture de type contrôle, et baisse du nombre de transferts et de mémorisations comparé à une architecture de type séquenceur avec implémentation des calculs d'adresses dans l'UT.

Nous allons maintenant passer à la deuxième étape de notre flot : la synthèse d'architecture à propre-

			Matériel UT					Matériel Umem	
Technique	Norme	Contrainte	Sous.	Add.	Abs.	EQMux	Reg.	Add.	Reg.
UT	CIF (768x576)	14us	2	3	2	1	95	0	2
UT + Umem			2	2	2	1	82	1	9

FIG. 5.21 – Résultats de synthèse de l'algorithme de "Recherche en 3 étapes".

Composant Matériel	Registre					Additionneur					Soustracteur			Absolue			Mux		
Dynamique (bits)	6	8	14	16	20	6	8	14	16	20	14	16	20	14	16	20	14	16	20
Surface (slices)	3	5	8	9	12	3	4	7	8	10	7	8	10	15	18	22	8	9	12
Latence (ns)	3,1	3,1	3,1	3,1	3,1	5,9	6	6,4	6,5	6,7	6,4	6,5	6,7	6,6	6,8	6,9	3,1	3,1	3,1

FIG. 5.22 – Caractéristiques des composants utilisés en fonction de leur dynamique.

ment parlé. Pour cela, nous allons utiliser l'outil GAUT pour générer l'architecture implémentant, à titre d'exemple, l'algorithme d'estimation de mouvements nommé " recherche en 3 étapes ". La contrainte temporelle est extraite du format HDTV (768×576 pixels à 25 images par secondes). *Au stade de développement actuel de notre flot de synthèse dans le coeur de l'outil, nous devons placer l'ensemble des données, appartenant à une même structure, au sein d'un même banc mémoire. Cette restriction limite le parallélisme exploitable par notre méthodologie. La taille des macroblocs considéré vaut* 8×8 *pixels.*.

Le Tableau 5.21 résume les résultats de synthèse. Nous pouvons observer les quantités de matériel allouées par chacune des 2 solutions à base de séquenceur. La solution de type contrôle n'a pas été synthétisée par GAUT car cet outil est dédié à la génération d'architectures basées sur l'utilisation de séquenceurs et n'aurait donc pas généré des résultats pertinents. Afin d'estimer les gains en surface apportés par notre approche, nous avons réalisé la synthèse logique des composants nécessaires aux architectures générées. Les caractéristiques en taille/surface de ces composants sont précisées dans la figure 5.22.

Pour comparer la pertinence d'une solution par rapport à l'autre, une analyse des architectures globales (UT + UMem) est nécessaire. Le Tableau 5.23 compare la surface nécessaire à l'implémentation des chemins de données et la complexité des machines d'état à mettre en oeuvre dans des architectures obtenues après synthèse. Nous pouvons remarquer un gain en surface de 11%, gain obtenu grâce à la réaffectation des calculs d'adresses au sein du séquenceur d'accès à la mémoire.

5.3.5 Conclusion

La méthode basée sur l'utilisation d'un séquenceur mémoire dans les applications contenant des accès indéterministes permet de réduire le nombre de calculs d'adresses à réaliser ainsi que le nombre de transferts d'adresses (adressages dynamiques) à effectuer. De plus, la réaffectation intelligente des

	Matériel UT		Matériel Umem		Surface globale	Gains
Technique	Surface	Controleur (bits/instruc)	Surface	Controleur (bits/instruc)		
UT	940	95	6	2	946	
UT + Umem	815	82	27	8	842	10.99%

FIG. 5.23 – Surface des architectures synthétisées.

calculs d'adresses dynamiques au sein de l'unité de mémorisation permet d'aboutir à une architecture où le nombre de mémorisations / transferts de données est encore diminué. La réaffectation des calculs d'adresses permettra par la suite, lors de synthèses contraintes par un nombre de bus, d'éviter d'encombrer ces derniers avec des adresses. Comme nous l'avons vu dans l'exemple pédagogique et dans l'estimation de mouvement, malgré l'utilisation d'un flot/outil de synthèse inadapté a priori à ce genre d'applications, notre approche permet une gestion automatique de l'indéterminisme dans les accès à la mémoire à l'aide d'une approche basée sur les séquenceurs d'accès à la mémoire. Elle autorise de plus un gain non négligeable en terme réduction des commutations (nombre de transferts inter-unités et mémorisations de données) et de la surface nécessaire (réduction de la quantité de registres, de la dynamiques des composants dédiés aux calculs d'adresses). Par ailleurs, la réaffectation des calculs d'adresses pourrait présenter d'autres avantages comme la réduction de la latence dans le cadre d'applications où les communications inter-unités nécessitent un nombre de cycles supérieur à 1 (bus faible consommation, bus pipeline, ...).

5.4 Chaîne de traitement DVB-DSNG (Projet ALIPTA)

5.4.1 Présentation

Le projet RNRT ALIPTA (2002-2004) visait à proposer, développer et expertiser de nouvelles méthodes pour la conception de systèmes sur puce pour les applications en télécommunication. Ces méthodes sont caractérisées par la spécification et l'intégration de composants virtuels de niveau algorithmique, génériques, flexibles et paramétrables que nous avons présentées dans le chapitre 1. L'approche proposée dans le cadre d'ALIPTA repose donc sur l'utilisation des technologies de synthèse d'architecture et de communication pour réaliser le lien vers l'implémentation matérielle. Les partenaires impliqués dans le projet étaient : ENST-Bretagne, TURBOCONCEPT, SACET, et THALES.

L'application considérée est un MODEM basé sur la norme *DVB-DSNG (Digital Video Broadcasting Digital Satellite News gatherings)* [DVB]. Cette norme intervient dans le cadre des transmissions numériques par satellite. Elle permet des transmissions point à point ou multipoints. Par exemple dans le domaine de la télévision, elle fournit, pour les acquisitions en temps réel d'événements, une réponse appropriée à un moindre coût pour établir des liaisons rapides entre des véhicules satellitaires et des studios. Le standard DVB-DSNG cible des taux de transfert compris entre $1.5Mb/s$ et $72Mb/s$. Le schéma présenté dans la figure 5.24 décrit de manière synthétique l'ensemble des traitements effectués lors à la réception.

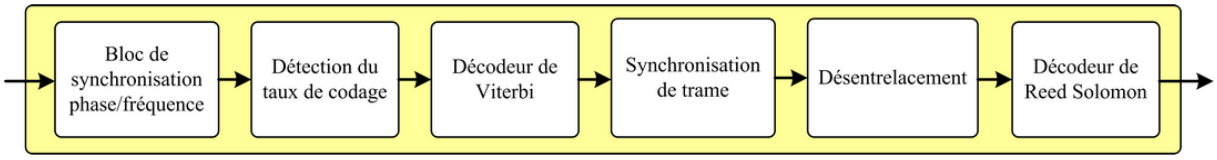

FIG. 5.24 – Schéma de principe d'un récepteur compatible avec le standard DVB-DSNG.

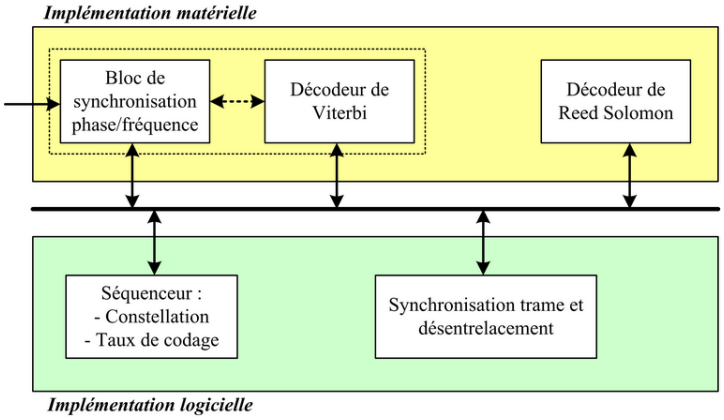

FIG. 5.25 – Architecture d'implémentation du récepteur DVB-DSNG.

Le partitionnement HW/SW de l'architecture a été défini en fonction du niveau de complexité d'implémentation et des performances cibles de la chaîne de traitement. La chaîne de réception ALIPTA est ainsi constituée de trois composants matériels (un décodeur de *Viterbi* $(7,1/2)$, un décodeur de *Reed Solomon* $(204,188)$ et un bloc de synchronisation phase/fréquence) et de deux composants logiciels (un bloc de séquençage (détermination de la constellation et du taux de codage) et une partie synchronisation/désentrelacement de trames) conformément à la figure 5.25. Les composants logiciels implémentent des algorithmes reposant essentiellement sur des manipulations de données en mémoire, les composants matériels implémentent quant à eux les traitements de type calculs intensifs.

La plateforme d'implémentation choisie est la plateforme *Excalibur* de chez *Altera*. Les parties logicielles sont implémentées sur un processeur $ARM922T$, les parties matérielles sur FPGA de type $EPXA10$. Un bus AHB permet aux différents composants de communiquer.

Le décodeur de Viterbi - L'algorithme de Viterbi peut être divisé en 3 parties principales :

1. le calcul des métriques de branches (MB) (qui représentent la probabilité de la transition de l'état s_i vers l'état s_j du codeur),
2. l'accumulation des métriques de branche pour chacun des états du codeur et la sélection du chemin survivant (étape généralement nommée Addition - Comparaison - Sélection (ACS)),
3. mémorisation des chemins survivants et remontée du treillis afin de choisir le symbole qui doit être décidé *(Survivor Memory Evaluation : SME)*.

Les performances des décodeurs de Viterbi sont généralement limitées par la partie ACS dont le résultat doit être connu avant de pouvoir débuter le calcul du suivant. Cela signifie qu'a chaque exécution d'un décodeur de Viterbi à N états, N calculs ACS doivent être réalisés. Deux techniques classiques permettent de gérer la mémorisation et le décodage du chemin survivant : la méthode dite du *"Trace-Back Algorithm (TBA)"* et l'algorithme d'échange des registres (REA) [Feyg93]. Afin de gérer les débits potentiellement élevés de la norme DVB-DSNG, la méthode REA est préférable. Cette méthode nécessite un registre à décalage qui contient le chemin survivant associé à un état du codeur. Les registres sont interconnectés sous la forme d'un treillis et leur mise à jour est effectuée à l'aide d'un échange de leur contenu. Etant donné que la techniques REA est implicitement de niveau transfert de registre (RTL), nous avons décidé de réaliser la description et la synthèse comportementale des parties MB et ACS du décodeur de Viterbi alors que la partie survivant a été codée directement au niveau RTL. Les résultats de synthèse obtenus sous différentes contraintes de débits sont présentés dans la sous section *"Résultats"*.

Le décodeur de Reed Solomon - Une estimation de la complexité du décodeur de Reed Solomon en terme d'opérations logiques a été réalisée avant de réaliser sa spécification comportementale. L'intérêt de cette analyse est double : obtention d'une liste des opérations nécessaires à l'implémentation de l'algorithme et calcul de sa complexité fonctionnelle en fonction des paramètres du décodeur (nombre de symboles reçus et nombre de données utiles). A partir de ces informations, il est possible de déterminer précisément la complexité des différentes parties qui composent le décodeur. Nous avons ainsi mis en évidence le besoin de développer des opérateurs adaptés aux corps de Galois (addition, soustraction et inversion). En effet, pour le $RS(204, 188)$, la plupart des opérations s'effectuent dans un corps de *Galois* $CG(256)$. Ce corps nécessite des conversions à l'aide de tables implémentées par des mémoires au niveau matériel. L'utilisation d'opérateurs matériels permettant de réaliser des calculs directement dans le corps de Galois a permis de diminuer les accès mémoires. Dans un second temps, nous avons pu remarquer qu'utiliser un opérateur de type multiplication-accumulation (MAC), permettait de réduire la complexité opératoire de 30%. L'ensemble des opérateurs opérant dans le corps de Galois et l'opérateur MAC ont ainsi été écrits en VHDL-RTL et synthétisés par les outils de synthèse d'Altera (Quartus). Les caractéristiques (surface, latence et consommation) extraites de la synthèse logique de ces nouveaux opérateurs "élémentaires" ont alors été ajoutées dans la bibliothèque caractérisant l'ensemble des opérateurs disponibles pour la synthèse de haut niveau au même titre que les autres opérateurs (abs, shl, shr, ...).

5.4.2 Résultats

Les descriptions comportementales génériques utilisées dans le projet ALIPTA nous permettent de faire varier différents paramètres applicatifs/fonctionnels tel que par exemple le nombre d'états du décodeur et/ou son débit. Les résultats obtenus lors de la synthèse architecturale des composants virtuels des décodeurs de Viterbi et de Reed Solomon implémentés sur FPGA à l'aide de l'outil GAUT sont présentés dans la figure 5.26. Chaque décodeur a été synthétisé pour différents débits répondant à la norme DVB-DSNG. Il faut noter que l'ensemble des synthèses architecturales a été effectué à partir d'une seule et même description algorithmique (une par type de décodeur). La synthèse du bloc de filtrage et de synchronisation phase fréquence a quant à elle été réalisée par Thalès et est confidentielle.

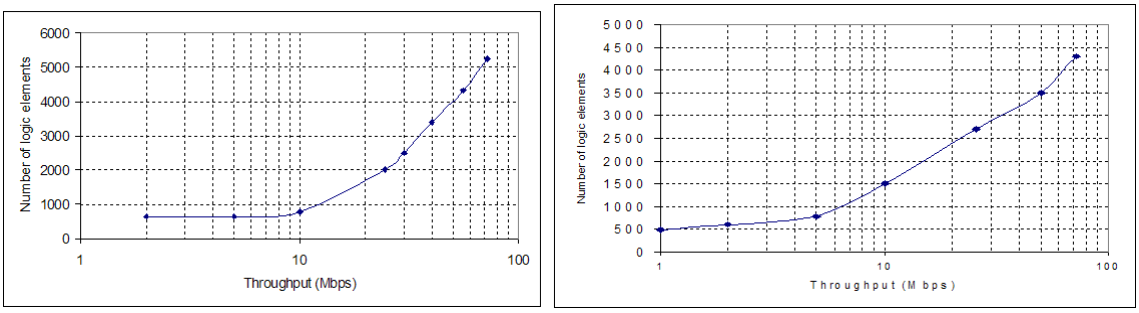

FIG. 5.26 – Résultats de synthèse (a) décodeur de Reed Solomon (b) décodeur de Viterbi.

Au final, la solution architecturale implémentant l'application complète a été validée par simulation pour un débit de *26Mbits/s*. Le bloc de synchronisation et désentrelacement implémenté en logiciel sur *ARM*922*T* limite en effet en pratique le débit de notre application. Toutefois, les résultats obtenus sont extrêmement intéressants dans la mesure où Thales a réussi à synthétiser avec l'outil GAUT un bloc de synchronisation phase/fréquence fonctionnant à *70 MHz*, à comparer à leur modèle RTL conçu à la main fonctionnant à *40 MHz*.

La conception du récepteur *DVB-DSNG* effectuée dans le cadre du projet ALIPTA a ainsi permis de valider l'intérêt d'un outil de synthèse de haut niveau comme GAUT pour la conception d'applications industrielles.

5.5 Conclusion

L'ensemble des expériences que nous avons présentées dans ce chapitre a permis de mettre en avant l'intérêt de notre modèle de représentation ainsi que l'approche de conception associée. Ainsi, nous avons montré que l'utilisation d'un flot de synthèse dédié aux applications déterministes peut être modifié afin de permettre la synthèse sous contraintes d'applications contenant de l'indéterminisme (contrôle, adressage, ...). Les analyses des différentes applications en TDSI traitées permettent de montrer l'efficacité de notre approche par rapport à une conception utilisant des outils de synthèse ciblant des architectures de type contrôle et illustrent l'aspect novateur de notre gestion des calculs d'adresses indéterministes à l'aide de séquenceurs (avec ou sans chemin de données). Il faut cependant garder à l'esprit que notre approche est volontairement adaptée aux applications majoritairement dominées par les calculs de type prédictif et non pas à celles où l'indéterminisme prévaut. Enfin, nous avons validé la capacité de notre outil à traiter des problèmes industriels de grande complexité même si, dans ce dernier cas, notre méthode d'optimisation n'a pas été considérée.

Chapitre 6

Bilan et Perspectives

Bilan

Nous avons proposé dans ce mémoire un nouveau modèle et son flot de synthèse associé pour la modélisation et la synthèse de *Composants Virtuels Comportementaux* pour les applications orientées traitement du signal, de l'image et télécommunications.

Le flot de conception proposé s'inscrit dans la démarche Adéquation Algorithme Architecture du projet *RNRT ALITPA (Définition et Application d'une méthodologie de développement pour les (IP) intellectual property de niveau comportemental dans les applications de télécommunication)* et est basé sur l'utilisation de techniques de synthèse de haut niveau sous contraintes d'intégration. L'outil de synthèse utilisé en pratique dans ces travaux est l'outil GAUT. Notre approche est basée sur un modèle de représentation original des composants virtuels de niveau algorithmique noté *CSFG (Control and Structure Flow Graph)* basé sur les *SDF (Signal Flow Graph)*. Le modèle permet la modélisation de sémantiques conditionnelles / indéterministes tout en conservant ses propriétés d'origine adaptées aux traitements intensifs de données. Les nouvelles techniques développées en adéquation avec les capacités du modèle permettent de lever certaines restrictions s'appliquant sur la sémantique d'entrée des applications. Le spectre des applications pouvant bénéficier des avantages de la synthèse de haut niveau sous contraintes d'intégration a ainsi pu être étendu.

D'un point de vue architectural, les unités fonctionnelles constituant l'architecture cible du composant ont été repensées en relation avec les nouvelles primitives algorithmiques à implémenter. L'ensemble des contrôleurs utilisés dans les différentes unités de l'architecture a ainsi été modifié.

Les transformations permettant de passer du modèle de représentation à son implémentation sur le modèle architectural ont été présentées. Notons toutefois que les modèles de contrôle et de hiérarchie présentés n'ont pas été intégrés à l'outil. La partie associée au traitement des séquences d'accès indéterministes à la mémoire dans une architecture à base de séquenceur a été plus amplement développée. Cette approche basée sur l'insertion d'un chemin de données dans le séquenceur d'accès à la mémoire permet de réduire certains paramètres (surface, consommation, latence) de l'architecture obtenue après synthèse dans le cadre d'application massivement déterministes. Le choix de l'implémentation des calculs

d'adresses dynamiques au sein de l'architecture est opéré de manière automatique à l'aide d'un outil qui décide si la réaffectation de l'opération est bénéfique (réduction du nombre de transferts global, consommation du circuit, surface, latence). Il est tout de même à noter que notre méthode cible les applications dominées par le traitement de données ; elle n'est donc pas adaptée à la synthèse d'applications dominées par le contrôle tel que par exemple le codage entropique, la recherche dans un dictionnaire, etc.

Nous avons présenté un ensemble de résultats obtenus en appliquant notre méthode à des algorithmes des domaines du Traitement du Signal et de l'Image (TDSI) et des Télécommunications. Trois expériences ont été menées. Dans la première expérience, présentée sous forme d'exemples pédagogiques, nous avons montré l'intérêt du nouveau formalisme de spécification afin de modéliser des structures conditionnelles/itératives (partage de ressources entre opérations mutuellement exclusives) et des structures hiérarchiques (réduction de la complexité du graphe afin de traiter des applications de complexité trop importante). Cette première série d'exemples pédagogiques permet de présenter les intérêts de notre approche.

La seconde expérience traite différentes techniques d'estimation de mouvement pour comparer les performances obtenues en affectant les calculs d'adresses au sein de l'UT ou de l'UMem. Dans l'exemple considéré, le gain sur le nombre de transferts de données entre les unités varie autour de 30% et en ce qui concerne la surface les gains sont dans l'exemple de 11%.

La dernière expérience présente la conception d'un décodeur *DVB-DSNG* qui a été réalisée dans le cadre du projet *ALIPTA*. Cette réalisation démontre qu'il est actuellement possible de concevoir des systèmes complexes à l'aide de méthodologies de synthèse de haut niveau.

Perspectives

Bien que l'approche proposée conduise à des résultats de bonne qualité sur les exemples traités, plusieurs améliorations et extensions peuvent être apportées comme nous l'avons mentionné à plusieurs reprises dans ce document.

Ainsi, un premier ensemble de travaux à considérer est l'implémentation totale du nouveau modèle de représentation et des techniques de gestion des structures conditionnelles et hiérarchiques dans le coeur de synthèse de l'outil GAUT. L'implémentation de ces fonctionnalités sera réalisée dans le cadre de la thèse de C. Andriamisaina.

Un deuxième ensemble de travaux visant la synthèse sous contraintes de canaux de communication (nombre de bus) et de mémorisation (nombre de registres) doit être réalisé. Le premier point concerne la synthèse d'applications où le concepteur maîtrise le débit maximum entre l'UT et l'UMem en fonction des paramètres de l'application et/ou de l'architecture. Le second point concerne la maîtrise du nombre de registres alloués lors de la synthèse de l'architecture. Cette contrainte (nombre maximum de registres) permettrait de limiter la surface du circuit en répartissant linéairement le parallélisme sur la cadence spécifiée par le concepteur. La limitation de l'exploitation du parallélisme imposée durant l'étape d'ordonnancement par ces nouvelles contraintes permettrait une meilleure maîtrise du coût matériel du circuit.

Un dernier ensemble de travaux concerne la gestion de plusieurs scénario d'exécution au sein d'une même architecture. Nous pouvons par exemple imaginer une application embarquée nécessitant un filtrage de type filtre FIR 128 points lorsque la qualité du signal reçu est mauvaise et un filtre d'une longueur moindre (par exemple 32 points) lorsque la réception est de bonne qualité. Cela permettrait entre autre de réduire la consommation du système de filtrage. Actuellement, avec l'outil GAUT, 2 architectures différentes seraient synthétisées. Il est néanmoins possible d'utiliser les techniques de partages d'opérateurs entre *scénarios mutuellement exclusifs* afin d'exploiter l'architecture du filtre à 128 points pour implémenter celui à 32 points. Les travaux de thèse de Cyrille Chavet où l'unité de communication (UCom) doit pouvoir supporter différentes configurations devraient recouper cette perspective.

Ces différentes perspectives s'intègrent dans le flot proprement dit de synthèse de haut niveau. A long terme, l'évolution des applications couplée à l'évolution de la technologie permettra d'appréhender des systèmes encore plus complexes. L'évolution rapide de la technologie et de ses utilisations (reconfiguration dynamique d'accélérateurs matériels autour de réseaux sur puce par exemple) permet de penser que les concepts de *Composants Virtuels* et de *Reprogrammation dynamique* puissent prochainement fusionner afin d'aboutir à l'apparition de composants virtuels multifonctions (chaque fonction proposée nécessitant une configuration particulière de l'architecture). Dans ce cas, les composants virtuels algorithmiques pourraient être composés d'un ensemble de fonctions exécutables à implémenter sur une même architecture cible (un composant virtuel algorithmique de décodage audio pourrait contenir la description des décodeurs de formats ogg, wma, aac, etc.). A nouveau, l'usage des techniques de partage des opérateurs entre, cette fois-ci, fonctions mutuellement exclusives serait particulièrement intéressant.

Bibliographie

Bibliographie Personnelle

Synthèse d'applications

[Abde05] N. Abdelli, P. Bomel, E. Casseau, AM Fouilliart, C. Jego, P. Kajfasz, **B. Le Gal**, N. Le Heno, and E. Martin. Hardware design based on virtual component systhesis. In *the Proceedings of the Euromicro Conference on Digital System Design (DSD)*, Porto, Portugal, 30 August - 3 September 2005.

[Cass04] E. Casseau, **B. Le Gal**, C. Jégo, N. Le Héno, and E. Martin. Reed-solomon behavioral virtual component for communication systems. In *the Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, Vancouver, Canada, 21-23 May 2004.

[LeGa04] **B. Le Gal**, E. Casseau, P. Bomel, C. Jégo, N. Le Héno, and E. Martin. High-level synthesis assisted rapid prototyping for digital signal processing. In *the Proceedings of the IEEE International Conference on Microelectronics (ICM)*, Tunis, Tunisia, 6-8 December 2004.

Flot de conception de haut niveau

[Cass05a] E. Casseau, **B. Le Gal**, P. Bomel, C. Jégo, S. Huet, and E. Martin. C-based rapid prototyping for digital signal processing. In *the Proceedings of the European Signal Processing Conference (EUSIPCO)*, Antalya, Turquie, 4-8 September 2005.

[Huet04] S. Huet, P. Bomel, E. Casseau, **B. Le Gal**, and O. Pasquier. Electronic system level to HW/SW design flow. In *the Proceedings of the Global Signal Processing Conference (GSPx)*, Santa Clara, CA USA, 27-30 Septembre 2004.

Modélisations / Transformations

[LeGa05a] **B. Le Gal**, E. Casseau, and E. Martin. Bounded budgeted parallel architecture versus control dominated architecture for hazard data-signal processors synthesis. In *the Proceedings of the SPIE Conference, Microtechnologies for the New Millennium*, volume 5837, pages 100–111, Sevilla, Espagne, 9-11 May 2005.

[LeGa05b] **B. Le Gal**, E. Casseau, S. Huet, and E. Martin. Pipelined memory controllers for DSP applications handling unpredictable data accesses. In *the Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 268–269, Tampa, Floride, USA, 11-12 May 2005.

[LeGa05c] **B. Le Gal**, E. Casseau, and E. Martin. Pipelined memory controllers for DSP real time applications handling unpredictable data acesses. In *the Proceedings of the European Signal Processing Conference (EUSIPCO)*, page 4p., Antalya, Turquie, 4-8 September 2005.

[LeGa05d] **B. Le Gal**, E. Casseau, C. Andriamisaina, S. Huet, and E. Martin. Séquenceur mémoire pour applications multimédia temps réel gérant les séquences d'accès indéterministes. In *the Proceedings of the GRETSI Conference*, Louvain-la-Neuve, Belgique, 6-9 Septembre 2005.

[LeGa05e] **B. Le Gal**, E. Casseau, C. Andriamisaina, and E. Martin. Dynamic memory acess managment and address computation for dataflow applications. In *the Proceedings IFIP International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 152–157, Perth, Australie, 17-19 October 2005.

Bibliographie Relative au Manuscrit

[Agha03] S. Agha and V.M. Dwyer. Algorithms and vlsi architectures for mpeg-4 motion estimation. In *1st ESC Division mini-conference*, 25th September 2003.

[Aho86] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques, and Tools*. Number 0201100886. Addison Wesley, 1986.

[Bacc91] F. Baccelli, G. Cohen, and B. Gaujal. Recursive equations and basic properties of timed petri nets. Technical report, INRIA - Sophia Antipolis , Equipe : MISTRAL, May 1991.

[Baco94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4) :345–420, 1994.

[Bala97] F. Balarin and al. *Hardware-Software Codesign of Embedded Systems – The POLIS Approach.* Kluwer Academic Publishers,, 1997.

[Beni01] L. Benini, L. Macchiarulo, A. Macii, E. Macii, and M. Poncino. From architecture to layout : partitioned memory synthesis for embedded systems-on-chip. In *DAC '01 : Proceedings of the 38th conference on Design automation*, pages 784–789, New York, NY, USA, 2001. ACM Press.

[Beni02] L. Benini and G. De Micheli. Networks on chips : A new soc paradigm. *Computer*, 35(1) :70–78, 2002.

[Berg91] R. Bergamaschi. The effect of false paths in high level synthesis. In *the Proceedings of the ICCAD'91*, pages 80–83, Santa Clara, November 1991.

[Bhat94] S. Bhattacharya, S. Dey, and F. Brglez. Performance analysis and optimization of schedules for conditional and loop-intensive specifications. In *the Proceedings of the 31st annual conference on Design automation (DAC'94)*, pages 491–496, New York, NY, USA, 1994. ACM Press.

[Blah02] R. E. Blahut. *Algebraic Codes for Data Transmission*. Number ISBN : 0521553741. Cambridge University Press, July 2002.

[Bome04] P. Bomel. *Plate-forme de prototypage rapide fondée sur la synthèse de haut niveau pour applications de radiocommunications numériques*. PhD thesis, Université de Bretagne Sud, Décembre 2004.

[Bond99] K. Bondalapati, P. C. Diniz, P. Duncan, J. Granacki, M. W. Hall, R. Jain, and H. Ziegler. Defacto : A design environment for adaptive computing technology. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 570–578, London, UK, 1999. Springer-Verlag.

[Boss04] L. Bossuet. *Exploration de l'espace de conception des architectures reconfigurables*. PhD thesis, Université de Bretagne Sud - LESTER, 2004.

[Camp89] R. Camposano and W. Rosentiel. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(2) :171–180, February 1989.

[Camp91] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1) :85–93, January 1991.

[Card04] J. M. P. Cardoso and P. C. Diniz. Modeling loop unrolling : Approaches and open issues. In *the Proceedings of the International Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS IV)*, pages 224–233, Greece, July 2004.

[Cart99] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse : Building a smarter memory controller. In *the Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 70–79, January 1999.

[Cass02] E. Casseau. Soc design using behavioral level virtual components. In *the Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, volume 2, pages 497–500, Dubrovnik, Croatie, 15-18 September 2002.

[Cass05] E. Casseau, B. Le Gal, P. Bomel, C. Jégo, S. Huet, and E. Martin. C-based rapid prototyping for digital signal processing. In *the Proceedings of the European Signal Processing Conference (EUSIPCO 2005)*, Antalya, Turquie, 4-8 Septembre 2005.

[Cata04a] Mentor Graphics Corporation. *Catapult C Synthesis C++ to Hardware Concepts*, release 2004b edition, June 2004.

[Cata04b] Mentor Graphics Corporation. *Catapult C Synthesis User's and Reference Manual*, release 2004b edition, June 2004.

[Cata04c] Mentor Graphics Corporation. *Catapult C Synthesis Style Guide*, release 2004b edition, June 2004.

[Cava03] P. Cavalloro, C. Gendarme, K. Kronlof, J. Mermet, J.V. Sas, K. Tiensyrja, and N. Voros. *System Level Design Model with Reuse of System IP*. Springer, 1 edition, September 2003.

[Cesa02] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 789–794, New York, NY, USA, 2002. ACM Press.

[Chai92a] V. Chaiyakul and D.D. Gajski. Assignment decision diagram and its uses in high-level synthesis. Technical report, University of California Irvine, October 1992.

[Chai92b] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. Minimizing syntactic variance with assignment decision diagrams. Technical Report TR-92-34, UC Irvine, April 1992.

[Chai92c] Viraphol Chaiyakul and Daniel D. Gajski. Assignement decision diagram for high-level synthesis. Technical Report ICS-TR-92-103, University of California, Irvine, 1992.

[Chai93] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *DAC '93 : Proceedings of the 30th international conference on Design automation*, pages 413–418, New York, NY, USA, 1993. ACM Press.

[Chao93] L.-F. Chao and A. LaPaugh. Rotation scheduling : a loop pipelining algorithm. In *the Proceedings of the 30th international conference on Design automation (DAC'93)*, pages 566–572, New York, NY, USA, 1993. ACM Press.

[Chil97] D. Chillet. *Méthodologie de conception architecturale des mémoires pour circuits dédiés au traitement du signal temps réel*. PhD thesis, ENSSAT-Université de Rennes, Janvier 1997.

[Choa94] L.-F. Chao. Optimizing cyclic data-flow graphs via associativity. In *the Proceedings of the Fourth Great Lakes Symposium on Design Automation of High Performance VLSI Systems (GLSV '94)*, pages 6–10, 1994.

[Chu89] C. M. Chu, M. Potkonjak, M. Thaler, and Rabaey. Hyper : An interactive synthesis environment for real time applications. In *the Proceeding of the International Conference on Computer Design*, pages 432–435, 1989.

[Ciar01] M. Ciaran and W. Xiaojun. Most often used path scheduling algorithm. In *the Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2001)*, volume 12, pages 289–295, Orlando, Florida, USA, July 22-25USA 2001.

[Corr03b] G. Corre, N. Julien, E. Senn, and E. Martin. Ordonnancement sous contraintes de mémorisation : une optimisation efficace des ressources lors de la synthèse d'architecture. In *Actes des Journées Francophones d'études Faible Tension Faible Consommation (FTFC)*, 2003.

[Corr05] G. Corre. *Gestion des unités de mémorisation pour la synthèse d'architecture*. PhD thesis, Université de Bretagne Sud, Juin 2005.

[Corr05b] G. Corre, N. Julien, E. Senn, and E. Martin. Réduction de l'influence du placement mémoire par la synthèse de haut niveau. In *the Proceedings of FTFC (journées d'études Faible Tension Faible Consommation)*, Mai 2005.

[Cous03] P. Coussy. *Synthèse d'Interface de Communication pour les Composants Virtuels*. PhD thesis, Université de Bretagne Sud, Decembre 2003.

[Cous05b] P. Coussy, E. Casseau, P. Bomel, A. Baganne, and E. Martin. A formal method for hardware ip design and integration under i/o and timing constraints. *To appear in ACM Transaction on Embedded Computing Systems*, 2005.

[Dasg95] A. Dasgupta and R. Karri. Simultaneous scheduling and binding for power minimization during microarchitecture synthesis. In *the Proceedings of the 1995 international symposium on Low power design (ISLPED '95)*, pages 69–74, New York, NY, USA, 1995. ACM Press.

[Dasg98] A. Dasgupta and R. Karri. High-reliability, low-energy microarchitecture synthesis. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 17(12) :1273–1280, 1998.

[Davi04] R. David, D. Lavenier, and S. Pillement. Du microprocesseur au circuit fpga : une analyse sous l'angle de la reconfiguration. *Technique et Science Informatiques*, 2004.

[DeMi02] G. De Micheli, R. Ernst, and W. Wolf, editors. *Readings In Hardware/Software Co-Designreadings In Hardware/Software Co-Design*. Number ISBN : 1-55860-702-1 in The Morgan Kaufmann Series in Systems on Silicon. Elsevier, 2002.

[DeMi88] G. De Micheli and D.C. Ku. Hercules - a system for high-level synthesis. In *DAC '88 : Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 483–488, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Deva87] S. Devadas and R.A. Newton. Algorithms for hardware allocation in datapath synthesis. *IEEE Transactions on CAD*, 8(7) :768–781, July 1987.

[Dini00] Pedro Diniz and Joonseok Park. Automatic synthesis of data storage and control structures for fpga-based computing engines. In *FCCM '00 : Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 91. IEEE Computer Society, 2000.

[Dini02] P. C. Diniz and J. Park. Data reorganization engines for the next generation of system-on-a-chip fpgas. In *FPGA '02 : Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 237–244, New York, NY, USA, 2002. ACM Press.

[DVB] *Digital Video Broadcasting (DVB) : Framing structure, channel coding and modulation for Digital Satellite News Gathering (DSNG) and other contribution applications by satellite, EN 301 210.*

[Elli00] J. P. Elliott. *Understanding High-Level Synthesis. A Practical Guide to High-Level Design*. Kluwer Academic Publishers, 2000.

[Fey04] F. Fey, R. Drechsler, and M. Ciesielski. Algorithms for taylor expansion diagrams. In *the Proceedings of the 34th International Symposium on Multiple-Valued Logic (ISMVL'04)*, pages 235–240, 2004.

[Feyg93] G. Feygin and P.G. Gulak. Architectural tradeoffs for survivor sequence memory management in viterbi decoders. *IEEE Transaction on Communications*, 41(3) :425–429, March 1993.

[Furh96] B. Furht, J. Greenberg, and R. Westwater. *Motion Estimation Algorithms for Video Compression*. Number ISBN : 0792397932. Springer, November 30 1996.

[Gail98] S. Gailhard, N. Julien, J.P. Diguet, and E. Martin. Methods to transform easily classical architectural synthesis tools to low power ones. In *the Proceedings of the 8th IEEE-Great Lakes Symposium on VLSI*, Louisiana, USA, Februrary 19-21 1998.

[Gajs92] D. Gajski et al. *High-Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[Gajs96] A. Orailoglu and D.D. Gajski. Flow graph representation. In *the Proceedings of the 23rd ACM/IEEE conference on Design automation (DAC'86)*, pages 503–509, Piscataway, NJ, USA, 1986. IEEE Press.

[Ghar90] H. Gharavi. The cross search algorithm for motion estimation. *IEEE Transactions on Communications*, 38(7) :950–953, July 1990.

[Gnae03] D. Gnaedig, M. Jezequel, and E. Boutillon. Les turbo codes à roulettes. In *Dans les actes du 19ème colloque sur le traitement du signal et des images (GRETSI)*, volume 2, pages 211–214, Paris, 8-11 Septembre 2003.

[Goss89] G. Goossens, J. Vandewlle, and H. De Man. Loop optimization in register-transfer scheduling for dsp-systems. In *DAC '89 : Proceedings of the 26th ACM/IEEE conference on Design automation*, pages 826–831, New York, NY, USA, 1989. ACM Press.

[Gran00] Thierry Grandpierre. *Modélisation d'architectures parallèle hétérogènes pour la génération automatique d'executifs distribués temps réel optimisés*. PhD thesis, Université de Paris Sud, UFR Scientifique d'Orsay, Novembre 30 2000.

[Guil03] A.-C. Guillou. *Synthèse architecturale basée sur le modèle polyédrique : validation et extensions de la méthodologie MMAlpha*. PhD thesis, Université de Rennes, December 2003.

[Gupt00] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *the Proceedings of the conference on Design Automation and Test in Europe (DATE '00)*, pages 9–13, New York, NY, USA, 2000. ACM Press.

[Gupt01] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Conditional speculation and its effects on performance and area for high-level snthesis. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 171–176. ACM Press, 2001.

[Gupt01a] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *the Proceedings of the 38th Conference on Design Automation (DAC'01)*, pages 269–272, New York, NY, USA, 2001. ACM Press.

[Gupt02] S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *ISSS '02 : Proceedings of the 15th international symposium on System Synthesis*, pages 261–266, New York, NY, USA, 2002. ACM Press.

[Gupt02a] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, A. Nicolau, T. Kam, M. Kishinevsky, and S. Rotem. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 898–903, New York, NY, USA, 2002. ACM Press.

[Gupt03] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. In *the Proceedings of the IEEE Conference on Computers and Digital Technique*, volume 150, pages 330–337. IEEE, USA, September 2003.

[Gupt03a] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, January 2003.

[Gupt03b] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *the Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, pages 10270–10275, Washington, DC, USA, 2003. IEEE Computer Society.

[Gupt03c] S. Gupta, M. Luthra, N.D. Dutt, R.K. Gupta, and A. Nicolau. Hardware and interface synthesis of fpga blocks using parallelizing code transformations. Invited talk at the special session on Synthesis For Programmable Systems at the International Conference on Parallel and Distributed Computing and Systems, November 2003.

[Gupt04] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK : : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Number 1402078374. Springer, 1 edition, 2004.

[Hall96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12) :84–89, 1996.

[Harv01] D. N. Harvala. *A Low Power Application-Specific Integrated Circuit (ASIC) implementation of Wavelet Transform/Inverse Transform*. PhD thesis, Air Force Inst Of Tech Wright-Pattersonafb Oh School Of Engineering, March 2001.

[Hett02a] S. Hettiaratchi, P. Cheung, and T. Clarke. Performance-area trade-off of address generators for address decoder-decoupled memory. In *DATE '02 : Proceedings of the conference on Design, automation and test in Europe*, page 902. IEEE Computer Society, 2002.

[Holm94] N.D. Holmes and D. Gajski. An algorithm for generation of behavioral shape functions. In *the Proceddings of the EDAC-ETC-EUROASIC Conference*, pages 314–318, 1994.

[Holm95] N. D. Holmes and D. D. Gajski. Architectural exploration for datapaths with memory hierarchy. In *EDTC '95 : Proceedings of the 1995 European conference on Design and Test*, page 340. IEEE Computer Society, 1995.

[Holt93] U. Holtmann and R. Ernst. Experiments with low-level speculative computation based on multiple branch prediction. *IEEE Transactions on VLSI Systems*, 1 :262–267, September 1993.

[Holt95] U. Holtmann and R. Ernst. Combining mbp-speculative computation and loop pipelining in high-level synthesis. In *EDTC '95 : Proceedings of the 1995 European conference on Design and Test*, page 550, Washington, DC, USA, 1995. IEEE Computer Society.

[Hong00] S. Hong and T. Kim. Bus optimization for low-power data path synthesis based on network flow method. In *ICCAD '00 : Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 312–317, Piscataway, NJ, USA, 2000. IEEE Press.

[Huan01] Xianglong Huang, Zhenlin Wang, and Kathryn S. McKinley. Compiling for the impulse memory controller. In *PACT '01 : Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 141–150. IEEE Computer Society, 2001.

[Huan93] S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, and J. F. Wang. A tree-based scheduling algorithm for control-dominated circuits. In *Proceedings of the 30th international on Design automation conference*, pages 578–582. ACM Press, 1993.

[ITRS03] ITRS. International technology roadmap for semiconductors. Technical report, Technical Report, 2003.

[Jain81] J.R. Jain and A.K. Jain. Displacement measurement and its application in interframe image coding. *IEEE Transactions on Communications*, 29, December 1981.

[Jain91] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *DAC '91 : Proceedings of the 28th conference on ACM/IEEE design automation*, pages 686–689, New York, NY, USA, 1991. ACM Press.

[Jain92] R. Jain, A.C. Parker, and N. Park. Predicting system-level area and delay for pipelined and non-pipelined designs. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 12(8) :955–965, August 1992.

[Jamk02] S. Jamkar, S. Belhe, S. Dravid, and M.S. Sutaone. A comparison of block-matching search algorithms in motion estimation. In *ICCC '02 : Proceedings of the 15th international conference on Computer communication*, pages 730–739, Washington, DC, USA, 2002. International Council for Computer Communication.

[Jant03] A. Jantsch and H. Tenhunen, editors. *Networks on Chips*. Number ISBN :0306487276. Springer, February 2003.

[Jego00] C. Jego, E. Casseau, and E. Martin. Interconnect cost control during high-level synthesis. In *the Proceedings of the 15th Design of Circuits and Integrated Systems Conference (DCIS'00)*, pages 507–512, Montpellier, France, November 2000.

[Jego99] C. Jego, E. Casseau, and E. Martin. Architectural synthesis with interconnection cost control. In *the Proceedings of the IFIP International Conference on Very Large Scale Integration*, pages pp.509–520, Lisboa, Portugal, December 1999. in "VLSI : System on a chip", Kluwer Academic Publishers.

[Jerr96] A.A. Jerraya, P. Kission, and M. Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[Juan94] Hsiao ping Juan, Viraphol Chaiyakul, and Daniel D. Gajski. Condition graphs for high-quality behavioral synthesis. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 170–174. IEEE Computer Society Press, 1994.

[Kall02] P. Kalla, M. Ciesielski, E. Boutillon, and E. Martin. High-level design verification using taylor expansion diagrams : First results. In *the Proceedings of the Seventh Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02)*, Cannes, France, October 2002.

[Kaou03] L. Kaouane, M. Akil, T. Grandpierre, and Y. Sorel. A methodology to implement real-time applications on reconfigurable circuits. In *the Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 188–200, 2003.

[Kath02] V. Kathail, S.l Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, and M. Sivaraman. Pico : Automatically designing custom computers. *Computer*, 35(9) :39–47, 2002.

[Keat98] M. Keating and P. Bricaud. *Reuse methodology manual for system-on-a-chip designs*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Khou98] K.S. Khouri, G.Lakshminarayana, and N.K. Jha. Impact :a high-level synthesis system for low power control-flow intensive circuits. In *the Proceedings of the Internationnal Conference of Design Automation and Test in Europe (DATE '98)*, volume 0, page 848, 1998.

[Kita91] K. Kitagaki, T. Oto, T. Demura, Y. Araki, and T. Takada. A new address generation unit architecture for video signal processing. In *the Proceedings of SPIE International Conference on Visual Communications and Image Processing*, pages 891–900, Boston, MA, USA, November 1991.

[Knap96] D. W. Knapp. *Behavioral Synthesis. Digital System Design Using the Synopsis Behavioral Compiler*. Prentice Hall, 1996.

[Koeg96] M. Koegst, K. Feske, and G. Franke. State assignment for fsm low power design. In *EURO-DAC '96/EURO-VHDL '96 : Proceedings of the conference on European design automation*, pages 28–33, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[Koga81] T. Koga et al. Motion compensated interframe coding for video conferencing. In *the Proceedings of the Nationnal Telecommunication Conference*, pages G5.3.1–G5.3.5, New Orleans, LA, November 1981.

[Kott04] K. A. Kotteri. *Optimal, Multiplierless Implementations of the Discrete Wavelet Transform for Image Compression Applications*. PhD thesis, Faculty of the Virginia Polytechnic, April 27 2004.

[Koun01] A.A. Kountouris, C. Wolinski, and J.C. Le Lann. High-level synthesis using hierarchical conditional dependency graphs in the codesis system. *J. Syst. Archit.*, 47(3-4) :293–313, 2001.

[Koun02] A.A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3) :380–412, 2002.

[Koun98] Apostolos A. Kountouris and Christophe Wolinski. False path analysis based on hierarchical control representation. In *ISSS '98 : Proceedings of the 11th international symposium on System synthesis*, pages 55–59, Washington, DC, USA, 1998. IEEE Computer Society.

[Koun99] A.A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs for mutual exclusiveness identification. In *VLSID '99 : Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, page 146, Washington, DC, USA, 1999. IEEE Computer Society.

[Kuch01] K. Kuchcinski and C Wolinski. Synthesis of conditional behaviors using hierarchical conditional dependency graphs and constraint logic programming. In *DSD '01 : Proceedings of the Euromicro Symposium on Digital Systems Design*, page 220, Washington, DC, USA, 2001. IEEE Computer Society.

[Laks97] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. Wavesched : a novel scheduling technique for control-flow intensive behavioral descriptions. In *the Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design (ICCAD '97)*, pages 244–250, Washington, DC, USA, 1997. IEEE Computer Society.

[Laks98] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *DAC '98 : Proceedings of the 35th annual conference on Design automation*, pages 108–113, New York, NY, USA, 1998. ACM Press.

[Lam03] C. W. Lam, L. M. Po, and C. H. Cheung. A new cross-diamond search algorithm for fast block matching motion estimation. In *the Proceeding of 2003 IEEE International Conference on Neural Networks and Signal Processing*, pages 1262–1265, Nanjing, China, December 2003.

[Lee92] T.-F. Lee, A.C.-H. Wu, D.D. Gajski, and Y.-L. Lin. An effective methodology for functional pipelining. In *ICCAD '92 : Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 230–233, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[LeGu91] P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le Maire. Programming real time applications with signal. Technical Report 1446, Rapport de recherche de l'INRIA - Rennes , Equipe : EPATR, Juin 1991.

[Li1998] J. Li and R. K. Gupta. An algorithm to determine mutually exclusive operations in behavioral descriptions. In *Proceedings of the conference on Design, automation and test in Europe*, pages 457–465. IEEE Computer Society, 1998.

[Lipp91] P. E. R. Lippens, J. L. van Meerbergen, A. van der Werf, W. F. J. Verhaegh, B. T. McSweeney, J. O. Huisken, and O. P. McArdle. Phideo : a silicon compiler for high speed algorithms. In *the Proceedings of the conference on European Design Automation (EURO-DAC '91)*, pages 436–441, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Lyuh03] Chun-Gi Lyuh and Taewhan Kim. High-level synthesis for low power based on network flow method. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3) :364–375, 2003.

[Lyuh04] C.G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *the Proceedings of the 41st annual conference on Design automation (DAC '04)*, pages 81–86, New York, NY, USA, 2004. ACM Press.

[Mahl93] S.A. Mahlke, W.Y. Chen, R.A. Bringmann, R.E. Hank, W.-M.W. Hwu, B.R. Rau, and M.S. Schlansker. Sentinel scheduling : a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4) :376–408, 1993.

[Mar92] E. Martin and J.L. Philippe. Noyau de l'outil de synthèse gaut. Technical Report B, LASTI / ENSSAT, Novembre 1992.

[Mar93] JL. Philippe E. Martin, O. Santieys. Gaut, an architecture synthesis tool for dedicated signal processors. In *Proceedings IEEE International European Design Automation Conference (Euro DAC)*, 1993.

[Mentor] Mentor Graphics. http ://www.mentor.com/c-design.

[MILP00] Projet MILPAT. Rapport d'avancement 3.2 - spécification et expérimentation d'une application industrielle à base d'ips. Technical report, RNRT, 2000.

[Mont98] J.C. Monteiro and A.L. Oliveira. Finite state machine decomposition for low power. In *DAC '98 : Proceedings of the 35th annual conference on Design automation*, pages 758–763, New York, NY, USA, 1998. ACM Press.

[Moon02] Joong-Seok Moon, W.C. Athas, P.A. Beerel, and J.T. Draper. Low-power sequential access memory design. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, Orlando, USA, 2002.

[More02] R.H. Morelos-Zaragoza. *The Art of Error Correcting Coding*. Number ISBN : 0471495816. John Wiley and Sons, April 19 2002.

[Muso95] E. Musoll and J. Cortadella. High-level synthesis techniques for reducing the activity of functional units. In *ISLPED '95 : Proceedings of the 1995 international symposium on Low power design*, pages 99–104, New York, NY, USA, 1995. ACM Press.

[Nico85] Alexandru Nicolau. Percolation scheduling : A parallel compilation technique. Technical report, Ithaca, NY, USA, 1985.

[Nico91] A. Nicolau and R. Potasmann. Incremental tree height reduction for high level synthesis. In *DAC '91 : Proceedings of the 28th conference on ACM/IEEE design automation*, pages 770–774, New York, NY, USA, 1991. ACM Press.

[OCB00] VSI Alliance On chip bus development working group. *Virtuel Component Interface Standard (OCB 2 2.0)*, 2000.

[Pand98] P. R. Panda et al. Incorporating dram access modes into high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits And Systems*, 17(2) :96–109, February 1998.

[Park01] J. Park and P. C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines. In *ISSS '01 : Proceedings of the 14th international symposium on Systems synthesis*, pages 221–226. ACM Press, 2001.

[Park86] Alice C. Parker, Jorge T. Pizarro, and Mitch Mlinar. Maha : a program for datapath synthesis. In *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 461–466. IEEE Press, 1986.

[Park91] I.C. Park and C.M. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *DAC '91 : Proceedings of the 28th conference on ACM/IEEE design automation*, pages 680–685, New York, NY, USA, 1991. ACM Press.

[Park93] I. Park, K. O'Brien, and A.A. Jerraya. Amical : architectural synthesis based on vhdl. *IFIP - Transactions A Computer Science and Technology*, 22 :219, 1993.

[Paul89] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on Computer-Aided-Design*, 8(6) :661–679, June 1989.

[Pena00] O. Penalba, J. M. Mendias, and M. C. Molina. Execution condition analysis in high level synthesis : a unified approach. In *ISSS '00 : Proceedings of the 13th international symposium on System synthesis*, pages 73–78, Washington, DC, USA, 2000. IEEE Computer Society.

[Pena02] O. Penalba, J. M. Mendias, and R. Hermida. A global approach to improve conditional hardware reuse in high-level synthesis. *J. Syst. Archit.*, 47(12) :959–975, 2002.

[Pena02b] 0. Penalba, J.M. Mendias, and R. Hermida. Source code transformation to improve conditional hardware reuse. In *the Proceedings of the Euromicro Symposium on Digital System Design (DSD'02)*, page 324, 2002.

[Pete81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Po96] L. Po and W. Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Trans. on Circuits and Systems for Video Technology*, 6(3) :313–317, June 1996.

[Poly91] C.D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *ICS '91 : Proceedings of the 5th international conference on Supercomputing*, pages 252–263, New York, NY, USA, 1991. ACM Press.

[Pota90] R. Potasman, J. Lis, Al. Nicolau, and D.D. Gajski. Percolation based synthesis. In *DAC '90 : Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 444–449, New York, NY, USA, 1990. ACM Press.

[Potk94] M. Potkonjak and J. Rabaey. Optimizing resource utilization by transformations. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 13(3) :277–292, 1994.

[Puri87] A. Puri, H-M. Hang, and D.L. Schilling. An efficient block-matching algorithm for motion compensated coding. In *the Proceedings of the IEEE Internationnal. Conf. Acoust., Speech, Signal Processing*, pages 25.4.1–25.4.4, April 1987.

[Quin89] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques*. Masson, 1989.

[Raba88] J. Rabaey, H. D. Man, J. Vanhoof, G. Goossens, and F. Catthoor. Cathedral ii : A synthesis system for multiprocessor dsp. *in Silicon Compilation*, AddisonWesley :311–360, 1988.

[Rim92] M. Rim and R. Jain. Representing conditional branches for high-level synthesis applications. In *Proceedings of the 29th ACM/IEEE conference on Design automation conference*, pages 106–111. IEEE Computer Society Press, 1992.

[RNRT02] ET FRANCE TELECOM R&D LESTER, LASTI. Rnrt milpat. Site Internet du Projet MILPAT. http ://lester.univ-ubs.fr/milpat.

[Sava02] Guillaume Savaton. *Méthodologie de conception de composants virtuels comportementaux pour une chaîne de traitement du signal embarquée.* PhD thesis, Laboratoire d'Electronique des Systèmes Temps Réel, UNIVERSITE DE BRETAGNE SUD, Decembre 2002.

[Schr00] R. Schreiber, S. Aditya, B.R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. In *ASAP '00 : Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 113, Washington, DC, USA, 2000. IEEE Computer Society.

[Schr02] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B.R. Rau, D. Cronquist, and M. Sivaraman. Pico-npa : High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. Syst.*, 31(2) :127–142, 2002.

[SDR05] The software defined radio forum. Internet Web Site http ://www.sdrforum.org, 2005.

[Seep00] R. Seepold and N.M. Madrid. *Virtual Components Design and Reuse*. Springer, 1 edition, 2000.

[Seo02] J. Seo, T. Kim, and P.R. Panda. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 608–611. ACM Press, 2002.

[Seo03] Jaewon Seo, Taewhan Kim, and Preeti Ranjan Panda. Memory allocation and mapping in high-level synthesis : an integrated approach. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(5) :928–938, 2003.

[Shar93] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *in IEEE Transactions on Very Large Scale Integration*, 1(2) :175–190, June 1993.

[Son00] Sonics Inc. *Open Core Protocol Specification 1.0*.

[Stol92] A. Stoll and P. Duzy. High-level synthesis from vhdl with exact timing constraints. In *DAC '92 : Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 188–193, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Su85] B. Su and S. Ding. Some experiments in global microcode compaction. In *MICRO 18 : Proceedings of the 18th annual workshop on Microprogramming*, pages 175–180, New York, NY, USA, 1985. ACM Press.

[Timm93] A. H. Timmer, M. J. M. Heijligers, and J. A. G. Jess. Fast system-level area-delay curve prediction. In *the Proceedings of the 1st Asia Pacific Conference on Chip Design Language (APCHDL)*, pages 198–207, 1993.

[Vemu95] R. Vemuri and S. Govindarajan. Scheduling algorithms for high-level synthesis. Course on Digital Design Environments, March 1995. University of Cincinnati.

[VSIA03] Vsi alliance. site internet. http ://www.vsi.org.

[VSIA97] VSI ALLIANCE. *Architecture Document Version 1.0, Rapport technique*, 1997.

[Waka89] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *the Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD'89)*, pages 62–65. IEEE Computer Society Press, 1989.

[Waka92] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *the Proceedings of the 29th ACM/IEEE conference on Design automation (DAC '92)*, pages 112–115, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Wess97] C. Wess and M. Gotschlich. Constructing memory layouts for address generation units supporting offset 2 access. In *ICASSP '97 : Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97) -Volume 1*, page 683, Washington, DC, USA, 1997. IEEE Computer Society.

[Wick99] S.B. Wicker and V.K. Bhargava. *Reed-Solomon Codes and Their Applications*. Number ISBN : 0-7803-5391-9. Wiley-IEEE Press, 1999.

[Wild94] D.K. Wilde and O. Sie. Regular array synthesis using alpha. Technical report, I R I S A, May 1994.

[Wu05] Y. Wu and G. Megson. A novel two pass hexagonal search algorithm for motion estimation. In *the Proceeding Of Visualization, Imaging, And Image Processing*, 2005.

[Wuyt96] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, and H. de Man. Flow graph balancing for minimizing the required memory bandwidth. In *ISSS '96 : Proceedings of the 9th international symposium on System synthesis*, page 127, Washington, DC, USA, 1996. IEEE Computer Society.

[Wuyt99] Sven Wuytack, Francky Catthoor, Gjalt de Jong, and Hugo J. De Man. Minimizing the required memory bandwidth in vlsi system realizations. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(4) :433–441, 1999.

[WWRF] R. Tafazolli, editor. *Technologies for the Wireless Future : Wireless World Research Forum (WWRF)*. Number ISBN : 0-470-01235-8. John Wiley and Sons Inc., October 2004.

Annexes

Annexe A

Les outils de synthèse d'architecture

A.1 L'outil de Synthèse DEFACTO

L'outil DEFACTO [Bond99] a été développé à la fin des années 90 par l'université de Californie du Sud. Son objectif est de raffiner automatiquement la description comportementale d'une application jusqu'à obtenir une description architecturale implémentant les fonctionnalités à réaliser (figure A.1). Le point d'entrée du flot de synthèse est une description haut niveau de l'application à implémenter, les langages d'entrée acceptés sont MatLab et le C.

Le flot de synthèse de l'outil DEFACTO est sensiblement différent de celui des outils de synthèse d'architecture classiques. Cette différence provient de l'architecture ciblée par l'outil : l'architecture cible est hétérogène, composée au minimum d'un processeur généraliste et d'unités configurables (FPGA, ASIC). Le processeur à pour but de s'occuper du pilotage des différents accélérateurs matériels. Chaque accélérateur matériel possède une mémoire interne et des canaux de communication vers d'autres accélérateurs ainsi que vers le contrôleur. Les canaux de communication sont utilisés afin de permettre le transfert des données calculatoires et les retours d'états. Une description de la topologie de l'architecture cible est nécessaire avant de commencer le processus de synthèse. Cette description doit comprendre des informations sur les accélérateurs (nombre de ports, capacité, ...) mais aussi sur les canaux de communication (débits, connexions, ...).

La description comportementale de l'application est transformée à l'aide du compilateur SUIF [Hall96], cela permet l'obtention d'un modèle de représentation parallélisé. Le compilateur SUIF est aussi employé afin de réaliser le partitionnement des différentes tâches (matériel/logiciel). Dans le modèle de représentation interne, certaines parties du graphe sont regroupées sous la forme de clusters, définissant les grappes d'opérations à exécuter sur des accélérateurs matériels différents. Les parties du code source qui sont facilement éligibles à une implémentation sur un accélérateur matériel sont généralement les coeurs de boucle dont sont composées nombre d'applications TDSI. Ces coeurs de boucle sont automatiquement parallélisés et implémentés sur différents accélérateurs matériels à la manière des architectures systoliques (boucles ne possédant pas de dépendances intra-itérations). Afin de pleinement exploiter les possibilités offertes par les accélérateurs matériels, la méthodologie prend en considération des capacités de reconfiguration des composants (possibilité de changer la fonction d'un accélérateur après un délai de

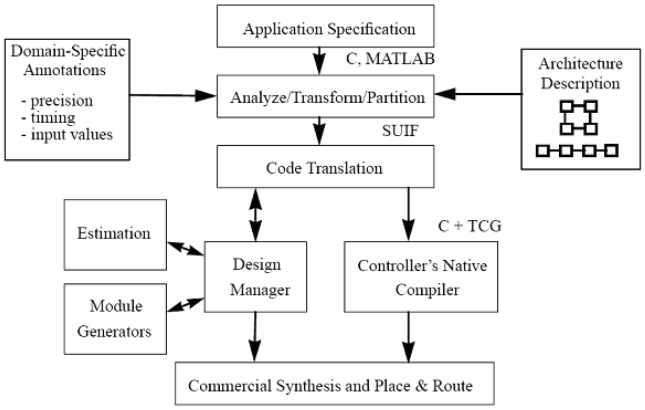

FIG. A.1 – Flot synthèse de l'outil DEFACTO.

reconfiguration).

En sortie, l'outil génère du code architectural de niveau RTL pour les accélérateurs matériels dédiés, et un code source en C pour décrire les actions que doit effectuer le contrôleur implémenté sur un processeur généraliste. La méthodologie proposée par cet outil est, par son principe, réservé aux applications contenant des nids de boucles.

Placement des données en mémoire - Afin de pouvoir exploiter le parallélisme global offert par l'application, l'outil PHDEO va réaliser une phase d'analyse du code source (et plus particulièrement des boucles imbriquées) [Card04]. Des techniques de transformation des boucles sont appliquées afin de réaliser un déroulage partiel des boucles permettant une meilleure analyse du parallélisme et des dépendances inter/intra itérations ceci afin d'implémenter les nids de boucles sur des accélérateurs matériels. Une fois les boucles déroulées partiellement, des transformations sont appliquées sur les indices et les variables afin de les rendre uniques (décomposition structure) permettant ainsi une implémentation systolique simplifiée. Ces transformations permettent d'extraire un placement des données en mémoire en fonction de l'accélérateur matériel sur lequel les calculs sont implantés. La méthodologie permet de générer un mapping mémoire des données au sein de bancs dédiés à un seul accélérateur matériel. Pour permettre à l'outil de synthèse de gérer au mieux le parallélisme d'accès aux données, une méthode de transformation de code est opérée [Dini02]. La transformation du code comportemental vise à modifier l'ordre d'accès aux données au sein des coeurs de boucle en modifiant les indices.

Malgré cela, toutes les données consommées par un accélérateur sont placées dans un banc unique. De plus, les séquences d'accès à la mémoire doivent être nécessairement déterministes, excluant les adressages dynamiques et conditionnels. Le placement des données dans les bancs mémoires est simple : ce dernier consiste à allouer au maximum un banc mémoire par structure de données. Cette approche limite le parallélisme qui peut exister au niveau des accès à la mémoire, provoquant un goulot d'étranglement qui va limiter les performances de l'architecture finale.

Au sein de l'architecture matérielle (figure A.2), des unités sont en charge du calcul des adresses des données qui doivent être accédées. Les architectures élémentaires de calcul d'adresses dédiées nommées

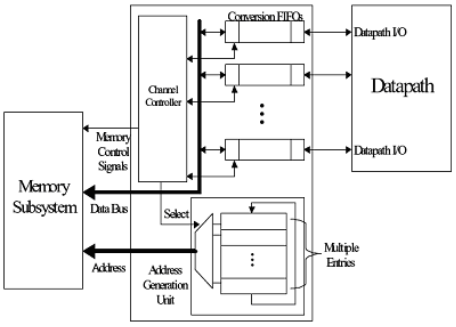

FIG. A.2 – Architecture du séquenceur mémoire utilisé par DEFACTO.

AGU (Address Generation Units) [Dini00] [Park01] ont pour objectif de réaliser les accès nécessaires à la mémoire extérieur de manière transparente vis à vis du chemin de données. Le séquenceur qui est composé d'AGUs est généré en fonction des séquences d'accès déterministes fournies post-ordonnancement du chemin de données, pour chaque banc mémoire alloué à l'application on trouve une AGU.

Les AGUs permettant de générer les adresses des données sont composées de registres et d'opérateurs (incrémentation et décrémentation). Au fur et à mesure de l'exécution de l'application, le contrôleur va faire évoluer les "pointeurs" mémoire en fonction de l'évolution normale des coeurs de boucle, puis, lorsque cela est nécessaire, un rechargement de la valeur du pointeur par un registre sera effectué. Les architectures des AGUs sont optimisées de manière spécifique pour le traitement des nids de boucles (évolution affine des indices, et donc des adresses). Une FSM en interne pilote l'ensemble des ressources allouées au séquenceur.

Le séquenceur ainsi généré a pour principal intérêt de décorréler le problème de la synthèse de l'architecture systolique du problème de la synthèse du chemin de données de la mémoire qui peut être externe au FPGA (figure A.2). Dans ce cas, le séquenceur va être employé pour gérer les conflits d'accès, réaliser des lectures/écritures décalées.

A.2 L'Outil de Synthèse SPARK

SPARK est un outil de synthèse comportementale développé à l'Université de Californie à San Diego dans le groupe "*Microelectronic Embedded Systems*" dirigé par R. Gupta. L'outil de synthèse de haut niveau SPARK possède comme point d'entrée une description comportementale écrite en C-ANSI. Le C-ANSI accepté possède quelques restrictions sémantiques : il n'est pas possible de synthétiser une description contenant des pointeurs, il ne supporte pas non plus les fonctions récursives, les sauts conditionnels et inconditionnels.

SPARK repose sur un flot de synthèse qui privilégie les algorithmes de calcul intensif. Une présentation de l'outil est disponible dans [Gupt02a] [Gupt03a]. Le flot de synthèse employé par SPARK est présenté dans la figure A.3. La description comportementale initiale est transformée par la phase de compilation en une représentation interne de type CDFG. Cette représentation est nommée *SPARK IR (Internal Representation*) et est composée de graphes de tâches où chaque tâche contient un graphe flot de données.

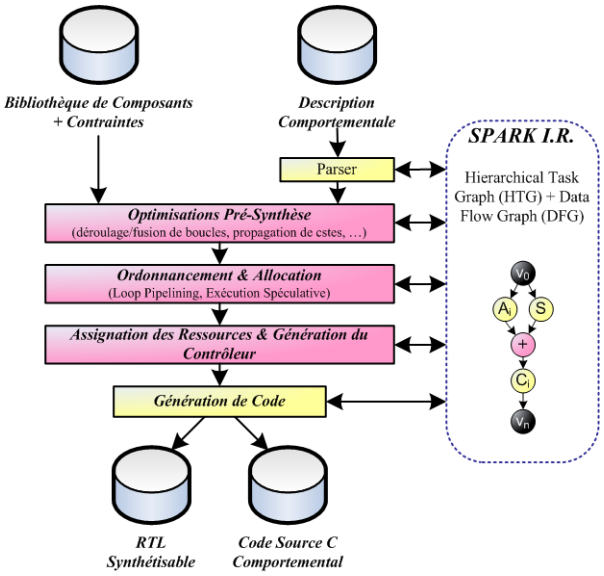

FIG. A.3 – Flot de synthèse de l'outil SPARK.

Afin de réaliser la synthèse, le concepteur doit fournir à l'outil une bibliothèque contenant les opérateurs qui pourront être utilisés durant les étapes de la synthèse ainsi que les contraintes devant guider la synthèse. Les contraintes acceptées par l'outil SPARK sont des contraintes matérielles (nombre et type des opérateurs disponibles). En fonction de ces paramètres, l'outil de synthèse va générer une architecture en optimisant sa latence. Les tâches de sélection et d'allocation des opérateurs sont à la charge du concepteur.

La première étape du flot de synthèse consiste à appliquer une suite d'optimisations sur la description fournie en entrée. Les optimisations appliquées sur le code source de la description sont les suivantes : mise en ligne des procédures *(Function Inlining)*, déroulage partiel ou total des boucles suivant les directives spécifiées par le concepteur (*Loop Unrolling*), fusion de boucles *(Loop Fusion)*, élimination des calculs communs (*Common Sub-Expression Elimination CSE*), élimination du code mort (*Dead Code Elimination*), extraction des invariants de boucle (*Loop-Invariant Code Motion*), réduction de la complexité des opérations (*Operation Strength Reduction*). La gestion du déroulage partiel ou total de chacune des boucles présentes dans la description comportementale fournie à l'entrée de l'outil se fait sous contrôle exclusif de l'utilisateur. Cela permet au concepteur de pouvoir expérimenter différents choix et ainsi choisir de manière manuelle le compromis qui lui semble optimal.

Méthode d'ordonnancement sous ressources contraintes - L'ordonnancement au sein de l'outil se fait en plusieurs étapes. Avant l'ordonnancement, des algorithmes de transformation vont modifier les structures conditionnelles (*Basic Bloc*) présentes dans le graphe afin d'accroître le parallélisme présent en déplaçant des opérations entre les différentes structures. Les algorithmes de "*Code Motions*" qui réalisent ces mouvements entre les différents blocs ont plusieurs objectifs :

1. Augmentation du partage des ressources matérielles : plus on regroupera d'opérations au sein d'une même structure conditionnelle, plus on pourra exploiter le parallélisme existant entre les

opérations et ainsi lisser le taux d'utilisation des ressources matérielles.

2. Diminution de la durée du chemin critique (latence de l'architecture). La réduction de la latence de l'architecture est une conséquence directe de l'augmentation du partage des ressources matérielles, réduisant ainsi la durée totale d'exécution des "basic blocs".

Une fois les transformations appliquées sur le graphe, chaque structure conditionnelle est ordonnancée de manière indépendante. Le CDFG une fois ordonnancé va servir à la génération de la machine à états finis (FSM) qui va piloter l'architecture décrite au niveau transfert de registre (RTL).

Déroulement de l'ordonnancement - Au niveau de l'ordonnancement, différentes optimisations sont apportées à l'aide d'heuristiques : "*Speculative, Percolation and Trailblazing Code Motion*" [Gupt01a], "*Dynamic Renaming of Variables*" [Gupt03a] afin là aussi d'augmenter le parallélisme entre les opérations. L'algorithme d'ordonnancement utilisé est de type "*list-scheduling*" [Gupt01a]. Suite à cet ordonnancement, l'assignation des opérations sur les opérateurs matériels est réalisée. Durant cette opération, la logique nécessaire à la communication entre les composants et les registres est générée. La dernière étape consiste à réaliser la génération de l'automate de contrôle qui va piloter l'architecture. Il est généré à partir du mapping des opérations qui vient d'être réalisé.

L'automate de contrôle qui est généré en sortie de l'outil SPARK correspond à une FSM de Moore dans laquelle on trouve des conditions sur les transitions permettant la gestion des boucles et des branches conditionnelles. En sortie, l'outil génère une architecture décrite au niveau VHDL-RTL synthétisable. L'outil propose aussi en sortie du VHDL de niveau comportemental (correspondant à la description d'entrée), ainsi que du C niveau RTL permettant de visualiser plus facilement les optimisations réalisées par l'outil et de réaliser une simulation afin de vérifier l'intégrité de la solution.

La partie mémorisation de l'architecture cible - Au sein de l'architecture cible de l'outil SPARK, un contrôleur mémoire permet d'accéder aux différentes données mémorisées dans les unités de mémorisations [Gupt03c] comme cela est présenté dans la figure A.4. Le contrôleur est constitué de plusieurs multiplexeurs qui vont permettre un routage des données depuis les bancs mémoire vers les unités de calculs. Les séquences d'accès traitées par le contrôleur ne sont pas statiques et des conflits peuvent être gérés [Gupt03c].

A.3 L'outil de synthèse PICO

PICO est développé par Hewlett-Packard depuis 5 ans. Il a donné naissance à une start-up intitulée *Synfora*. Le système *PICO* complet est un outil de Synthèse/Co-Design. Il propose une méthodologie capable, à partir d'une description algorithmique en langage C, de produire une architecture implémentant la-dite application. L'architecture est basée sur un ensemble d'accélérateurs matériels élémentaires nommés *NPA (Non- Programmable Accelerator*) qui permet d'accélérer le traitement des nids de boucles dans les applications TDSI [Schr02] [Kath02]. L'architecture cible est présentée en figure A.5. Elle se

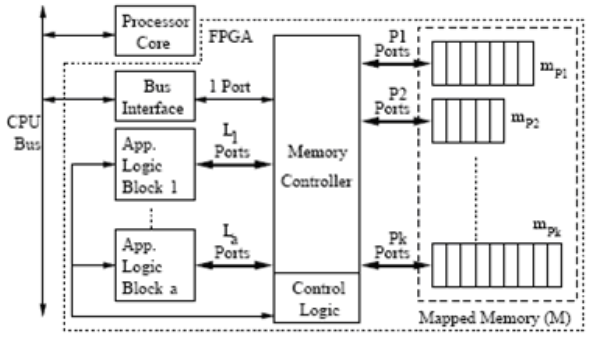

FIG. A.4 – Architecture du contrôleur mémoire utilisé (SPARK).

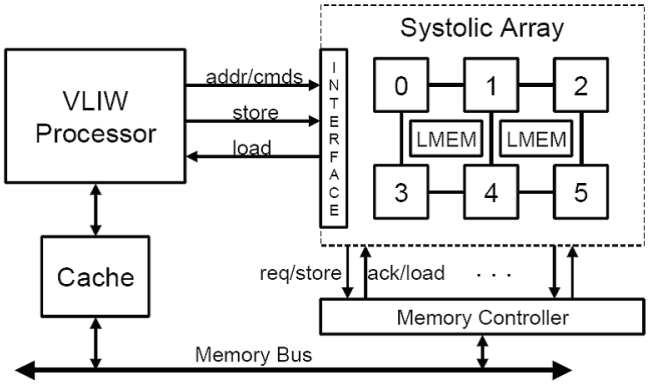

FIG. A.5 – Architecture cible de l'outil PICO-NPA.

compose d'un processeur VLIW qui va réaliser les opérations extérieures aux nids de boucles et qui va piloter un réseau systolique de processeurs.

La première étape consiste à isoler la partie calcul intensif du code et de la dériver en un NPA. La synthèse du NPA est réalisée sous contrainte de latence, l'objectif étant de respecter la contrainte temporelle tout en minimisant le coût de l'architecture. Le concepteur peut aussi contraindre la synthèse en jouant sur la bande passante maximum allouée à la mémoire (limitation du nombre de transferts simultanés).

Lors de la synthèse des processeurs du réseau, l'outil va chercher les unités fonctionnelles dans des bibliothèques où les composants sont déjà caractérisés (surface, délai). L'outil exploite les différentes largeurs de bits entre les opérations et les données lors de la sélection du matériel à mettre en oeuvre. L'architecture générée après synthèse est décrite au niveau RTL.

Chaque processeur composant le réseau systolique est composé de manière identique à ses voisins. Il est composé de files de registres et d'unités fonctionnelles comme cela est décrit dans la figure A.6.

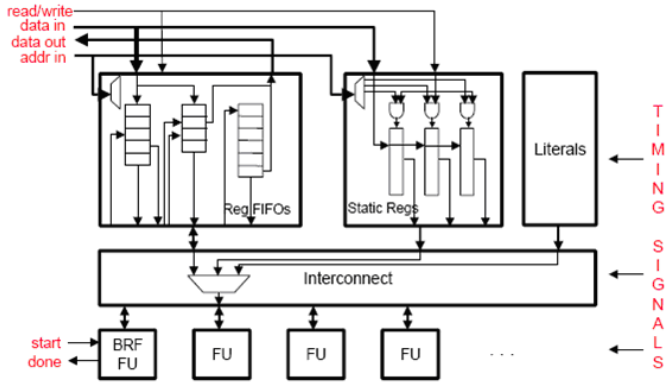

FIG. A.6 – Architecture d'un processeur (PICO).

A.4 L'outil Catapult-C

L'outil Catapult-C est un outil de synthèse de haut niveau, développé par *Mentor Graphics* [Cata04b] [Cata04c] [Mentor]. Le point d'entrée de l'outil est du C++ ANSI écrit sans notion temporelle. Cela rend l'outil compatible SystemC. Le point de sortie de l'outil Catapult-C est une description de niveau RTL ciblant indépendamment les technologies ASIC et FPGA. Les langages de niveau RTL sont VHDL et Verilog, les scripts générés ciblent les outils ModelSim, Design Compiler et Précision RTL.

L'outil autorise le concepteur à utiliser les formats de données présents dans le langage C/C++ à condition que ces formats soient entiers. Durant la phase de synthèse, en fonction des formats des données, un nombre de bits différents sera utilisé pour réaliser les calculs au sein du chemin de données *(bool(1), char(8), short (16), int(32), long(32))*. Il est aussi possible d'utiliser nativement les formats proposés par SystemC et qui sont plus flexibles (*sc_int*). Dans ce cas, l'utilisateur peut définir de manière exacte les largeurs de bits qu'il veut employer pour chacune de ses variables. Ces informations seront prises en considération au niveau de la synthèse.

Lors de la génération de la représentation interne, les techniques habituelles de propagation de constantes et d'analyse des bornes des boucles sont employées afin d'optimiser la description de manière automatique.

Les optimisations mémoire - Il existe plusieurs voies permettant de gérer la mémoire dans l'outil Catapult-C. Dans un premier temps, l'outil va détecter la présence de vecteurs ou de tableaux au sein de la description algorithmique. En fonction de leur taille, il va les convertir soit en files de registres, soit en mémoires internes ou mémoires externes. Une fois que ces données ont été mappées dans des bancs mémoires séparés, l'utilisateur peut décider d'appliquer un certain nombre de transformations pour optimiser l'architecture :

– Sélection du type de mémoire : l'utilisateur peut spécifier à l'outil avant synthèse quel est le type de mémoire qu'il faut mettre en oeuvre. Les différentes possibilités sont : file de registre multiplexée, mémoire mono-port ou multi-ports.
– Fusion des bancs mémoire alloués : une fois le type des mémoires spécifié, il est possible de fusionner

différents tableaux au sein d'une même mémoire afin de réduire la taille des structures de contrôle ainsi que les éléments alloués et non utilisés.
– Augmentation de la taille des bus : afin d'augmenter le parallélisme au sein du chemin, l'outil va augmenter la taille du bus de données afin de transmettre *n* données contiguës en parallèle. Les blocs de données de taille n qui sont maintenant les blocs de transfert élémentaires sont statiques en fonction du positionnement des données au sein du banc mémoire. Cette technique est intéressante dans les applications réalisant des accès contiguës tout au long de leur exécution (mais donc inutilisable pour les accès poinçonnés).

Transformations et implémentation des structures de contrôle - Les structures conditionnelles sont prise en considération au niveau du modèle de représentation interne qui possède des arcs de dépendance de données ainsi que des arcs de dépendance de contrôle permettant de modéliser les opérations conditionnées. Ce modèle de représentation à base d'arcs conditionnels permet aussi de représenter les opérations mutuellement exclusives en vue de partager les opérateurs durant la phase d'ordonnancement. L'analyse qui est réalisée afin de déterminer les véritables exclusions mutuelles au niveau de la description algorithmiques est une analyse structurelle (analyse des structures conditionnelles du code source). Cette technique ne permet la détection que des branches mutuellement exclusives d'une structure *if-then-else* ou *switch-case*. Le traitement des boucles au sein de l'outil est laissé au soin du concepteur. Ce dernier devra choisir pour chacune d'elles le traitement qu'il souhaite lui appliquer : enroulée, déroulage partiel ou déroulage complet. Les différentes boucles de l'architecture seront implémentées de manière séquentielle même dans le cas où il n'y a pas de dépendance de données entre elles. Cette contrainte est imposée par le modèle de représentation interne. Afin de contourner ce problème, une méthode de fusion de boucle semi-automatique est proposée au concepteur. Cela permet de paralléliser des traitements qui auraient été séquentialisés.

Interfaçage avec le reste du système - En ce qui concerne les interfaces de communication, l'outil utilise des méthodes permettant d'interfacer un grand nombre de ports d'entrées / sorties : *streaming*, mémoire simple ou double port, *FIFO*, bus *AMBA* ou tout autre composant de communication généré par le concepteur à l'aide de l'outil CatapultC library builder tool. Il est aussi possible de préciser à l'outil où sont mappées les données lorsque l'on souhaite utiliser des mémoires partagées afin de communiquer avec le système.

Utilisation des composants caractérisés - Afin de réaliser la sélection, l'outil va rechercher des composants dans des librairies. Ces derniers ayant déjà été caractérisés, des informations fiables sur leurs coûts en surface, délais, consommation, etc. leur sont associés. La synthèse se fait sous contraintes de ressources et d'interfaces. Lors de la synthèse, l'outil est capable de gérer les différentes largeurs de bits des données et opérations manipulées. Durant les différentes étapes de la synthèse, des conseils sont prodigués au concepteur sur les goulots d'étranglement présents dans le circuit. Cela doit lui permettre de modifier la description algorithmique manuellement afin de réduire les problèmes de limitation des accès mémoire (bande passante) et de dépendances dans les coeurs de boucle (empêchant l'exploitation du parallélisme).

Annexe B

Détection des opérations mutuellement exclusives

La méthode présentée par [Pena00] vise à détecter dans une description comportementale les branches mutuellement exclusives à l'aide d'une analyse de la description algorithmique de manière équivalente à l'approche présentée dans [Waka89]. L'idée est d'annoter chacune des opérations du graphe à l'aide d'une condition d'exécution. Pour évaluer la possibilité pour deux opérations de s'exécuter en même temps, il suffit de comparer leurs conditions d'exécution respectives.

La méthode d'analyse de la description et de détection des opérations mutuellement exclusives est composée de 4 étapes principales :

1. Transformation de la description comportementale : avant l'analyse, une phase de transformation du code va être appliquée afin de simplifier l'analyse en modifiant les expressions et les structures conditionnelles. La finalité des transformations est d'obtenir un réseau de conditions "basiques" permettant de réaliser une analyse plus simple des branches (opérations) mutuellement exclusives. Pour cela, il faut décomposer les structures conditionnelles en utilisant la propriété de distributivité. Une méthode de duplication des expressions commune est aussi appliquée sur la description, l'objectif étant d'éliminer les variables internes pour rendre la phase d'analyse plus simple et plus performante.
2. Extraction des conditions d'exécution de chacune des opérations : une *EC* (*Condition d'Exécution*) est un couple (c, v) où c est une condition de base et v la valeur de cette condition avec $v \in \{true, false\}$. Chaque opération de la description est alors annotée par son *EC*.
3. Unification des opérations : si l'on trouve des opérations identiques (même opération et même opérandes) et qu'elles possèdent la même condition d'exécution, alors on les rassemble. Cela est nécessaire afin de regrouper les opérations transformées par la méthode de duplication utilisée dans la première partie.
4. Identification des opérations mutuellement exclusives : pour toutes les opérations, les conditions d'exécution sont comparées 2 à 2 afin de détecter quels sont les couples d'opérations mutuellement exclusives. Au final, on peut créer un tableau de taille $N \times N$ contenant toutes les paires possibles

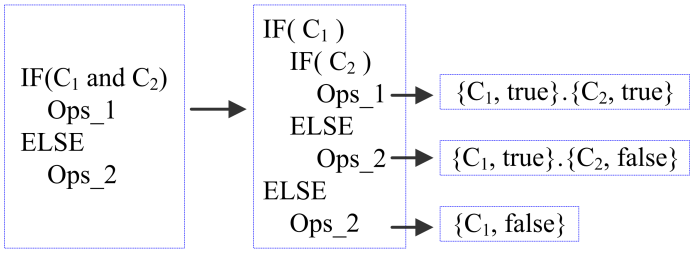

FIG. B.1 – Transformation du code et Extraction des "Condition d'Exécution".

EC_{op1} = { (a=1), false}	EC_{op2} = { (a!=1), true}

FIG. B.2 – Cas de conditions d'exécution syntaxiquement différentes.

d'opérations, avec un marquage pour celles qui sont mutuellement exclusives.

Considérons l'exemple présenté figure B.1. L'exemple est composé d'une structure conditionnelle dont la condition est composée de 2 expressions notées C_1 et C_2. Ces 2 expressions conditionnelles sont de type basiques (opérations logiques). Afin de simplifier l'analyse des conditions, la description algorithmique va subir une transformation qui va éliminer les conditions dites "complexes". Une fois cette transformation appliquée, il faut extraire pour chaque opération sa condition d'exécution afin de pouvoir établir quels sont les couples mutuellement exclusifs par la suite. L'extraction des conditions d'exécution des opérations est réalisée sur l'ensemble des opérations.

Une fois les conditions d'exécution extraites, les opérations sont évaluées en couple afin de déterminer si elles sont mutuellement exclusives. Prenons les deux opérations $\{Ops_1, Ops_2\}$ avec leurs conditions d'exécution respectives : $EC_{ops1}(c_1, v_{set1})$ et $EC_{ops2}(c_2, v_{set2})$. Ces deux opérations sont mutuellement exclusives si et seulement si les conditions d'exécution EC_{op1} et EC_{op2} partagent au moins une même condition qui dans un cas vaut une certaine valeur et dans l'autre cas son contraire. Dans notre exemple, les couples d'opérations mutuellement exclusives sont Opr_1, Opr_2. En ce qui concerne les groupes d'opérations Opr_2 qui sont apparus, on les a regroupés car les EC des 2 groupes étaient compatibles et leur union a donné $C_1, false$ comme règle d'exécution.

Afin d'assurer un meilleur niveau de reconnaissance des opérations mutuellement exclusives, il est nécessaire de transformer dans certains cas les conditions d'exécution afin de normaliser leurs expressions pour se détacher des problèmes syntaxiques. En effet, les deux conditions de la figure B.2 sont sémantiquement équivalentes mais syntaxiquement différentes lors de leur comparaison. Pour pouvoir remédier à ce type de problèmes, il est nécessaire d'appliquer des transformations sur les conditions d'exécution de manière à posséder une représentation unique pour des conditions syntaxiquement différentes et sémantiquement identiques.

Li et Gupta présentent quant à eux une technique [Li1998] nommée *"table de décision temporelle" (Time Decision Table)* qui permet d'extraire de la description comportementale les exclusions mutuelles structurelles, calculatoires et comportementales. Elle se détache des méthodes présentées précédemment par

l'analyse qui est faite à haut niveau et qui permet de retrouver la majorité des opérations mutuellement exclusives grâce à leur modèle de représentation interne.

La méthode d'écrite dans [Koun98] utilise le modèle de représentation *CDG (Conditional Dependency Graph)* composé d'un *HCDG (Hierarchical Conditional Dependency Graph)* présenté par Kountouris dans [Koun02]. Le modèle de représentation de l'algorithme HCDG est relié à un *BDD (Binary Decision Diagram)* au sein du CCFG. La méthode présentée afin de trouver les opérations mutuellement exclusives est une évolution de l'algorithme "*False Path Analysis*" [Berg91]. La méthode permet la détection d'une partie importante des branches mutuellement exclusives. Malgré cela, les inégalités trop complexes ne sont pas prises en considération. Afin de détecter les opérations mutuellement exclusives, on élabore des graphes de conditions suivant les équations présentes dans les structures conditionnelles. A partir de ce graphe, on va analyser les compatibilités/incompatibilités entre les différentes structures conditionnelles afin de détecter les combinaisons qui ne pourront jamais être exécutées en même temps. La technique présentée par Kountouris a pour avantage de prendre en compte les inéquations lors de la recherche des chemins mutuellement exclusifs. Cependant, la méthode basée sur un raisonnement booléen a ses limites dans le cadre des inéquations complexes.

Définition B.0.1 (*Exclusions Mutuelles Comportementales*)

Deux opérations sont dites mutuellement exclusives comportementalement si et seulement si elles appartiennent à des structures conditionnelles différentes qui ne peuvent pas être exécutées au sein du même scénario d'exécution.

Définition B.0.2 (*Exclusions Mutuelles Calculatoires*)

Deux opérations sont dites mutuellement exclusives calculatoirement si et seulement si l'utilisation des résultats produits par ces 2 opérations est mutuellement exclusif.

Penalba et Hermida ont eux exposé dans [Pena00] [Pena02] [Pena02b] une évolution des méthodes présentées précédemment. L'amélioration de la technique permet d'augmenter la détection des opérations mutuellement exclusives. Elle prend en compte les opérateurs relationnels. Grâce à cela, l'analyse permet de détecter les exclusions mutuelles structurelles, comportementales et calculatoires.

Annexe C

Les modèles de représentation

Un langage tel que le C ou le VHDL permet d'exprimer un nombre (trop) important de propriétés d'une spécification. Ces propriétés peuvent être ou non acceptées par un outil de synthèse de haut niveau (HLS) et suivant l'outil aboutir à des interprétations et donc des optimisations de l'implémentation différentes. Nous examinons dans un premier temps les modèles couramment utilisés dans les outils de haut niveau (GAUT, Catapult-C, SPARK, MAHA, etc.).

Les modèles calculatoires permettent idéalement l'expression de une ou plusieurs propriétés. Certains modèles sont plutôt appropriés à des applications dominées par les traitements *(Data-Dominated)* ou dominées par le contrôle *(Control-Dominated)*.

C.1 Le modèle "Data Flow Graph" (DFG)

Les graphes de type "flot de données" (DFG) sont des graphes orientés par un nombre fini d'arcs. Dans ce modèle, les noeuds représentent les opérations et les arcs modélisent les transferts de données entre l'opération productrice et le ou les consommateurs de cette donnée (figure C.1). Au cours du temps, un acteur (noeud du graphe) ne peut être déclenché que lorsqu'il possède une quantité suffisant de données présente à ses entrées. Chaque opération à chacune de ses exécution consomme une donnée sur chacun de ses arcs d'entrées et combine ces entrées de manière à produire une donnée sur chacun des ses arcs de sortie. Les données présentes sur les arcs doivent être produites avant d'être consommées. Cela induit une relation de dépendance d'exécution entre les différents noeuds du graphe. Cette relation implique qu'un noeud ne peut être exécuté qu'après complétion de ses prédécesseurs (noeuds reliés à ses arcs d'entrée) et introduit donc un ordre partiel d'exécution des opérations. Dans le cas d'opérations indépendantes (opérations sans chemin avant ou arrière entre elles), il n'existe pas de relation d'ordre d'exécution. Ces dernières peuvent donc être exécutées dans n'importe quel ordre. Les graphes de type DFG permettent de mettre en évidence, par analyse des dépendances, les opérations pouvant être exécutées en parallèle.

La différence entre un graphe flot de signaux (SDF) et un graphe flot de données (DFG) provient des opérations de type délai (figure C.2). Ces opérations sont utilisées dans le domaine du traitement du signal pour exprimer l'utilisation d'une donnée dont la valeur est calculée dans une itération précédente de

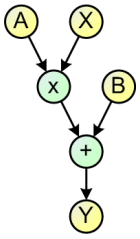

FIG. C.1 – Exemple de DFG modélisant l'équation $Y = A.X + B$.

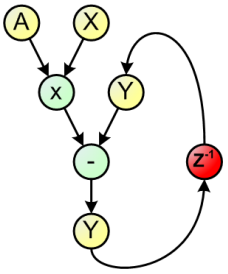

FIG. C.2 – Exemple de SFG possédant un vieillissement sur la donnée Y.

l'algorithme (opérateur Z^{-1} en automatique et TDSI qui porte en particulier sur le vieillissement des signaux et les signaux récursifs de la spécification). Un graphe flot de signaux est un graphe polaire orienté $GFS(V,E)$ où l'ensemble des noeuds $V = \{v_0, \ldots, v_n\}$ représente les opérations, v_0 et v_n étant respectivement le noeud source et le noeud puits. L'ensemble d'arcs $E = \{(v_i, v_j)\}$ représente les dépendances entre les noeuds opérations. Le graphe flot de signaux contient $|V| = n+1$ noeuds. Un arc $e_{i,j} = (v_i, v_j)$ représente une dépendance de données $(v_i \rightarrow v_j)$ entre les opérations v_i et v_j telle que pour toute itération de GFS, l'opération v_i doit démarrer son exécution avant celle de v_j. Pour les dépendances de données, l'exécution de v_j ne peut commencer qu'après la complétion de l'opération v_i.

La modélisation d'algorithme à base de *DFG* est employée dans les outils de synthèse de haut niveau orientés vers les applications TDSI. Ce modèle de représentation présente un avantage considérable : il permet une exploitation du parallélisme maximum de l'algorithme puisque les seules contraintes d'exécution sont les dépendances de données. Il est à noter que les graphes de type *DFG* peuvent être étendus de manière à devenir hiérarchiques.

En contrepartie, il est difficile de représenter des structures de contrôle avec ce formalisme. Même s'il existe des transformations qui permettent de transformer des structures conditionnelles *if-then-else* sous forme de flot de données [Rim92], cela empêche alors une prise en compte efficace de ces structures conditionnelles, par exemple pour le partage des opérateurs. De même, ce modèle de représentation ne permet pas de représenter les structures itératives.

Une des principales limitations des *DFG* réside dans leur incapacité à exprimer les comportements non déterministes où le nombre d'itérations d'une boucle pourrait par exemple varier en fonction des données fournies au composant. Une modélisation de type graphe flot de contrôle et de données (*CDFG*) permet de traiter une plus large classe de descriptions comportementales et laisse le choix d'un ensemble plus

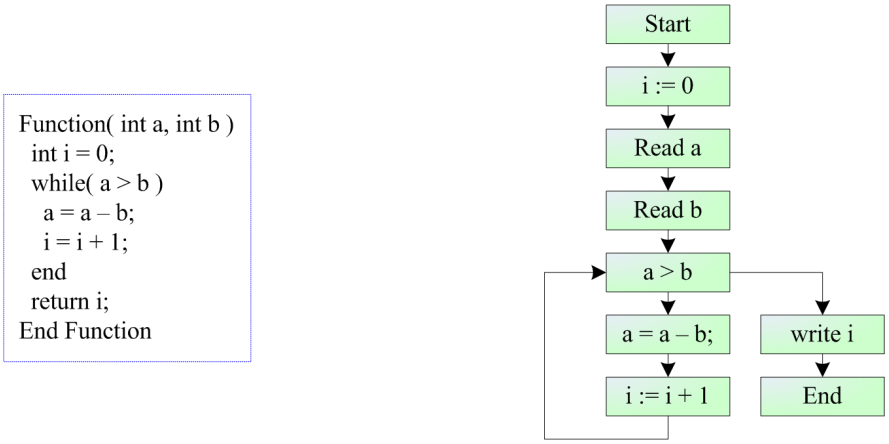

FIG. C.3 – Description comportementale et sa représentation équivalente sous forme de CFG.

vaste de solutions architecturales.

C.2 Le modèle "Control Flow Graph" (CFG)

Les graphes de type "flot de contrôle" *(CFG)* [Aho86] [Gajs96] sont des graphes orientés où les noeuds représentent les structures conditionnelles ou des opérations et où les arcs modélisent des dépendances de contrôle et de séquentialité. Les noeuds conditionnels permettent l'expression de sémantiques de contrôle telles les branches conditionnelles et les boucles. Ce type de graphe est directement dérivé des réseaux de Pétri et est généralement utilisé dans la modélisation d'applications dominées par le contrôle. Les noeuds servant à modéliser le contrôle possèdent une sémantique analogue aux structures de contrôles rencontrées dans les langages de programmation impératifs (mise en séquence, exécution conditionnelle, boucles).

A un instant donné un seul état du graphe de contrôle peut être actif. La contrepartie de cette faculté à représenter de manière optimum les structures de contrôle est l'inexpression du parallélisme entre les opérations.

C.3 Le modèle "Control Data Flow Graph" (CDFG)

Un graphe flot de données et de contrôle est la combinaison d'un graphe flot de contrôle (CFG) avec un ensemble de DFG [Gajs92]. Le CFG modélise les dépendances de contrôle entre les différents DFG au moyen de noeuds spécifiques dont la sémantique est analogue aux structures de contrôles rencontrées dans les langages de programmation impératifs (mise en séquence, exécution conditionnelle, boucles).

La structure du graphe est donc hiérarchique. Cela permet une exploitation locale du parallélisme (interne aux structures de contrôle). Les méthodologies d'exploration architecturale à partir de spécifications algorithmiques, rédigées dans des langages impératifs comme le langage C, font un usage intensif de modèles de type CDFG (figure C.4). Une telle modélisation autorise l'ordonnancement des tâches

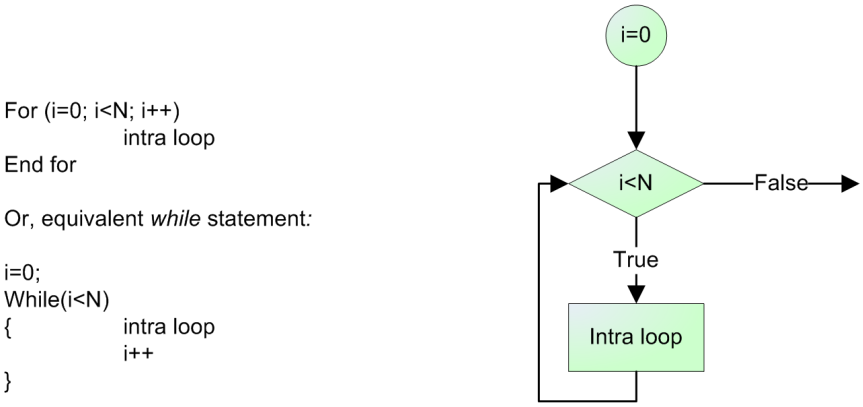

FIG. C.4 – Exemple de modélisation d'une boucle à l'aide d'un CDFG.

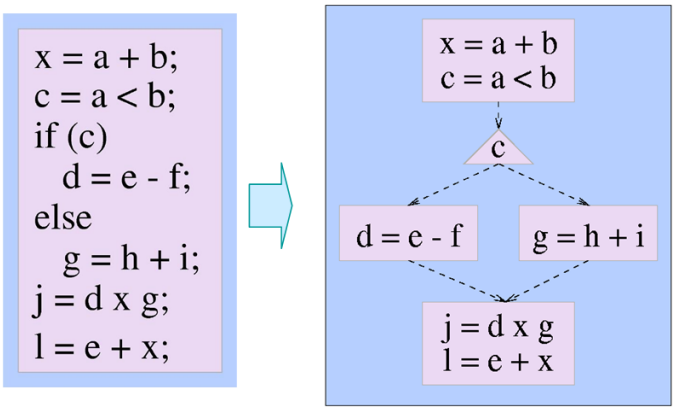

FIG. C.5 – Exemple de CDFG ou le parallélisme est limité.

en fonction de leurs dépendances de données et de contrôle. Les techniques permettant d'exploiter le parallélisme au travers des structures conditionnelles sont malheureusement généralement complexes.

Une autre limitation importante du modèle provient de l'impact du style d'écriture de la description algorithmique qui, pour des descriptions sémantiquement équivalentes, va fournir des graphes différents à cause des noeuds représentant les structures conditionnelles. Il est donc nécessaire pour ce modèle de représentation que le concepteur qui spécifie l'application connaisse le modèle de représentation utilisé ainsi que les méthodes de transformation permettant de passer de la description au modèle. Un exemple montrant la limitation du parallélisme due à la description est présenté dans la figure C.5.

C.4 Assignment Decision Diagram (ADD)

Le modèle de représentation nommé *ADD (Assignement Decision Diagram)* a été développé par D. Gajski en 1992 [Chai92b] [Chai92c]. Afin de réduire l'impact de l'écriture de la description sur le modèle de représentation, les structures conditionnelles sont représentées de manière équivalente à ce qui a été proposé pour étendre les sémantiques des DFG [Rim92]. Cela à pour effet de mettre à plat les différents

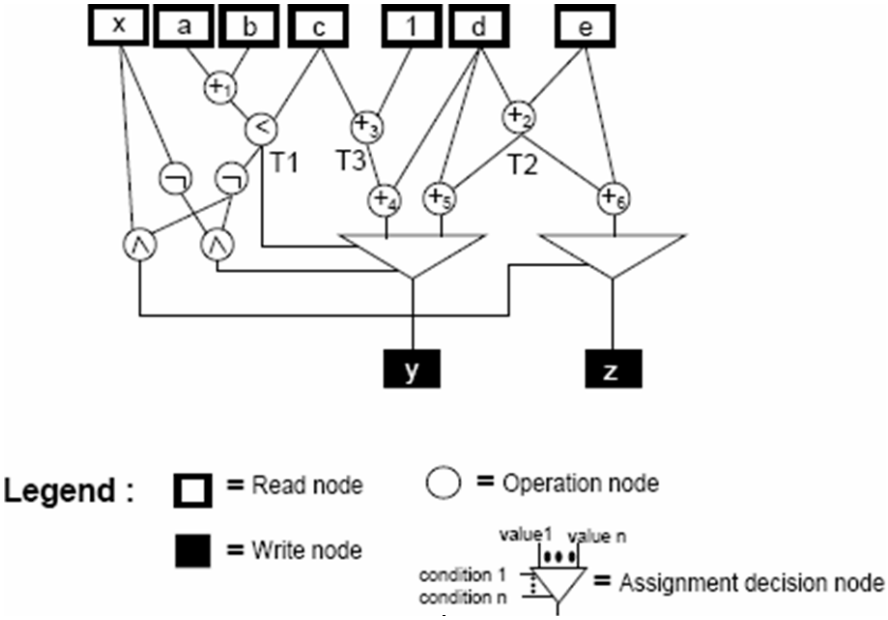

FIG. C.6 – Exemple de modélisation réalisée grâce au modèle ADD [Chai93].

calculs à effectuer dans chacune des branches conditionnelles entraînant une exécution spéculative puis un choix de résultat (figure C.6).

L'idée développée dans ce modèle vient du constat que pour paralléliser l'ensemble des opérations contenues dans les structures conditionnelles, il faut uniquement conditionner l'affectation des mémorisations des données et leur écriture sur les ports de sortie. Pour cela, Gajski développe un nouveau type de noeud nommé "noeud de décision d'assignation" (ADN), qui en fonction de ses paramètres, va décider quelle est la valeur à assigner au résultat. L'utilisation des noeuds ADN permet d'exécuter de manière spéculative toutes les opérations conditionnelles. Seules les affectations sont soumises aux résultats des conditions.

La représentation sous forme d'ADD est composée de 4 parties bien distinctes : l'assignation des résultats, la condition d'assignation des résultats, la décision d'assignation et les calculs des valeurs à assigner. Ces quatre parties sont composées à l'aide de 4 types de noeuds : les opérations, les lectures de données, les écritures de données et les conditions d'assignation.

C.5 Le modèle "Condition Graph" (CG)

Le graphe de condition nommé GC et défini par Juan dans [Juan94] est un graphe dont l'objectif premier est de représenter les conditions d'exécution des opérations. Le graphe prend comme point de départ une description comportementale représentée sous la forme d'un ADD. Dans la représentation ADD la condition d'exécution d'une opération est décrite comme une expression arithmétique qui se ramène à un résultat booléen $\{true, false\}$. Le graphe de condition est composé de 2 types de noeuds : les noeuds de données et les noeuds opérations. Les noeuds opérations représentent les opérations qui réalisent les

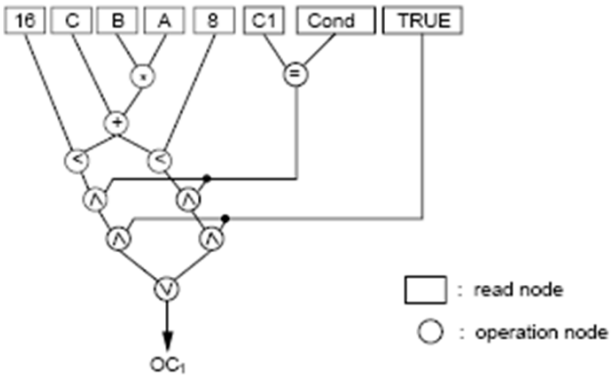

FIG. C.7 – Exemple de modélisation réalisée à l'aide d'un graphe CG.

calculs des conditions.

Une fois dérivée, le graphe de condition représente tous les calculs des conditions relatives aux opérations contenues dans le graphe ADD. Ce graphe est ensuite mis à profit dans une méthode d'identification des opérations mutuellement exclusives.

C.6 Hierarchical Conditional Dependence Graph (HCDG)

Le modèle de représentation nommé *HCDG (Hierarchical Conditional Dependency Graph)* est un modèle de représentation défini par Kuchcinski dans [Kuch01] afin de permettre une modélisation des programmes écrits dans le langage SIGNAL [LeGu91] [Koun01]. Ce modèle de représentation a été défini afin de permettre une modélisation adaptée des structures conditionnelles dans des applications orientées flot de données et de contrôle. L'auteur se base sur le constat que les graphes de type DFG ne sont pas adaptés à la modélisation du contrôle et que les $CFG/CDFG$ sont inadéquats pour le partage de ressources matérielles entre les opérations mutuellement exclusives. Afin de résoudre ce problème, un graphe regroupant les avantages des 2 approches a été développé (figure C.8).

Le graphe est hiérarchique sur 2 niveaux : le niveau inférieur est constitué d'un DFG tripartite constitué de noeuds mémoires et traitements élémentaires nommé "*Conditional Dependency Graph*" ; il représente une séquence d'opérations non conditionnelles. Au sein de ce DFG, les calculs des conditions sont réalisés à l'aide d'opérations logiques. Ces dernières produisent des données qui sont mémorisées dans des noeuds mémoires particuliers nommés *gardiens/horloge*. Ces noeuds sont utilisés pour modéliser les dépendances de contrôle qui régissent l'exécution des noeuds comme le font les dépendances de données. Ces gardiens permettant ou non l'exécution des opérations en fonction des résultats conditionnels, sont reliés au niveau hiérarchique supérieur afin de recréer un arbre de composition des structures conditionnelles nommé "Guard Hierarchy" comme cela est présenté dans la figure C.8.

Cette composition hiérarchique des relations entre les différentes structures conditionnelles permet d'analyser et d'exploiter l'exclusion mutuelle existant entre les différentes branches conditionnelles [Koun99]. Cette dernière est réalisée tout en conservant une sémantique de DFG pour la modélisation des données

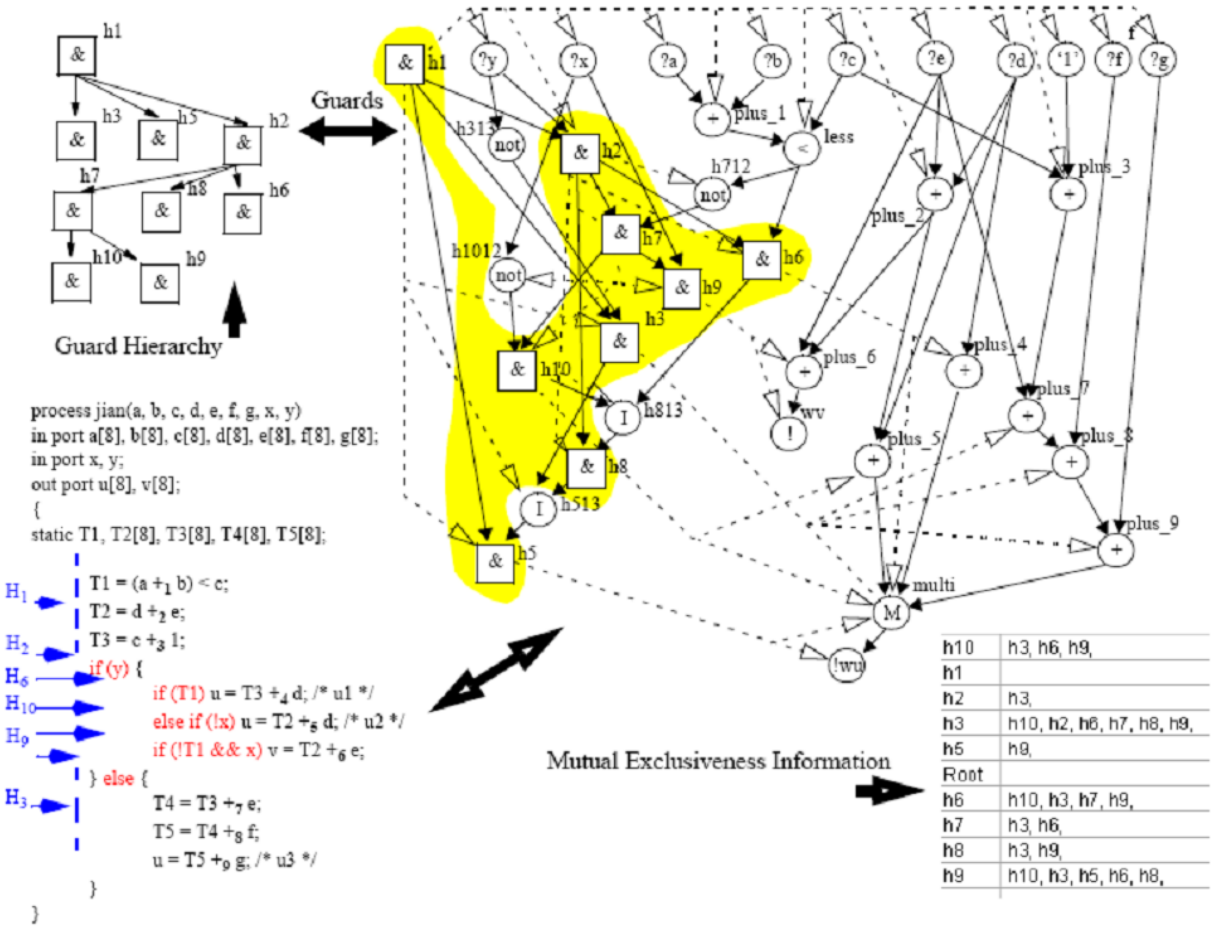

FIG. C.8 – Exemple de représentation réalisée à l'aide d'un HCDG.

et des opérations. Toutefois, ce modèle ne permet pas de modéliser des structures conditionnelles complexes telles les boucles.

C.7 Les autres modèles de représentation

D'autres modèles de représentation existent et sont dédiés à la modélisation de propriétés précises. On peut citer :

– Le modèle "*Hierarchical Task Graph (HTG)*" [Poly91] utilisé dans la modélisation de programmes contenant des parties calculatoires parallèles. Il est en partie basé sur les sémantiques des CDFG.
– Les réseaux de Pétri, composés de quatre éléments de base : un ensemble de places, un ensemble de transitions, une fonction d'entrée (des transitons vers les places) et une fonction de sortie (des places vers les transitions). Ce modèle accepte un formalisme mathématique qui définit les propriétés liées à la structure et aux règles de transition [Pete81]. Deux propriétés intrinsèques importantes des réseaux de Pétri sont leur nature asynchrone et la concurrence. Les événements peuvent arriver de manière complètement indépendante ; il n'y a donc pas d'horloge pour piloter les transitions. Souffrant comme les FSM de l'absence de hiérarchie, des extensions aux réseaux de Pétri hiérarchisés (HRDP) ont été proposées. La nature totalement asynchrone des réseaux peut être modifiée par l'utilisation des réseaux de pétri temporisés. De nombreuses variantes des réseaux de Pétri ont été proposées dans la littérature, permettant d'intégrer des propriétés particulières.