N° d'ordre 01 ISAL 0054 Année 2001

# THÈSE

*présentée*

**DEVANT L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON**

*pour obtenir*

**LE GRADE DE DOCTEUR**

**FORMATION DOCTORALE :** DISIC
**ÉCOLE DOCTORALE :** EDIIS

PAR

**Antoine Fraboulet**

(D.E.A. de l'ENS de Lyon - Université Lyon I - INSA de Lyon)

## OPTIMISATION DE LA MÉMOIRE ET DE LA CONSOMMATION DES SYSTÈMES MULTIMÉDIA EMBARQUÉS

**Soutenue le :** 23 novembre 2001 **devant la Commission d'Examen**

**Jury**

| | |
|---|---|
| Pr Michel AUGUIN, université de Nice, | rapporteur |
| Pr Francky CATTHOOR, université catholique de Louvain (Belgique), | examinateur |
| Pr Anne MIGNOTTE, INSA de Lyon, | directeur |
| Dr Jacques-Olivier PIEDNOIR, société Cadence, | examinateur |
| CR-INRIA HDR Tanguy RISSET, IRISA de Rennes, | rapporteur |
| Pr Yves ROBERT, ENS de Lyon, | président |

**Cette thèse a été préparée au Laboratoire de l'Informatique du Parallélisme (LIP) à l'ENS de Lyon et au Laboratoire d'Ingénierie de l'Informatique Industrielle (L3I) de l'INSA de Lyon**

# Résumé

L'évolution des techniques et des outils de compilation logicielle et de synthèse automatique de matériels permet maintenant de concevoir de manière conjointe (*Codesign*) des systèmes électroniques intégrés sur une seule puce de silicium, appelés « *System on Chip* ». Ces systèmes dans leurs versions embarquées doivent répondre à des contraintes spécifiques de place, de vitesse et de consommation. De plus, les capacités sans cesse croissantes de ces systèmes permettent aujourd'hui de développer des applications complexes comme les applications multimédia. Les applications multimédia travaillent, entre autres, sur des images et des signaux de grande taille ; elles génèrent de gros besoins en place mémoire et des transferts de données volumineux, traités par des boucles imbriquées. Il faut donc se concentrer sur l'optimisation de la mémoire lors de la conception de telles applications dans le monde de l'embarqué. Deux moyens d'action sont généralement mis en œuvre : le choix des architectures (hiérarchies mémoire et mémoires caches) et l'adéquation du code décrivant l'application avec l'architecture générée. Nous développerons ce second axe d'optimisation de la mémoire et comment transformer automatiquement le code de l'application, en particulier les boucles, pour minimiser les transferts de données (grands consommateurs d'énergie) et la place mémoire (grande utilisatrice de surface et d'énergie).

# Résumé en anglais – ***Abstract***

The development in technologies and tools for software compilation and automatic hardware synthesis now makes it possible to conceive in a joint way (Codesign) the electronic systems integrated on only one silicon chip, called "System on Chip". These systems in their embedded versions must answer specific constraints of place, speed and consumption. Moreover, the unceasingly increasing capacities of these systems makes it possible today to develop complex applications like multimedia ones. These multimedia applications work, amongst other things, on images and signals of big size; they generate large memory requirements and data transfers handled by nested loops. It is thus necessary to concentrate on memory optimizations when designing such applications in the embedded world. Two means of action are generally used: the choice of a dedicated memory architecture (memory hierarchy and caches) and adequacy of the code describing the application with the generated architecture. We will develop this second axis of memory optimizations and how to transform automatically the implementation code, particularly nested loops, to minimize data transfers (large consumer of energy) and memory size (large consumer of surface and energy).

# Table des matières

**Introduction** **1**

**1 Codesign et architectures** **7**
1.1 Évolution de la conception de circuits intégrés . . . . . 7
1.2 Conception de systèmes sur silicium . . . . . 10
1.3 Architectures des *SoC* et consommation . . . . . 13
1.3.1 Les unités de calcul . . . . . 14
1.3.2 Les communications . . . . . 20
1.3.3 Les mémoires . . . . . 21
1.3.4 Hiérarchies de mémoires et mémoires caches . . . . . 27
1.3.5 Conclusion sur les architectures et les hiérarchies . . . . . 34
1.4 Motivations pour les transformations de programme . . . . . 35
1.4.1 Applications cibles . . . . . 35
1.4.2 Optimisation mémoire et placement dans le flot de conception . . . . . 36

**2 Transformations pour une architecture définie** **39**
2.1 Analyse de programme et transformations . . . . . 39
2.1.1 Dépendances de données . . . . . 40
2.1.2 Représentations usuelles des dépendances . . . . . 42
2.1.3 Considérations sur les langages utilisés et la clarté des programmes . . . . . 44
2.2 Accès aux données et critères d'optimisation . . . . . 44
2.2.1 Place mémoire d'une application . . . . . 45
2.2.2 Localité des accès . . . . . 45
2.3 Transformations pour une architecture matérielle fixe . . . . . 47
2.3.1 Transformations pour organiser les données en mémoire . . . . . 47
2.3.2 Transformations de boucles . . . . . 48
2.4 Optimisation conjointe de l'architecture et du code : *DTSE* . . . . . 51
2.4.1 Transformations de boucles et de contrôle pour la DTSE . . . . . 54
2.4.2 Placement géométrique . . . . . 55
2.4.3 Méthode constructive pour l'ordonnancement . . . . . 56
2.4.4 Extension de l'heuristique proposée . . . . . 61
2.4.5 Réduction de la complexité . . . . . 64
2.4.6 Résultats expérimentaux . . . . . 64
2.5 Conclusion . . . . . 66

**3 Transformations pour une architecture abstraite 69**
3.1 Place mémoire d'un système et localité des calculs . . . 69
3.2 Possibilités offertes par les transformations . . . 70
3.3 Fusion de boucles pour la minimisation des tableaux temporaires . . . 71
3.3.1 Travaux existants sur la fusion . . . 72
3.3.2 Modélisation du problème . . . 73
3.3.3 Détection des tableaux supprimables . . . 74
3.3.4 Conflits entre tableaux . . . 75
3.3.5 Fusion et réécriture du code . . . 78
3.3.6 Expérimentations sur des graphes aléatoires . . . 79
3.3.7 Conclusion sur la fusion . . . 80
3.4 Alignement de boucles pour la minimisation des dépendances entre itérations . . . 81
3.4.1 Modélisation du problème pour les boucles simples . . . 81
3.4.2 Formulation du programme linéaire en nombres entiers (PLNE) . . . 82
3.4.3 Polynomialité du problème et algorithme . . . 83
3.4.4 Alignement multidimensionnel . . . 89
3.4.5 Conclusion sur l'alignement . . . 91
3.5 Combinaison des transformations et conclusion . . . 92

**4 Mise en pratique et intégration des transformations 95**
4.1 Cadence *VCC : Virtual Component Codesign* . . . 96
4.1.1 Définition des blocs comportementaux . . . 97
4.1.2 Définition des blocs architecturaux . . . 98
4.1.3 Assignation ou *Mapping* . . . 99
4.1.4 Simulations d'un système . . . 100
4.1.5 Analyse des simulations . . . 101
4.2 Expérimentations avec VCC : estimations et optimisations pour la mémoire . . . 101
4.2.1 Optimisations indépendantes de l'architecture . . . 103
4.2.2 Modification de l'architecture et du comportement . . . 104
4.3 Conclusion . . . 106

**5 Conclusion et perspectives pour la synthèse de systèmes 107**

**Bibliographie 109**

**Publications 117**

# Table des figures


1.1 Synthèse de haut niveau : flot de conception simplifié . . . . . . . . . . . . . . . . . . . 8
1.2 Principales phases de conception conjointe d'un système . . . . . . . . . . . . . . . . . 11
1.3 Conception conjointe matériel/logiciel : partitionnement et assignation . . . . . . . . 12
1.4 Exemple d'architecture pour un SoC . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.5 Transistor à effet de champs MOS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1.6 Élément de stockage d'une mémoire SRAM : bascule D . . . . . . . . . . . . . . . . . . 22
1.7 Structure simplifiée d'une mémoire SRAM 4x2 . . . . . . . . . . . . . . . . . . . . . . 23
1.8 Organisation d'une SRAM utilisant un décodage en deux étapes . . . . . . . . . . . . 24
1.9 Schéma de principe d'une cellule de mémoire DRAM . . . . . . . . . . . . . . . . . . 25
1.10 Conception d'un bloc de mémoire DRAM . . . . . . . . . . . . . . . . . . . . . . . . 25
1.11 Architecture d'une hiérarchie mémoire . . . . . . . . . . . . . . . . . . . . . . . . . . 28
1.12 Architecture d'une mémoire cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
1.13 Mémoire auxiliaire : *Scratch Pad Memory* . . . . . . . . . . . . . . . . . . . . . . . . 32
1.14 Optimisations dans le flot de conception . . . . . . . . . . . . . . . . . . . . . . . . . 37

2.1 Transformations pour une architecture donnée . . . . . . . . . . . . . . . . . . . . . . 40
2.2 Transformations pour une architecture dédiée . . . . . . . . . . . . . . . . . . . . . . . 40
2.3 Graphe de dépendances développé . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
2.4 Graphe de dépendance réduit utilisant des vecteurs de distance . . . . . . . . . . . . 43
2.5 Exemple de fusion de boucles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
2.6 Exemple de décalage de boucle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
2.7 Exemple de pavage de boucle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
2.8 Exemple d'échange de boucles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
2.9 Flot d'optimisation simplifié de la méthodologie DTSE . . . . . . . . . . . . . . . . . 53
2.10 Cône de dépendance et cône d'ordonnancements possibles . . . . . . . . . . . . . . . 56
2.11 Code de la spécification initiale . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
2.12 Espace d'itération commun initial du code de la figure 2.11 (N=5) . . . . . . . . . . 57
2.13 Espace d'itération commun final . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
2.14 Mauvais choix pour l'ordre des vecteurs . . . . . . . . . . . . . . . . . . . . . . . . . 58
2.15 Meilleur choix possible d'ordonnancement . . . . . . . . . . . . . . . . . . . . . . . . 60
2.16 Génération du code pour la figure 2.15 . . . . . . . . . . . . . . . . . . . . . . . . . . 61
2.17 Exemple d'échec pour l'heuristique simple . . . . . . . . . . . . . . . . . . . . . . . . 61
2.18 Meilleur ordonnancement pour la détection du mouvement dans MPEG-4 . . . . . . 65
2.19 Autre ordonnancement possible pour la détection du mouvement dans MPEG-4 . . . 65

3.1 Comparaison sur un cas simple de la maximisation de la localité proposée par McKinley et Kennedy et de la minimisation des tableaux de calculs temporaires. . . . . . . 72
3.2 Exemple de programme et son graphe de dépendances associé . . . . . . . . . . . . . 74

3.3 Clôture transitive du graphe de la figure 3.2 pour la détection des tableaux supprimables 75
3.4 Conflit direct de fusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
3.5 Conflit sur un cycle de fusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
3.6 Importance de l'ordre des fusions . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.7 Graphe de dépendances groupé et code modifié . . . . . . . . . . . . . . . . . . . . 79
3.8 Modélisation des dépendances dans une boucle . . . . . . . . . . . . . . . . . . . . 82
3.9 Transformation du graphe pour la résolution à l'aide d'un algorithme de flot . . . . . 85
3.10 Transformation du graphe pour la résolution à l'aide d'un algorithme de flot . . . . . 85
3.11 Exemple de la figure 3.8 après minimisation des variables temporaires d'itération . . 89
3.12 Modélisation des dépendances pour un nid de boucles . . . . . . . . . . . . . . . . 89
3.13 Exemple de la figure 3.12 après minimisation des variables temporaires . . . . . . . . 92

4.1 Espace de recherche entre coût et performances . . . . . . . . . . . . . . . . . . . 96
4.2 Diagramme comportemental d'un système de messagerie vocale . . . . . . . . . . . . 97
4.3 Architecture cible d'un système de messagerie vocale . . . . . . . . . . . . . . . . . 98
4.4 Assignation du comportement pour le système de messagerie vocale . . . . . . . . . . 99
4.5 Diagramme comportemental utilisé pour les tests . . . . . . . . . . . . . . . . . . . 102
4.6 Exemple de départ pour les transformations . . . . . . . . . . . . . . . . . . . . . 102
4.7 Diagramme de l'architecture cible et du mapping utilisé . . . . . . . . . . . . . . . 103
4.8 Exemple de départ après fusion de boucles . . . . . . . . . . . . . . . . . . . . . . 104
4.9 Exemple de départ après fusion et réutilisation de la mémoire . . . . . . . . . . . . 104
4.10 Diagramme d'une architecture possible utilisant une mémoire statique locale . . . . . 105

# Introduction

La conception de systèmes numériques a été divisée très tôt entre conception de matériels et conception de logiciels. Ce n'est que récemment que les deux disciplines ont atteint un niveau permettant de les recombiner.

La conception logicielle a commencé dans les années 1950 grâce à la standardisation des processeurs programmables. L'utilisation de ces processeurs était faite par manipulation directe du code machine binaire mais est très vite passée au niveau assembleur grâce à des programmes permettant d'effectuer automatiquement la traduction. Il a fallu moins d'une décennie pour arriver au niveau d'abstraction supérieur représenté par la compilation de langages évolués et structurés. Depuis lors, la conception logicielle n'a cessé d'évoluer vers des niveaux d'abstraction toujours plus élevés (programmation orientée objet ou spécification formelle) grâce à l'évolution des outils.

Dans le même temps, les concepteurs de matériel ont, eux aussi, mis en place des niveaux d'abstraction toujours plus élevés. Les premières conceptions étaient réalisées en manipulant directement les transistors d'un circuit ; mais l'évolution des outils de conception physique a permis l'utilisation de portes logiques et a automatisé les étapes de placement et de routage des transistors. Puis, les outils de synthèse logique et séquentielle ont permis d'utiliser des machines à états finis et des équations booléennes en s'appuyant sur les outils de conception physique. Ces quinze dernières années ont vu apparaître la conception en transferts de registres puis la conception comportementale permettant de générer des circuits à partir d'algorithmes écrits dans des langages aussi structurés que pour la conception de logiciels.

L'évolution et l'automatisation des outils de conception de matériel ont été plus lentes que celles du logiciel, sans doute à cause d'un espace de recherche plus important et d'exigences très appuyées de la part des concepteurs sur la qualité des optimisations mises en œuvre, mais il est maintenant à nouveau nécessaire de recombiner conception du matériel et du logiciel. Les techniques ont atteint un niveau suffisant pour qu'une conception puisse être compilée soit pour une architecture standard, soit sur une architecture synthétisée sous forme de matériel spécialisé, soit encore comme une combinaison des deux créant ainsi la *synthèse de systèmes*.

Les systèmes embarqués (téléphones portables, agendas personnels électroniques ou les systèmes de guidage et de vidéo embarqués dans les engins mobiles) utilisent cette dernière méthode de conception car ils doivent répondre à des contraintes spécifiques de place, de consommation et de performances. Les applications multimédia travaillent entre autres sur des flux d'images et des signaux ; elles génèrent de gros besoins en mémoire et des transferts de données volumineux. Il faut donc se concentrer sur l'optimisation de la mémoire lors de la conception de telles applications dans le monde de l'embarqué.

Selon la « loi de Moore », la densité des circuits en nombre de transistors par puce double tous les 18 mois. Cette « loi », remarquablement vérifiée depuis presque 20 ans, a permis d'augmenter la puissance des machines ainsi que la capacité des mémoires embarquées sur le silicium. Un des mauvais côtés de cette intégration de plus en plus poussée est que les mémoires classiques, composées essentiellement de condensateurs, n'ont pu suivre les évolutions en performances et intégration demandées par les processeurs modernes. Il n'est pas rare de trouver maintenant des machines avec des mémoires principales ne fonctionnant qu'au dixième de la vitesse du processeur. Un processeur a pourtant besoin de lire, et éventuellement d'écrire, dans la mémoire à chaque instruction qu'il exécute. Cette différence de performance pénalise énormément le bon déroulement d'un programme et limite la quantité de données qu'il peut manipuler, même si ses performances en calcul pur ne sont pas exploitées pleinement.

L'atténuation de ce problème de différence de performances a nécessité la mise en place de hiérarchies complexes de mémoires (mémoires caches) ainsi que le développement de techniques d'optimisations de code permettant d'utiliser au mieux cette structure. Ces hiérarchies jouent un rôle primordial sur la gestion des communications entre calcul et stockage des données.

Par ailleurs, si la surface des circuits diminue pour les circuits logiques, il n'en va pas de même pour la surface des mémoires classiques. Celles-ci sont encore volumineuses et il n'est pas rare de trouver des systèmes dont plus de la moitié de la surface est utilisée par de la mémoire. Cette surface ainsi que la consommation électrique qu'elle engendre constituent, avec les limites de performances et de bande passante, un goulot d'étranglement qu'il est devenu nécessaire de prendre en compte lors de la conception d'un système, et plus particulièrement pour les systèmes embarqués car ils sont généralement alimentés par des batteries limitées en taille et en poids.

Les solutions pour améliorer les mémoires des systèmes sont nombreuses et peuvent être rangées dans deux grandes catégories : la construction de hiérarchies mémoires matérielles dédiées et les optimisations du code des programmes effectuées lors de la compilation des applications.

## Choix architecturaux pour la mémoire : applications multimédia

On peut maintenant construire des machines avec de grandes quantités de mémoire cache embarquée sur la même tranche que le processeur. Cependant cette technique ne fonctionne que dans des cas très réguliers et il arrive souvent qu'elle soit inefficace pour certains types d'applications ayant des accès à la mémoire irréguliers ou utilisant des motifs de répétition ne convenant pas aux algorithmes de gestion des caches. L'utilisation de caches dans les systèmes embarqués présente de nombreux inconvénients. Tout d'abord la présence d'un cache introduit des temps d'accès variables aux données et complique de façon significative les méthodes de prédiction de temps d'exécution des systèmes ; ce qui est très dommageable pour le cas des applications contraintes dans un environnement temps réel. Bien qu'il soit possible de déterminer un temps d'exécution pour « le pire cas » ce calcul est une surestimation très pessimiste des performances réelles d'un système. Une des conséquences de cette surestimation du temps est que les compilateurs ne peuvent optimiser complètement le code pour la plate-forme cible et qu'il est souvent nécessaire de sur-dimensionner celle-ci pour s'affranchir des aléas de temps d'accès. Enfin, certains types d'applications, comme les applications multimédia et dans une moindre mesure les applications réseaux utilisant des protocoles évolués comme ATM, utilisent des accès à la mémoire très réguliers mais qui ne correspondent pas toujours aux techniques de gestion utilisées dans les mémoires caches.

Dans cette thèse on s'intéresse plus particulièrement aux applications à utilisation intensive de la mémoire comme c'est le cas dans le domaine du multimédia. Ces applications ont la particularité de nécessiter de grandes quantités de mémoire et d'en faire un usage intensif pour manipuler des flux de son, de vidéo ou d'images fixes. Elles impliquent un parcours régulier de la mémoire et des contraintes temporelles. Ceci n'est réalisable que sur une architecture mémoire hiérarchique complexe. D'après les estimations du *2000 International Technology Roadmap for Semiconductors*, près de la moitié de la surface des systèmes actuels est dédiée à la mémoire, cette proportion devrait passer à 94% aux alentours de 2014. Il est donc extrêmement important de prendre en compte les optimisations possibles sur la mémoire et les communications dans les hiérarchies pour contrôler la taille et la consommation des systèmes.

Les systèmes embarqués sont, par nature, dédiés à une utilisation spécifique prévue dès la conception et sont souvent utilisés pour n'effectuer qu'une seule tâche ou un ensemble fixe et bien défini de tâches. Ces particularités permettent de dimensionner précisément un système en fonction de la charge de travail qu'il aura à effectuer. On peut ainsi avoir la liberté de concevoir un système complet avec une architecture composée de ressources (processeurs, mémoires, mémoires caches, mémoires dédiées, bus de communications... ) empruntées à des technologies standards ou bien de concevoir des modules spécifiques pour le système à l'aide d'outils adaptés.

Les architectures sont construites sur mesure et sont dédiées afin d'être le mieux dimensionnées possible par rapport aux compromis entre coût, performances, encombrement physique et consommation. De plus en plus de systèmes embarqués sont des systèmes portables fonctionnant sur batterie. Il est donc capital de prendre en compte la gestion de la consommation d'énergie dès le départ de la conception afin de permettre d'avoir une autonomie plus importante ou des batteries de plus petite taille. Une architecture classique de système embarqué est composée d'une ou plusieurs unités de calcul (processeurs généralistes ou dédiés, circuits spécialisés) et d'une hiérarchie mémoire comportant différents niveaux de cache et plusieurs modules de mémoire gérés indépendamment. Un mauvais choix d'architecture mémoire (taille des mémoires caches, contrôle de leur gestion, nombre de niveaux) introduit un goulot d'étranglement très complexe, voire impossible à contourner dans la suite de la conception d'un système. L'architecture de la hiérarchie mémoire doit donc être taillée sur mesure, comme tout le reste du système, pour pouvoir profiter au maximum de chaque composant et limiter les échanges de données.

## Sur la base d'un système dédié : exploitation des hiérarchies mémoire

Un moyen de gérer la complexité de conception des systèmes embarqués est d'en modifier le comportement au niveau du code source de l'application multimédia pour faciliter les décisions de conception et tirer parti au maximum des ressources de la hiérarchie mémoire du système. Ces modifications doivent être prises en compte au moment où l'on décide de l'*organisation* du système et de la répartition des tâches entre les différentes unités de calcul.

Les modifications d'un comportement pour l'optimisation mémoire ne sont malheureusement pour l'instant que très peu supportées par les outils de compilation matérielle et logicielle. De telles optimisations modifient la façon dont on accède aux données en tenant compte de l'architecture et de la hiérarchie mémoire. Elles sont à la fois extrêmement dépendantes et contraintes par la micro architecture de la machine cible. De plus, elles sont souvent difficiles à mettre en œuvre pour un concepteur humain et il est assez facile d'introduire des erreurs lors de l'optimisation manuelle d'un

code alors que beaucoup de temps a déjà été passé à valider la fonctionnalité d'un système.

Le type d'application visé (à utilisation intensive de mémoire) impose une architecture à hiérarchie mémoire complexe : banc de registres, cache matériel et/ou logiciel, mémoire embarquée et mémoire externe... Un des facteurs les plus importants d'optimisation de la mémoire est donc la *taille* de cette hiérarchie. De plus, la consommation d'un accès à la mémoire est liée au niveau contenant la donnée, plus la donnée réside dans un niveau de hiérarchie éloigné du processeur plus l'accès sera lent et consommateur. L'amélioration de la *localité* temporelle des données est donc un premier facteur d'optimisation de la consommation. La localité temporelle est définie par la distance en temps séparant la production d'un calcul de l'utilisation de son résultat en un autre point du programme. Plus cette distance est petite, meilleure est la localité temporelle.

La taille de la mémoire et la localité des accès sont difficilement mesurables tant que l'organisation du système n'est pas encore définie. Pourtant c'est bien à ce niveau de conception que l'on doit s'assurer de ne pas aller dans une mauvaise direction. La conception de système nécessite donc de nouvelles techniques d'optimisation de la mémoire lors de la conception conjointe du matériel et du logiciel.

## Combiner les architectures et la compilation

La conception de systèmes doit combiner les deux méthodologies : construction d'une architecture pour les calculs, les communications et la mémoire et compilation sur cette architecture (la compilation pouvant être entendue ici comme la synthèse d'un matériel dédié). Les outils d'aide à la conception et les méthodologies qui se mettent en place tendent vers cette ambitieuse direction : le *Codesign* ou conception conjointe matérielle/logicielle. En effet, les progrès en intégration ont permis ainsi de mettre en place des systèmes complets tenant sur une seule puce de silicium. La tendance actuelle des développements propose donc d'intégrer l'ensemble d'un système (application, système temps réel, processeur(s), bus de communications et hiérarchies mémoires) sur une seule puce. Ces systèmes sont appelés des *System on Chip* (*SoC*). Les environnements de conception assistée par ordinateur proposent aujourd'hui de combiner des fonctions de conception, d'intégration, de simulation et d'estimation de composants matériels et logiciels. Les décisions de conception restent cependant entièrement manuelles et tout repose sur le savoir-faire du concepteur.

Le travail de cette thèse se tourne vers les optimisations d'une description d'application multimédia pour en optimiser la mémoire dans le but de minimiser la consommation et permettre de guider le partitionnement entre matériel et logiciel, lors des premières étapes de la conception. Plus précisément, la thèse se découpera ainsi :

**Le chapitre 1** présente la synthèse de circuit et son évolution logique vers la synthèse de systèmes. Nous présentons ensuite les différentes solutions architecturales disponibles pour la construction des SoC. Cette présentation met en avant les principales sources de consommation représentées par les unités de calcul, les communications et la mémoire. Nous y présentons en parallèle les techniques de minimisation de la consommation autres que celles développées dans cette thèse et qui leur sont complémentaires.

**La chapitre 2** présente les transformations de programme d'une manière générale et plus particulièrement celles disponibles pour la gestion des hiérarchies de mémoires. Une grande part des

transformations concerne celles utiles pour les mémoires caches matérielles dans un contexte où l'architecture cible est définie. Nous y présentons également les transformations pour les mémoires caches logicielles ainsi que la méthodologie DTSE définie à IMEC pour laquelle une étape de transformation a été développée pendant cette thèse. Les transformations de ce chapitre concernent des architectures de système n'utilisant qu'une seule unité de calcul.

**Le chapitre 3** présente les transformations utilisables *avant* l'étape de partitionnement de la spécification. Ces transformations sont effectuées sans connaître a priori les particularités de l'architecture cible ni la répartition des fonctionnalités sur les différentes unités de calcul. Parmi ces transformations, nous en présentons deux nouvelles qui ont été développées dans le cadre de cette thèse : la fusion de boucle pour minimiser le nombre de tableaux temporaires et l'alignement de boucles pour la minimisation des variables temporaires inter-itérations.

**Le chapitre 4** concerne l'application des méthodes formelles présentées dans les chapitres précédents dans un cadre pratique. Pour définir plus précisément les critères de combinaison des méthodes et pour percevoir une méthodologie d'aide à la transformations de boucles, il faut réaliser des expérimentations de code d'applications réelles. Une collaboration avec les équipes de recherche de la société Cadence nous a permis d'utiliser le logiciel VCC et d'effectuer des tests d'intégration dans la méthodologie proposée par cet outil. Nous présentons dans ce chapitre l'intégration des méthodes présentées dans le cadre de la conception de système.

**Le chapitre 5** conclut cette thèse par des perspectives sur les transformations de programme et plus généralement sur les outils et méthodes utilisées dans le domaine de la conception conjointe de système.

# Chapitre 1

# Codesign et architectures

Cette thèse ayant pour but l'optimisation de la mémoire et de la consommation pour la synthèse de systèmes embarqués, nous commencerons par présenter ce que sont la synthèse d'architectures et la synthèse de systèmes ainsi que les choix *architecturaux* se présentant à un concepteur. Ces présentations seront générales et permettront de motiver les chapitres suivants.

## 1.1 Évolution de la conception de circuits intégrés

Les technologies et les techniques utilisées pour la fabrication des circuits à très haute intégration (*Very Large Scale Integration – VLSI*) proposent maintenant de synthétiser des circuits extrêmement complexes. Des densités de plusieurs millions de portes pour les circuits logiques et de plusieurs dizaines de millions pour les circuits réguliers, comme les mémoires, sont devenues courantes.

La synthèse de circuit est un processus de transformation d'une description *comportementale* vers une description *structurelle* permettant de générer la description *physique* d'un circuit. Cette transformation est comparable à la compilation de langages de haut niveau comme C ou Pascal en leur équivalent sous forme de programme en langage assembleur. La synthèse, aussi appelée raffinement de description (*Design Refinement*), s'effectue en une succession d'étapes représentées de manière simplifiée à la figure 1.1. Chacune de ces étapes de transformation est chargée de produire des informations, ainsi que des *contraintes*, pour guider la synthèse et générer un circuit répondant aux *critères* d'optimisation fixés par le concepteur.

On définit la synthèse de haut niveau (*High Level Synthesis–HLS*) comme l'étape de transformation permettant de passer d'un niveau de description comportementale, écrite dans un langage de haut niveau comme C, C++, SDL ou le modèle comportemental de VHDL, vers une description structurelle en transferts de registres (*Register Tranfer Level – RTL*). Le niveau RTL utilise des langages de description de matériels (*Hardware Description Languages – HDL*) comme VHDL, Verilog ou HardwareC. Ces langages de description diffèrent des langages de description comportementale car ils intègrent la capacité d'exprimer le temps, par une ou plusieurs horloges, et la notion de concurrence matérielle.

Le flot de données au niveau RTL est spécifié en utilisant trois types de composants : des unités fonctionnelles (unités arithmétiques et logiques, multiplieurs), des unités de stockages (registres, mé-

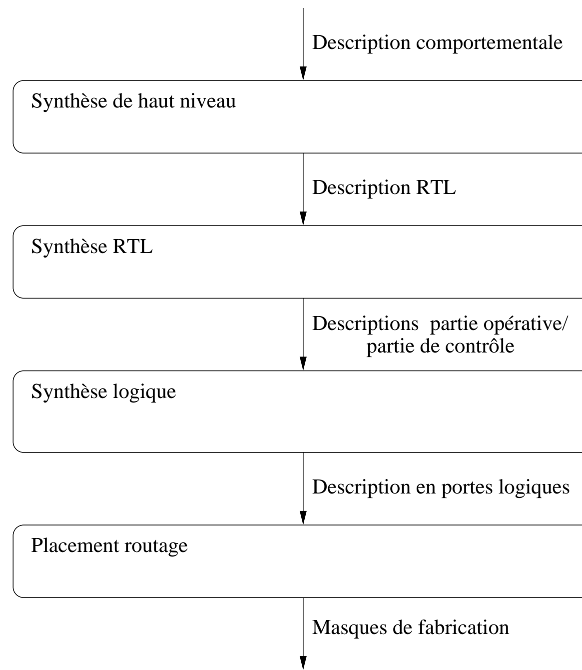



FIG. 1.1 – Synthèse de haut niveau : flot de conception simplifié

moires) et des interconnexions (bus et multiplexeurs). Les trois principaux problèmes de la synthèse de haut niveau sont *l'ordonnancement*, *l'allocation* de ressources et *l'affectation* de ces ressources. L'ordonnancement assigne à chaque opération de la description comportementale un pas de contrôle pendant lequel elle devra être effectuée. Un pas de contrôle correspond en général à un cycle d'horloge du système, unité basique de temps pour les systèmes numériques synchrones. L'allocation choisit des unités fonctionnelles et des éléments de stockage dans une bibliothèque de composants précaractérisés correspondant à la technologie dont dispose le fondeur. Ces bibliothèques peuvent proposer plusieurs choix pour chaque type d'opérateurs (différents types de multiplieurs, arithmétique classique ou utilisant une représentation redondante ...). Enfin, l'affectation permet de faire le lien entre opérations et unités fonctionnelles, variables et éléments de stockage, transferts de données et liaisons câblées ou bus de communication.

Ces trois étapes sont fortement interdépendantes. Un ordonnancement optimal des opérations en pas de contrôle ne peut être fait sans prendre en compte le coût et les performances des unités fonctionnelles choisies lors de l'allocation. De la même manière une allocation efficace ne peut être effectuée sans prendre en compte le parallélisme et la concurrence entre les opérations introduits par l'ordonnancement. De plus, les contraintes de coût, de surface, de performances et de consommation nécessitent la prise en compte de nombreux compromis. Ainsi, une architecture proposant la plus petite surface sera constituée du minimum de composants les plus lents nécessitant un grand nombre de pas de contrôle pour effectuer les opérations. D'un autre côté la recherche de performance nécessite l'allocation de nombreuses unités fonctionnelles rapides permettant de maximiser l'exécution parallèle des calculs, cette utilisation de nombreux composants rapides impliquant un surcoût en surface du circuit et en consommation. Ces étapes sont la plupart du temps NP-Complètes ou NP-Difficiles et sont parfois effectuées à l'aide de programmes linéaires en nombres entiers (PLNE) ou bien d'heuristiques permettant d'avoir une solution proche de l'optimal en un temps raisonnable.

La description RTL est à son tour transformée en une description de plus bas niveau en deux parties. La synthèse des parties opératives (PO) est effectuée en utilisant des bibliothèques de composants standards (registres, additionneurs, multiplieurs ou bien unités arithmétiques et logiques complètes) et la partie de contrôle (PC), permettant de séquencer les opérations, est générée sous forme de logique combinatoire à partir d'une machine à états finis ou d'un graphe de contrôle. Cette description de l'architecture est aussi appelée description PC/PO (partie contrôle/partie opérative).

La synthèse logique est l'étape suivante du flot de synthèse. On définit ici les éléments de la partie opérative ainsi que la logique de contrôle sous forme d'un réseau de portes booléennes. Les descriptions des parties opératives sont définies dans des bibliothèques proposées par les fondeurs selon la technologie de fabrication qui sera utilisée. La partie combinatoire est générée en utilisant des algorithmes de minimisation de fonctions booléennes (comme ceux utilisant des arbres de décision binaires).

Le placement/routage est la dernière étape de synthèse avant la fabrication des masques servant à la fabrication physique du circuit. On définit dans cette étape la place de chaque porte logique sur le circuit ainsi que le cheminement des fils connectant les portes entre elles. Les masques produits à partir de ce placement/routage servent à construire les transistors ainsi que leurs connexions par des superpositions de couches de métal et de semi-conducteurs.

La synthèse de haut niveau a été un sujet de recherche extrêmement actif durant les 15 dernières années et demeure encore un sujet où de nombreux problèmes restent ouverts.

Les avantages obtenus en augmentant le degré d'abstraction des descriptions sont nombreux : les temps de conception sont plus courts, les circuits contiennent moins d'erreurs, on peut explorer un espace de solutions plus vaste et la technologie devient accessible à un plus grand nombre de personnes. Cependant, la synthèse de haut niveau pose des problèmes que n'avaient pas rencontrés les chercheurs et industriels lors de la mise en place des autres niveaux de synthèse. Les raisons de ces difficultés sont multiples. Le niveau de placement/routage repose sur des fondations formelles provenant de la théorie des graphes et des algorithmes sur les graphes. La synthèse logique utilisant, quant à elle, le formalisme des algèbres de Boole. Étendre la synthèse à des niveaux d'abstraction supérieurs pose un problème certain à cause de l'absence d'un formalisme adapté. Mais plus important encore, cette extension accroît de manière gigantesque l'espace de recherche à explorer dans la quête de la solution optimale. Une approche générale permettant d'explorer toutes les solutions possibles ne pourra sans doute jamais exister [Weh95]. L'utilisation de la synthèse de haut niveau a été en partie rejetée par les concepteurs à cause du manque de maîtrise de l'architecture finale, des difficultés pour mettre en place des communications efficaces entre les sous systèmes, et du manque de maîtrise des notions de temps et de concurrence.

Les outils de Conception Assistée par Ordinateur pour la synthèse (CAO) disponibles actuellement ainsi que les technologies de synthèse VLSI ont atteint un niveau de maturité suffisant pour que la maîtrise technologique ne représente plus, en soi, un avantage décisif pour les industriels. Le temps de mise sur le marché (*time to market*) est maintenant habituellement vu comme critère aussi important dans la conception, sinon plus, que la surface, la vitesse ou la consommation d'un circuit. Cependant, il reste tout de même impraticable de concevoir ex nihilo un circuit de plusieurs dizaines de millions de portes, le coût de conception d'un circuit spécifique croissant exponentiellement avec la taille de ce circuit. L'augmentation très rapide de la complexité des circuits ainsi que la réduction des temps de conception, présentée dans le tableau 1.1, nécessite donc de la part des méthodes et des outils de conception une amélioration significative de l'automatisation et l'utilisation de niveaux d'abstraction plus élevés encore que la synthèse de haut niveau qui atteint aujourd'hui ses limites.

| | 1997 | 1998 | 1999 | 2002 |
|---|---|---|---|---|
| Technologie de fabrication ($\mu$m) | 0,35 | 0,25 | 0,18 | 0,13 |
| Coût d'une usine de fabrication (milliards de $) | 1,5–2 | 2–3 | 3–4 | >4 |
| Temps de conception d'un produit (mois) | 18–12 | 12–10 | 10–8 | 8–6 |
| Temps de conception d'un produit dérivé (mois) | 8–6 | 6–4 | 4–2 | 3–2 |
| Complexité des puces (portes logiques) | 200–500K | 1–2M | 4–6M | 10–25M |

TAB. 1.1 – Évolution des procédés de fabrication et de la complexité des circuits [CCH+99]

On ne considère plus la synthèse de haut niveau comme une étape généraliste mais comme une synthèse appliquée à *un type d'architecture*.

Les fondeurs espèrent pouvoir construire des circuits dépassant le milliard de transistors sur une même puce d'ici 2010 [Ham99] mais aucun outil ni concepteur n'est, à ce jour, capable d'utiliser une telle ressource de manière efficace. Le passage de la conception de circuits au niveau des transistors à la conception au niveau des portes logiques puis l'emploi, dans le début des années 1980, de bibliothèques de composants, ont permis de faire de grandes avancées en synthèse. De la même manière, le niveau d'abstraction requis actuellement passe par la réutilisation de composants hétérogènes de grande taille (*IP reuse*). Ces composants permettent de concevoir des SoC aussi bien des circuits spécialisés que des modules logiciels tournant sur des processeurs.

## 1.2 Conception de systèmes sur silicium

Un SoC [CCH+99] est défini dans les ouvrages comme un circuit intégré complexe comprenant tous les éléments d'un *produit fini complet* sur *une même puce de silicium*. Il intègre en général un processeur programmable, de la mémoire, des interfaces pour des périphériques et des circuits spécialisés pour accélérer le traitement de certaines fonctions. La présence de processeurs programmables ainsi que de circuits spécifiques implique un lien extrêmement fort entre la partie logicielle et le matériel dans la conception conjointe (*Codesign*) des *SoC* [RB95]. La présence de périphériques nécessite également la prise en compte d'éléments analogiques et pourra inclure dans un futur proche des systèmes mécaniques opto ou micro-électroniques (*opto/micro-electronic mechanical components – O/MEMS*).

Ces systèmes sont le plus souvent conçus et réalisés pour répondre à un besoin applicatif spécifique. On a affaire alors à un système prévu pour n'effectuer qu'une seule tâche ou un ensemble fixe et bien ordonné de tâches. Ces particularités permettent de dimensionner précisément une conception en fonction de la charge de travail que le système aura à effectuer.

Les méthodologies de conception pour ce type de design convergent aujourd'hui vers un flot, résumé sur la figure 1.2, comportant trois phases principales.

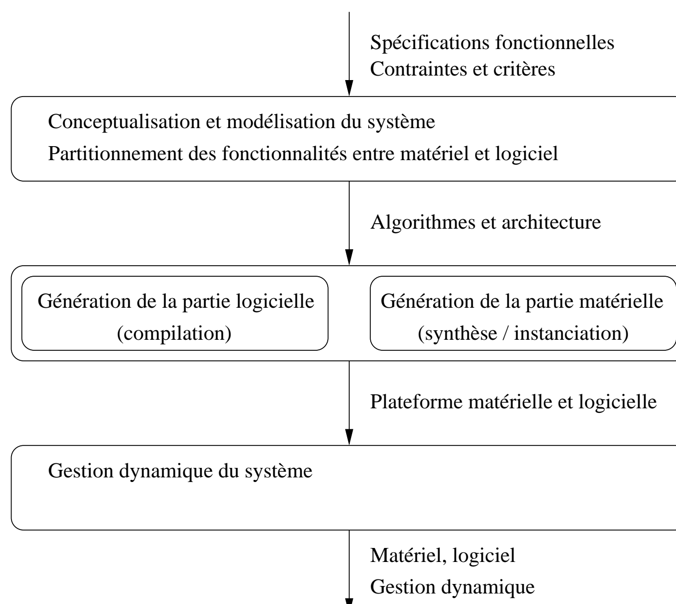



FIG. 1.2 – Principales phases de conception conjointe d'un système

La première étape permet, à partir d'un ensemble de spécifications et de contraintes, de mettre en place un modèle d'architecture ainsi que le découpage des fonctionnalités entre matériel et logiciel. À ce stade, le concepteur n'a qu'une vision abstraite du système, représenté par ses fonctionnalités. Le choix des algorithmes utilisés ainsi que celui du support matériel ou logiciel pour accomplir une fonctionnalité ont une très grande influence sur les performances d'un système. De même, les choix comme la largeur de mots, l'utilisation de calculs en virgule fixe ou flottante ou bien les compromis sur la précision des calculs (par exemple ceux concernant la qualité des images dans les applications multimédia) sont effectués durant cette phase. Les choix effectués ici ne concernent toutefois pas les problèmes et détails d'implémentation qui sont laissés à l'étape suivante.

La deuxième phase est la réalisation des parties matérielles et logicielles. Les besoins matériels définis dans la phase précédente sont ici créés : en utilisant la synthèse de haut niveau pour avoir des composants spécifiques (*Application Specific Integrated Circuit – ASIC*) ou en instanciant des composants choisis dans une bibliothèque. La partie logicielle est pour sa part compilée selon le support programmable qui lui a été attribué : cœur de processeur, de DSP ou processeur spécifique (*Application Specific Integrayed Processor – ASIP*). Un cœur de processeur est défini par son jeu d'instructions et son implémentation logique afin de pouvoir l'intégrer et l'adapter à un SoC.

Enfin, une fois le système réalisé, il reste à mettre en place la gestion de son fonctionnement. Cette gestion comprend le contrôle des ressources de la plateforme matérielle mais aussi les interactions avec le monde extérieur par l'intermédiaire des périphériques. Cela nécessite un ordonnanceur minimal de tâches, pouvant aller jusqu'à l'inclusion d'un système d'exploitation temps-réel (*Real Time Operating System – RTOS*). La présence d'un système d'exploitation doit, bien sûr, être prévue dès la première phase de conception mais l'implémentation de celui-ci ainsi que la gestion des ressources ne peuvent être effectuées avant de connaître avec précision la plateforme matérielle [BBPM99].

Une méthodologie comme celle présentée figure 1.2, fait apparaître très tôt les problèmes rencontrés pour concevoir un système et permet de construire une architecture globale adaptée aux

contraintes et critères du concepteur. En effet, des retours rapides sur les décisions, obtenus par simulation ou par estimations de haut niveau, sont utilisés pour évaluer l'influence des choix pris à chaque étape. Nous nous focalisons ici plus particulièrement sur la première étape de synthèse dont le détail est représenté sur la figure 1.3.

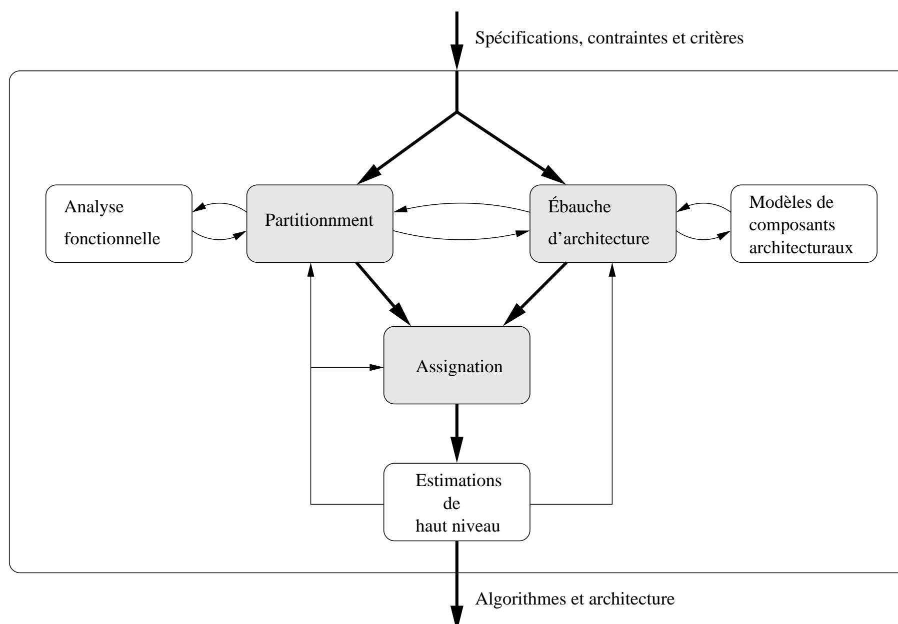



FIG. 1.3 – Conception conjointe matériel/logiciel : partitionnement et assignation

Une fois la spécification fonctionnelle terminée et vérifiée, par simulation ou validation formelle, une architecture doit être définie au mieux pour la plateforme finale. Les solutions architecturales sont très nombreuses et les composants utilisés peuvent être conçus spécifiquement, réutilisés à partir d'une autre réalisation ou provenir de fournisseurs externes. Les composants architecturaux sont définis ici par des modèles abstraits (processeur, ASIC, mémoire, mémoire cache, bus de communication ...) et non des implémentations spécifiques. La définition de cette architecture ne peut se faire que conjointement avec le partitionnement de la spécification fonctionnelle. Une analyse fonctionnelle est utilisée afin de cerner les parties critiques, qui seront de préférence assignées à un circuit spécifique, et les autres parties pour lesquelles la souplesse d'utilisation du logiciel pourra être mise en avant.

Cette interaction entre partitionnement et assignation permet d'explorer les différentes solutions qui peuvent être mises en œuvre et ainsi déterminer une plateforme la mieux dimensionnée possible et respectant toutes les contraintes. Les problèmes d'optimisation des critères doivent impérativement être pris en compte lors de cette étape car des décisions importantes sur l'*organisation* du système y sont prises. Ne vouloir considérer les phases d'optimisation qu'ultérieurement est généralement inefficace et clairement insuffisant. D'un autre côté, il ne faut pas non plus se limiter à optimiser les premières phases de la conception car les résultats en surface, vitesse et consommation dépendront *in fine* du soin apporté aux détails et à la structure exacte de l'implémentation.

Une des motivations de cette thèse étant la minimisation de la consommation électrique générée par la mémoire en tant que ressource architecturale de haut niveau, nous avons choisi de présenter dans le prochain chapitre les principaux types de composants utilisés dans les SoC. Cette présenta-

tion succincte mettra en avant les principales sources de consommation et présentera les solutions existantes au niveau de l'implémentation permettant de réduire cette consommation.

## 1.3 Architectures des *SoC* et consommation

Une architecture matérielle est souvent définie dans les ouvrages d'informatique comme ayant une unique unité de calcul principale [PH94]. Cette unité de calcul propose une interface d'utilisation (*Instruction Set Architecture – ISA*) fournissant une abstraction claire et bien définie entre matériel et logiciel. Cette unicité d'interface est également valable dans le cas de machines multi-processeurs car ces machines sont souvent construites à l'aide d'unités de calcul identiques.

Les systèmes que nous considérons ici sont, pour leur part, perçus comme des architectures parallèles utilisant plusieurs unités de calcul *hétérogènes* (avec des jeux d'instruction différents) ou des circuits spécialisés. Une grande part de la conception est donc dédiée à l'*organisation des systèmes*. On emploie souvent le terme de *macro-architecture* pour se référer à ce type de système et micro-architecture pour les machines à jeu d'instruction unique.

L'association japonaise des industries électroniques (*Electronic Industries Association of Japan – EIAJ*) a défini un planning pour les technologies d'automatisation de conceptions électroniques afin de concevoir un « cyber-giga-chip » aux environs de 2002 [EIA98]. Ce système doit incorporer une mémoire dynamique (DRAM), une mémoires flash, des cœurs de processeurs génériques, des cœurs de processeurs pour le traitement du signal (DSP), du matériel de contrôle pour des protocoles et pour le traitement du signal, des blocs analogiques, des unités matérielles spécialisées (ASIC) et des bus de communications embarqués. Cette description de système donne une bonne illustration de la complexité à laquelle devront faire face les concepteurs des futurs SoC [CCH+99].

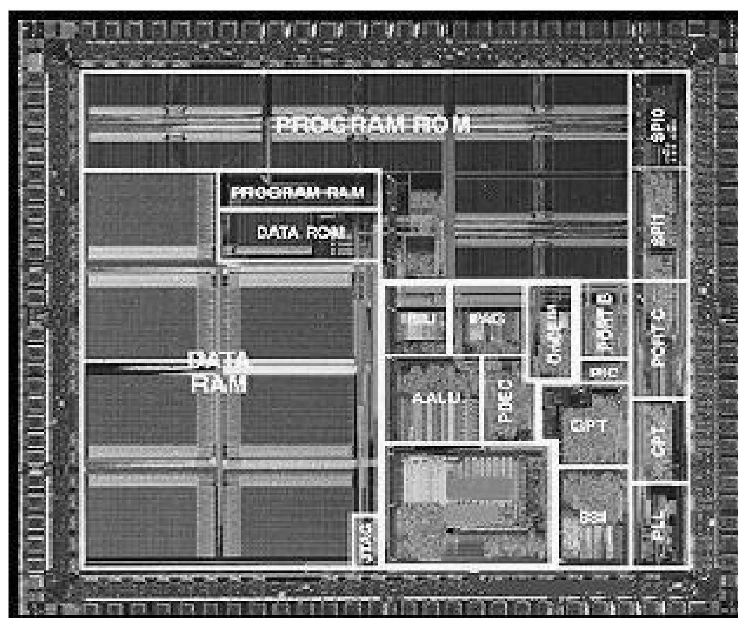



FIG. 1.4 – Exemple d'architecture pour un SoC

Le compromis entre consommation et vitesse est devenu très important dans les conceptions récentes. Plusieurs facteurs ont révélé la portée centrale de cet arbitrage : les systèmes sont pour une très grande part portables et alimentés par une batterie limitée en taille et en poids ; les architectures haute-performance posent de plus en plus de problèmes de dissipation thermique ; et il est tout simplement devenu *nécessaire* de contrôler la consommation des équipements électroniques comme l'a démontré le problème de rationnement électrique imposé dans la *Silicon Valley* l'hiver dernier suite à une demande de consommation bien supérieure à celle que pouvait fournir le réseau électrique local !

Les prochaines sections présentent les trois principales sources de consommation électrique dans un système électronique. Ces sources sont réparties entre les unités de *calcul*, les *communications* et la *mémoire*. Nous abordons également les techniques couramment utilisées pour réduire la consommation de ces composants. Ce panorama ne prétend pas être exhaustif car la bibliographie disponible dans ce domaine se compte en milliers d'articles. Nous ne présentons pas non plus les optimisations disponibles au niveau physique car elles dépassent le cadre de cette thèse.

### 1.3.1 Les unités de calcul

Les unités de calcul, comme toute logique numérique, sont conçues à partir de transistors. Les transistors peuvent être construits à partir de jonctions PN (deux types différents de semi-conducteurs) ou bien utiliser une technique à effet de champs (*Field Effect Transistor – FET*). C'est ce type de transistor qui est maintenant utilisé pour les circuits VLSI avec une technologie de cellules CMOS (*Complementary Metal–Oxyde–Semiconductor*) comportant deux transistors MOS de types complémentaires. Cette technologie est choisie pour sa faible consommation et ses possibilités importantes d'intégration. Le schéma d'un transistor à effet de champs de type MOS est représenté figure 1.5.

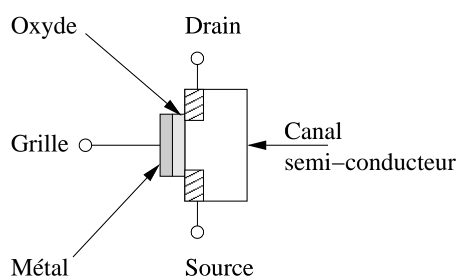



FIG. 1.5 – Transistor à effet de champs MOS

Les transistors permettent d'avoir deux types de fonctionnement : l'application d'une tension de commande $e$ sur la grille empêche toute circulation du courant entre le drain et la source, on est alors en régime *bloqué* ; lorsque la tension est nulle à l'entrée $e$ le régime est dit *saturé* et le courant est libre de circuler.

Ces régimes de fonctionnement saturé et bloqué font des transistors des commutateurs statiques qui sont à la base de l'élaboration de toutes les portes logiques. La consommation électrique des cellules CMOS se divise en deux catégories : la consommation électrique statique (dissipation thermique causée par le passage du courant dû aux imperfections des composants) et la consommation électrique dynamique (activité de transition des cellules). La consommation statique est négligée ici

car elle dépend des caractéristiques physiques des transistors. Les entrées commandées en tension ne consomment aucun courant et seules les commutations de niveau donnent lieu à une consommation de courant. On retrouve ces propriétés de consommation au niveau des circuits complets [BB96] pour lesquels les principales méthodes d'optimisation se concentrent sur la tension d'alimentation $V_{dd}$ (qui est un des facteurs les plus importants car la puissance dissipée durant les transitions varie selon $V_{dd}^2$), la limitation de la fréquence d'horloge et le nombre moyen de changements d'état des transistors.

Les unités de calcul d'un système embarqué peuvent être prises en compte selon différents niveaux de complexité. Le niveau le plus haut est celui des cœurs de processeurs génériques. L'utilisation de cœurs est populaire car elle permet de diminuer le temps de conception d'un système en laissant les problèmes d'optimisation au niveau de la compilation logicielle. Les cœurs de processeurs RISC ainsi que ceux de processeurs de traitements du signal (*Digital Signal Processors–DSP*) sont maintenant d'utilisation courante. Cependant, ces cœurs restent de trop grands consommateurs d'énergie comparés à des solutions utilisant du matériel spécialisé. Un processeur peut, par exemple, très facilement faire tourner un programme de transformé de Fourier ou un filtre du 5$^{\text{ème}}$ ordre en exécutant une suite d'additions et de multiplications. D'un autre côté, une implémentation matérielle spécifique peut être effectuée à l'aide de deux additionneurs, d'un multiplieur et de quelques registres dont le fonctionnement sera directement câblé. La différence de consommation entre ces deux architectures est de plusieurs ordres de grandeur en faveur des circuits dédiés.

Dans cette partie nous présentons les différentes solutions d'optimisation existantes pour intégrer les unités de calcul dans les systèmes embarqués. Ces solutions sont présentées selon le type de ressource utilisé : les cœurs de processeurs, les processeurs dédiés (ASIP) et les circuits spécialisés (ASIC).

### Processeurs génériques

Les processeurs génériques des machines de bureau et serveurs sont partagés principalement entre les architectures x86 d'Intel et les compétiteurs représentés par les architectures PowerPC d'IBM et Motorola, UltraSparc de Sun, PA-RISC de Hewlett-Packard, Alpha de DEC ou MIPS de MIPS Technologies. Ces architectures, conçues au départ pour les machines de bureau, sont construites dans la quête de performances et non pour la consommation. On peut prendre à titre d'exemple (encore que ce ne soit pas forcément un bon exemple pour les performances) le cas du Pentium 4 d'Intel, en technologie 0,18$\mu$m, qui dissipe 130 watts en régime de fonctionnement normal ! Comme le faisait remarquer David Patterson [Pat98] en 1998 :

> "Intel specialized in designing microprocessors for the desktop PC, which in five years may no longer be the most important type of computer. Its successor may be a personal mobile computer that integrates the personal computer with a cellular phone, digital camera, and video game player... Such devices require **low-cost, energy-efficient** microprocessors, and Intel is far from the leader in this area."

Les processeurs les plus courants dans le domaine de l'embarqué sont des architectures RISC (*Reduced Instruction Set Computer*) comme les processeurs ARM et Thumb d'Advanced RISC Machine, Hitachi SuperH ou MIPS16 mais de plus en plus de modèles 32 bits sont disponibles dans une version « embarquable » comme le montre l'apparition d'architectures x86 d'Intel, Sparc de Sun ou Coldfire de Motorola prévues pour les systèmes embarqués.

Les cœurs de processeurs sont de plus en plus utilisés car ils offrent une grande souplesse d'utilisation. En effet, pour la plupart, ces systèmes doivent pouvoir être reprogrammés ou mis à jour une fois commercialisés. Cette souplesse s'obtient au détriment des critères de performances et de consommation pour les raisons suivantes :

- chaque instruction doit être chargée puis décodée avant d'être exécutée.
- les opérations sont limitées par le jeu d'instruction. Une opération complexe doit être découpée par le compilateur en opérations de base.
- le fonctionnement est dans la plupart des cas séquentiel et il est difficile d'exploiter la totalité du parallélisme présent dans les applications.

Les solutions pour adapter les processeurs génériques au monde de l'embarqué sont nombreuses. La plus courante reste l'abaissement de la tension d'alimentation. Cette technique nécessite d'être prise en compte dès le départ de la conception du processeur comme le montre l'exemple de l'Intel Pentium 100Mhz, premier processeur Intel utilisant cette technique, dont la micro-architecture (les chemins de calcul critiques) avait dû être entièrement revue. En effet, une tension d'alimentation plus faible augmente le délai de transition des portes et ralentit le fonctionnement du processeur.

Cette diminution de la tension d'alimentation peut être effectuée de manière dynamique dans les processeurs à voltage variable par un dispositif matériel permettant l'ajustement du voltage et de la fréquence d'horloge. Cette gestion est faite par le niveau « gestion du système » selon la charge de calcul des tâches à ordonnancer.

Les unités fonctionnelles ou les différentes parties d'un processeur ne sont en général pas toutes utilisées en même temps. Une autre technique consiste donc à annuler temporairement le signal d'horloge (*clock gating*) dans certaines parties. Un module qui n'est plus séquencé par le signal d'horloge est à l'état de repos et ne consomme plus de courant. Cette coupure peut apparaître dès le décodage des instructions car on sait alors quelles unités seront actives dans les prochains cycles mais cela nécessite une attention particulière lors de la conception de la micro-architecture. En effet, il faut s'assurer que les unités fonctionnelles seront réactivées à temps pour les prochaines opérations et que la présence de portes logiques supplémentaires sur le signal d'horloge ne génère pas une dérive d'horloge trop importante (*clock skew*). De plus, la remise en tension de larges portions d'un circuit peut générer des pics de consommation devant être pris en compte lors de la conception des grilles d'alimentation [TSR+98].

Les processeurs haute-performance utilisent des techniques très consommatrices, comme les techniques d'exécution spéculatives ou l'exécution des instructions dans le désordre. Les processeurs performants pour le critère de consommation utilisent dans la majorité des cas des architectures plus simples et plus régulières car le gain de performances apporté par ces techniques ne compense pas le surcoût de consommation qu'elles impliquent.

Les architectures CISC (*Complex Instruction Set Computer*) offrent, par définition, une densité de code bien meilleure que les architectures RISC. Une instruction CISC peut commander des suites d'opérations plus complexes que celles permises par les instructions RISC. Cependant, les architectures CISC nécessitent une logique de décodage bien plus coûteuse car les instructions sont de longueur variable et utilisent des constructions variées. Les instructions RISC sont de longueur fixe et leur structure est régulière, mais elles nécessitent une bande passante vers la mémoire plus importante car elles utilisent plus de mots d'instructions pour une même fonctionnalité. La densité du code d'un programme influence la consommation car moins un code est dense, plus il nécessite de mots d'instructions et donc d'activité de gestion et de transferts depuis la mémoire.

Des améliorations ont été proposées pour essayer de réduire le coût mémoire des programmes. La méthode retenue par le fabricant Hitachi est de réduire la longueur des mots d'instructions à 16 bits, au lieu de 32. Cette approche permet de simplifier le décodage sauf dans le cas des instructions contenant une valeur immédiate ou une adresse mémoire. Enfin, la compression de code [CAP99] permet aussi d'avoir des gains importants en bande passante. Cela nécessite un circuit spécialisé permettant de faire la décompression à la volée des instructions. Le processeur ARM Thumb intègre ce mécanisme en permettant d'utiliser un jeu d'instructions sur 16 bits décodé à la volée, par une logique câblée dans l'unité de décodage, en son équivalent sur 32 bits.

L'augmentation de la densité du code et le réduction de la consommation peuvent aussi être effectuées par l'ajout d'instructions spéciales. De plus en plus de processeurs intègrent maintenant des instructions que l'on ne pouvait trouver auparavant que dans les DSP. Parmi ces instructions on peut noter l'opération de multiplication-addition (*Multiply Accumulate – MAC*) permettant d'enchaîner les deux opérations dans le même cycle d'horloge, les instructions spécialisées pour les applications dominées par les données comme celles permettant d'effectuer des opérations parallèles SIMD (*Simple Instruction Multiple Data*) sur des mots (*Subword parallel instructions*). Les architectures spécialisées dans le traitement du signal (*Digital Signal Processor – DSP*) poussent ce principe à l'extrême en proposant des instructions très complexes. Le problème principal de ces architectures est qu'il est très difficile de concevoir des compilateurs permettant de tirer pleinement parti de ces instructions. Dans la majorité des cas la programmation doit être effectuée à la main, en assembleur, par un expert.

Les cœurs peuvent aussi être privés de certaines fonctions selon les besoins et des fonctionnalités non utilisées peuvent être enlevées du design. On peut par exemple priver un cœur de son unité arithmétique flottante si celle-ci n'est pas utilisée par le système. Cela permet de réduire la surface et la consommation générées par le cœur.

La conception de processeurs performants représente un défi pour la recherche dans les thèmes liés à la consommation. La première priorité reste toujours la baisse de la tension d'alimentation mais de nouveaux types de circuits ainsi que des méthodologies adaptées ont besoin d'être développés pour traiter les problèmes de distribution d'alimentation électrique et les dérives d'horloge introduits par les mises hors-tension des composants. D'autre part, les micro-architectures actuelles se tournent vers un plus grand parallélisme au niveau des instructions. Pour l'instant ces conceptions sont uniquement pensées pour la performance, mais la consommation doit être mise au premier plan si l'on veut pouvoir mettre en place une interface entre matériel et logiciels proposant un bon compromis pour les systèmes embarqués.

#### Processeurs spécialisés

Les processeurs spécialisés (*Application Specific Integrated Processor – ASIP*) sont un compromis entre les processeurs génériques et les circuits spécialisés. Ils permettent d'avoir la souplesse de programmation des processeurs tout en étant construits selon les particularités d'un type d'applications. Les recherches concernant la conception de processeurs spécialisés sont encore peu développées et restent très en retrait par rapport aux techniques disponibles pour les processeurs génériques ou les circuits spécialisés [BM00].

La construction de ces architectures repose sur une utilisation intensive de méthodes de profilage pour adapter au mieux les architectures aux applications cibles.

La réduction du jeu d'instructions (*Instruction Subsetting*) permet de réduire le nombre d'instructions définies par l'interface de programmation. L'application est compilée en utilisant le jeu d'instructions complet, le code est ensuite profilé et les instructions qui n'apparaissent jamais ou qui peuvent être remplacées par d'autres, avec une perte de performance minime, sont retirées du jeu d'instructions. Cette réduction simplifie la logique de décodage et diminue la longueur des mots d'instruction. On réduit donc la consommation et la bande passante nécessaire pour le chargement des instructions.

Les processeurs superscalaires (ayant plusieurs unités fonctionnelles pouvant fonctionner en parallèle) peuvent être eux aussi adaptés en fonction de l'application. On peut modifier ici le type et le nombre d'unités fonctionnelles utilisées ainsi que la taille du banc de registre [CMST00]. La méthode est itérative et nécessite de simuler l'exécution de l'application pour chaque variation de paramètre. Toutefois, le jeu d'instructions ne change pas et l'application n'est compilée qu'une seule fois et utilisée pour toutes les simulations.

L'utilisation de machines superscalaires à programmation explicite (*Very Large Instruction Word – VLIW*) peut elle aussi bénéficier de cette technique. Son utilisation dans ce cas est plus délicate car elle impose de recompiler l'application pour chaque simulation. On doit donc modifier le compilateur utilisé en fonction des changements de paramètres de l'architecture. Le compilateur PICO, développé chez Hewlett-Packard permet de générer une architecture VLIW et le compilateur associé à partir d'une spécification des fonctionnalités [ARK99].

La conception de processeurs spécialisés requiert des outils complexes d'estimations et de simulations encore très peu développés. Ceci rend leur optimisation délicate et l'utilisation de processeurs spécialisés est assez récente. Ce thème n'a, pour l'instant, bénéficié que de peu de recherches par rapport aux processeurs génériques ou aux circuits spécialisés.

#### Circuits spécialisés

Les circuits spécialisés (*Application Specific Integrated Circuit – ASIC*) sont les unités de calcul les plus performantes et les plus économes en énergie car elles utilisent uniquement le minimum de ressources nécessaire et tirent un profit maximal du parallélisme disponible. Leur usage doit cependant être justifié dans les systèmes car elles sont très chères à fabriquer, ne sont pas modifiables en cours d'évolution des produits et ne sont généralement pas réutilisables dans d'autres systèmes.

Ce type d'architecture s'affranchit des problèmes de chargement et de décodage des instructions rencontrés par les processeurs programmables car son fonctionnement est câblé. Cette connaissance du programme permet une utilisation très efficace de techniques bien établies comme le parallélisme, le *retiming* [LS91] ou les pipelines [PH94]. Par exemple, la génération d'un circuit parallèle pipeliné ayant un débit de sortie deux fois supérieur au débit imposé par les contraintes permet de réduire la tension d'alimentation du circuit, ce qui a pour effet de le ralentir, jusqu'à ce que le circuit produise le débit de sortie imposé. Les techniques de pipeline sont limitées car, dans les pipelines très profonds (de 15 à 20 étapes), une part substantielle du temps est prise par le temps de réponse des registres. On ne peut pas augmenter de façon arbitraire la profondeur des pipelines car les registres occuperaient alors une surface trop importante et poseraient des problèmes de surcharge électrique sur le signal d'horloge. Le parallélisme est lui aussi limité par l'augmentation de surface induite par la duplication des chemins de données. De plus, les techniques de pipeline et de parallélisme sont limitées, comme dans le cas des processeurs génériques, par la présence de contrôle dans les programmes.

Lorsque le circuit de départ est déjà optimisé pour la vitesse, le rendre plus rapide afin d'abaisser sa consommation devient extrêmement complexe. On peut alors proposer des circuits dans lesquels les parties non critiques sont alimentées par une tension plus faible [WCR+99] ou même proposer des variations dynamiques de la tension d'alimentation [LWH97, JR97]. La présence de sources d'alimentation de voltages différents pose des problèmes de réalisation (présence de convertisseurs de tension dans le circuit, grilles de distribution multiples) mais cette technique a déjà été utilisée sur des circuits et a prouvé sa faisabilité technique.

Un circuit peut également être cadencé par plusieurs sources d'horloge. Ici encore ce sont les parties non critiques qui sont ralenties. On obtient alors des circuits globalement asynchrones mais localement synchrones [MHK+98]. Cette méthode a de plus l'avantage de simplifier la gestion de la propagation du signal d'horloge et limite les effets de dérives dus à la vitesse de propagation finie du courant dans les circuits. Les techniques d'asynchronismes locaux sont encore peu explorées mais des transformations sur des circuits existants ont permis d'atteindre des réductions de la consommation allant jusqu'à 70% [HMK+99] par rapport à leur équivalent totalement synchrone. La variation dynamique de la fréquence d'horloge permet également d'avoir des gains en consommation. L'utilisation extrême en est représentée par la coupure du signal d'horloge dans les parties temporairement non utilisées (*clock gating*).

Une part importante de la consommation provient des changements d'état du circuit (*switching activity*). La minimisation du nombre d'opérations nécessaires pour effectuer un calcul permet en grande partie de réduire ce nombre de changements d'état [HPK97] mais n'est pas suffisante car il faut aussi prendre en compte les changements dus aux opérandes [LRJD99]. Si une donnée apparaît plusieurs fois dans un calcul et est utilisée pour un même type d'opérations successives, par exemple plusieurs additions d'une variable $a$, les étapes d'ordonnancement et d'allocation doivent prendre en compte cette propriété afin que le même additionneur soit utilisé pour ces opérations. En effet, si on utilise le même additionneur alors l'entrée correspondant à la valeur $a$ est déjà positionnée et ne nécessite pas de changement d'état. L'ordonnancement est ici important car il faut s'assurer que ces additions ne sont pas exécutées dans le même cycle pour que la ressource soit effectivement disponible.

La synthèse comportementale sous contrainte de consommation électrique est un sujet de recherche déjà très développé. Le lecteur pourra trouver une étude détaillée des différentes méthodes utilisées dans [MPS98] et [Ped96].

#### Les architectures asynchrones

Les architectures synchrones sont construites sur deux postulats : les signaux manipulés sont binaires et le temps est discret. Les signaux binaires permettent d'utiliser une logique booléenne pour les calculs. La discrétisation du temps permet de simplifier la conception de la logique de contrôle. Les architectures asynchrones [Hau95, BJN99] sont des architectures utilisant des signaux binaires mais pas de signal d'horloge. L'enchaînement des calculs y est effectué par signalisation et propagation des résultats le long des chemins de données. Ce mécanisme requiert un doublement des fils présents sur le circuit et donc une augmentation de la taille. Cependant, l'utilisation de l'asynchronisme possède de nombreux avantages et contrebalance avantageusement ce surcoût.

- On s'affranchit des problèmes de synchronisation globale des circuits et de dérive d'horloge car il n'y a, par définition, pas d'horloge globale distribuée.
- La mise en veille des parties non utilisées est faite naturellement et de manière optimale par

construction alors que cette mise en veille requiert une attention spéciale dans les circuits synchrones (annulation du signal d'horloge – *clock gating*).
- Les circuits asynchrones mettent en avant les *performances moyennes* de chaque partie du circuit et non celles du pire cas, comme cela est imposé dans les circuits synchrones.
- Les architectures asynchrones sont modulaires car elles s'affranchissent de nombreux problèmes posés par les interfaces de modules (seul le protocole est important ici). Chaque sous-système peut être optimisé indépendamment des autres sans se préoccuper de la synchronisation des entrées ou des sorties du module.
- Le fonctionnement s'adapte aux contraintes physiques extérieures. Les performances d'un circuit sont en effet soumises à des facteurs externes comme la température de fonctionnement et la régularité de la tension d'alimentation. Les concepteurs de circuits synchrones doivent considérer les conditions les plus défavorables et fixer les contraintes d'horloge alors que les circuits asynchrones peuvent s'adapter automatiquement et leur vitesse de fonctionnement s'adapte aux contraintes physiques externes.

On peut voir que les circuits asynchrones présentent de nombreux avantages. S'ils ne sont pas plus utilisés, pour l'instant, dans les conceptions c'est aussi qu'ils ont quelques désavantages. Tout d'abord ils sont plus compliqués à concevoir que les circuits synchrones. La conception d'un circuit synchrone peut être faite en construisant la logique combinatoire de contrôle et en la temporisant avec des registres. Tous les aléas de transitions peuvent être résolus en fixant une période d'horloge suffisamment longue pour le circuit. L'état dynamique d'un circuit asynchrone est beaucoup plus délicat et est à la charge du concepteur. Le diagramme de contrôle est, pour les mêmes raisons, plus difficile à concevoir et le bon séquencement des opérations n'est pas assuré par la temporisation discrète des opérations. Les outils de synthèse logique, de partitionnement, de placement et de routage actuels doivent être modifiés pour la fabrication de circuits asynchrones. Malgré ces difficultés, des processeurs asynchrones complets sont déjà disponibles [FEG00] et il ne fait nul doute qu'ils feront bientôt partie des composants standard utilisés dans la conception de systèmes.

### 1.3.2 Les communications

Les SoC modernes sont partitionnés en blocs et en sous-systèmes afin de mieux contrôler la complexité des réalisations. Ces blocs sont conçus pour pouvoir opérer de façon indépendante afin de permettre leur réutilisation et la simplification de leur interconnexion. L'assemblage de ces blocs doit donc se faire en utilisant des moyens de communication standardisés tout en restant adaptés aux besoins du système. La gestion et la synthèse des communications font partie intégrante du processus de codesign et nécessitent un soin particulier car les communications représentent une part importante de la consommation électrique globale des systèmes. De plus, un système embarqué est rarement coupé du monde extérieur et intègre donc des moyens de communication avec des périphériques externes.

Les communications internes ou externes d'un système imposent la mise en place de mécanismes complexes tels que l'utilisation de bus standardisés (par exemple des bus PCI, USB ou Firewire) organisés en hiérarchies et contrôlés par des gestionnaires d'arbitration. La complexité et la diversité des modules utilisés dans les SoC permettent de rapprocher leur architecture et leur mode de fonctionnement des notions utilisées dans le domaine des réseaux et des systèmes répartis. Les communications sont ainsi découpées en couches et en protocoles. La représentation la plus couramment utilisée est organisée selon les couches suivantes [BM00, CCH+99] :

- applicative : niveau d'interaction des composants, les communications sont vues comme des canaux abstraits ;
- transaction : gestion des liaisons point à point entre les modules ;
- transferts : gestion des protocoles permettant d'effectuer une communication sur un bus ;
- couche physique : câblage physique des bus et gestion de la synchronisation.

L'utilisation d'un modèle comme celui-ci permet de réduire la complexité de conception de chacune des couches en les considérant comme indépendantes. Les recherches en systèmes VLSI basse consommation ont, pour l'instant, porté leurs efforts sur les couches les plus basses de cette pile : la couche physique et la couche de transferts.

Comme pour les unités de calcul, la réduction de consommation s'obtient ici encore en abaissant la tension d'alimentation au niveau physique et en minimisant le nombre moyen de transitions. Le niveau physique ne sera pas traité ici, un résumé des techniques de commutation basse tension et d'encodage des données pourra être trouvé dans [BM00]. La réduction du nombre moyen de transitions que nous présentons ici utilise les corrélations spatio-temporelles des données pour construire des séquences de transfert minimisant les changements d'état.

#### Réduction des transitions sur un bus

Ces techniques minimisent les transitions sur le bus d'adresse concernant le chargement des instructions. Ces adresses sont très souvent séquentielles (comme les adresses consécutives permettant de charger le code d'un bloc de base de programme depuis la mémoire). On peut alors mettre en place une signalisation, appelée code T0 [BMM+98], réalisée par un câblage supplémentaire, permettant de ne pas modifier l'état d'un bus si la requête courante concerne une adresse consécutive à la précédente.

Les transferts des données ne sont pas nécessairement successifs mais la modification des adresses réelles des données en mémoire permet d'utiliser un code de Gray pour générer les adresses qui seront utilisées [PD99, PDN99b]. Un code de Gray minimise les transitions sur le bus car la distance de Hamming entre deux adresses consécutives est réduite à un (on ne change qu'un seul bit du mot d'adresse pour avoir l'adresse suivante).

La réduction du nombre de transferts sur les bus de données résulte de l'optimisation transferts sur les hiérarchies de mémoires et constitue l'objet de cette thèse. Elle sera donc présente tout au long des chapitres suivants.

### 1.3.3 Les mémoires

Un système complet utilise différents types de mémoires selon les besoins que l'on peut classer selon deux catégories : les mémoires mortes et les mémoires vives.

Les mémoires mortes, ou « non volatiles », permettent de conserver l'information mémorisée même en l'absence de tension d'alimentation. Ces mémoires sont utilisées pour stocker les informations nécessaires au démarrage et au fonctionnement des systèmes (programmes d'initialisations, système d'exploitation et applications). Ces mémoires ne sont souvent utilisables qu'en lecture seule et sont dans ce cas des mémoires ROM (*Read Only Memory*). La construction des mémoires mortes ne sera pas étudiée ici car elles n'ont que très peu d'influence sur la consommation des applications dominées par les données.

Les mémoires vives, contrairement aux mémoires mortes, nécessitent une tension d'alimentation permanente pour assurer leur fonction de stockage. Une coupure, même très brève, de la tension d'alimentation provoque la perte des données. Les RAM (*Random Access Memory*) sont des mémoires vives permettant d'avoir un accès aléatoire (par opposition à séquentiel) aux données. Ces accès sont définis par une adresse et un mode de lecture ou d'écriture.

Nous considérons principalement deux grands types de technologies pour les cellules élémentaires de stockage des RAM. Le premier correspond à des mémoires statiques appelées SRAM (*static random access memory*) et sert à constituer les éléments mémoires embarqués sur la puce de silicium comme les registres ou les mémoires caches. Ces mémoires offrent les meilleures performances mais ont un coût relativement élevé et sont consommatrices d'énergie. Le second type de cellule correspond aux mémoires dites DRAM (*dynamic random access memory*). Ce type de mémoire n'est, pour l'instant, pas embarqué sur le silicium (les mémoires DRAM embarquées commencent tout juste à apparaître) et permet d'avoir des mémoires de grande capacité à un faible coût, au détriment d'un temps d'accès nettement supérieur aux mémoires SRAM.

De manière générale, une mémoire peut être découpée en *bancs* ou *blocs* comme nous le développons dans les deux sections suivantes. Ce découpage permet de réduire le coût d'un accès mais augmente en contrepartie la complexité de décodage des adresses et la surface utilisée pour une même capacité. Ces deux facteurs réduisent le gain obtenu en consommation. La construction d'une mémoire utilisant des sous-blocs d'adressage de manière optimale est un des nombreux problèmes à considérer lors de la conception. Une présentation détaillée des mémoires et de leur construction peut être trouvée dans [PH94].

### Les mémoires statiques

Les mémoires SRAM embarquées ainsi que les bancs de registres sont constitués à partir d'un même schéma de cellule de base présenté sur la figure 1.6. Ces cellules élémentaires sont des circuits intégrés composés à partir de logiques. Le stockage de l'information est assuré par les portes NOR montées en rebouclage. Une porte NOR agit comme un inverseur si l'autre entrée est nulle. Les portes NOR montées en opposition enregistrent la valeur de l'état sauf si l'entrée d'horloge est à 1, auquel cas la valeur de l'entrée de donnée remplace la valeur enregistrée. Les portes ET permettent de synchroniser les changements d'état sur le signal d'horloge.

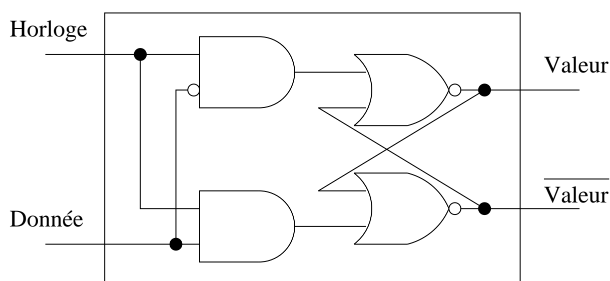



FIG. 1.6 – Élément de stockage d'une mémoire SRAM : bascule D

Du fait de l'utilisation de la même technologie une cellule SRAM comporte de 4 à 6 transistors ; on peut appliquer sur ce type de mémoire les mêmes techniques de minimisation de la consommation que pour les unités de calcul (conception physique, variation de voltage et de fréquence d'horloge).

Les mémoires construites à partir de ces éléments contiennent des réseaux de cellules avec (généralement) un port d'accès unique qui permet soit une lecture soit une écriture. Les mémoires SRAM ont un temps d'accès constant quelle que soit la donnée, bien que les caractéristiques des accès en lecture et en écriture soient souvent différentes.

Une mémoire de type 4x2 (8 bits répartis en 4 lignes de 2 bits) est représentée de manière simplifiée sur la figure 1.7. Un module de mémoire nécessite en plus des cellules de stockage une logique de décodage des adresses ainsi que des signaux d'autorisation de sortie et d'écriture pour chaque bascule. Les signaux d'autorisation d'écriture et de sortie ne sont pas représentés pour simplifier le schéma. Pour permettre à plusieurs sources d'écrire sur un ligne unique, un émetteur trois états est utilisé. Un émetteur trois états possède deux entrées : un signal de donnée et un signal « autoriser sortie ». Si celui-ci est désimposé la sortie d'une bascule D se trouve dans un état de haute impédance ce qui permet à un autre émetteur trois états dont le signal « autoriser sortie » est imposé de définir la sortie partagée. Il est primordial que le signal « autoriser sortie » ne soit imposé que pour un des émetteurs trois états au plus ; sinon les émetteurs trois états peuvent essayer de positionner la ligne de sortie différemment. Le signal « autoriser écriture » fonctionne de la même manière et sert à provoquer l'écriture de la valeur imposée sur la ligne d'entrée dans la bascule sélectionnée. L'utilisation d'un ensemble d'émetteurs trois états distribués conduit à une mise en œuvre plus efficace qu'un multiplexeur de grande taille centralisé.

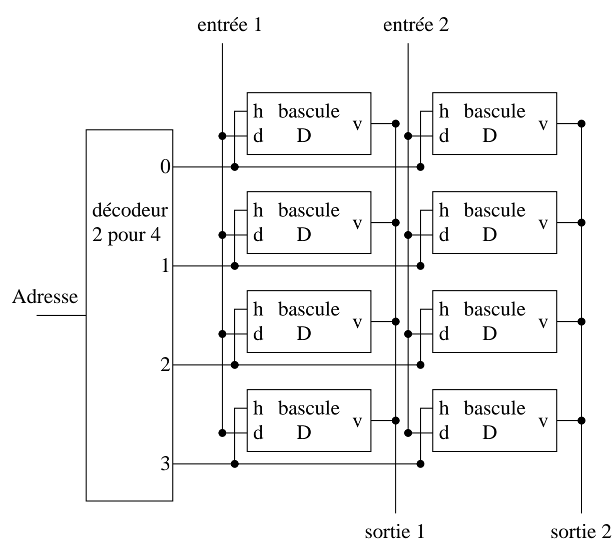



FIG. 1.7 – Structure simplifiée d'une mémoire SRAM 4x2

Le modèle de construction présenté nécessite un très grand décodeur et un nombre correspondant de lignes de mots élevé. Par exemple, dans une SRAM de 16Kx8, nous aurions besoin d'un décodeur 14 vers 16000 et de 16000 lignes de mots (qui sont les lignes utilisées pour activer les différentes bascules) !

Pour contourner ce problème, les mémoires de grande taille sont organisées sous forme de tableaux rectangulaires (ou blocs), comme présenté à la figure 1.8, et elles utilisent un processus de décodage en deux étapes. Un premier décodeur génère les adresses de ligne pour chaque bloc ; puis un ensemble de multiplexeurs est utilisé pour sélectionner un bit dans chacun des blocs. Comme nous le verrons, le processus de décodage en deux étapes est très important pour comprendre le fonctionnement des DRAM.

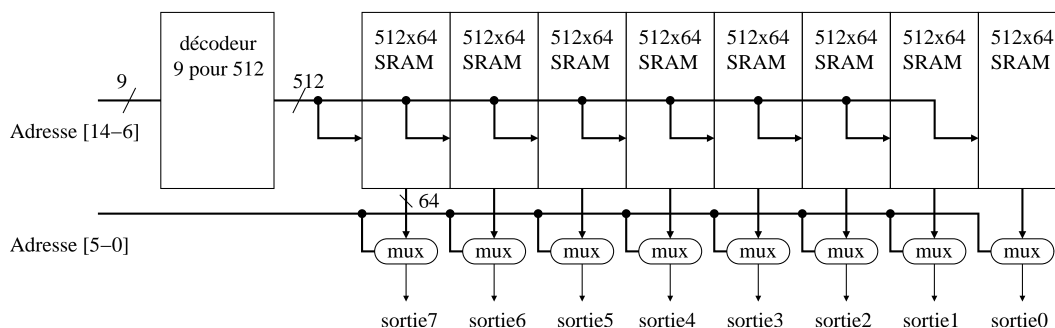



FIG. 1.8 – Organisation d'une SRAM utilisant un décodage en deux étapes

Les mémoires SRAM rapides seront appelées par la suite des « mémoires de premier niveau » ou des « mémoires embarquées ».

### Les mémoires dynamiques

Malgré la souplesse et la simplicité d'utilisation des SRAM, le besoin s'est fait sentir d'utiliser des RAM dynamiques pourtant plus complexes d'emploi.

Les raisons, au nombre de deux, sont très simples :

- à prix égal, mais surtout à encombrement égal, les RAM dynamiques ont une capacité quatre fois supérieure à celle des RAM statiques ;
- à capacité égale, leur consommation est inférieure.

Ceci s'explique par le fait que la cellule élémentaire d'une DRAM ne comporte que deux transistors, contre six pour les SRAM.

Dans une SRAM la valeur stockée dans une cellule est conservée dans une paire de portes inverseuses, et tant que le composant est alimenté en électricité, la valeur peut être conservée indéfiniment. Dans une RAM dynamique, la valeur conservée dans une cellule est stockée sous forme de charge électrique dans un condensateur (réalisé physiquement par un transistor ayant une couche de silice isolante entre la grille et le canal). La structure d'un élément de mémoire dynamique est présentée sur la figure 1.9. La charge du condensateur permet de déterminer l'état de la cellule mémoire et de différencier les états logiques « 0 » et « 1 ». Un unique transistor est ensuite utilisé pour accéder à cette charge stockée, soit pour en lire la valeur soit pour modifier la valeur. Lorsque le signal sur la ligne de mots est imposé, le transistor est ouvert, il connecte le condensateur à la ligne de bits. Si l'opération est une écriture, alors la valeur à écrire est placée sur la ligne de bits. Si la valeur est un 1, le condensateur sera chargé. Si la valeur est un 0, le condensateur sera déchargé. La lecture procède de la même manière en transmettant la valeur de charge du condensateur sur la ligne de bits.

C'est la présence de ces condensateurs qui constitue une des limitations des mémoires actuelles. Les condensateurs ne sont pas parfaits et nécessitent un rafraîchissement périodique de leur charge afin de compenser les pertes dues à leur imperfection. Ce rafraîchissement est effectué par une lecture puis une réécriture des éléments mémoires. Les constructeurs indiquent que la mémoire doit être rafraîchie toutes les 64ms. Si tous les bits d'une mémoire devaient être lus puis réécrits individuellement les DRAM de grande taille seraient en permanence en train d'être rafraîchies et seraient indisponibles pour le processeurs. Contrairement à ce que l'on pourrait penser, ce rafraîchis-

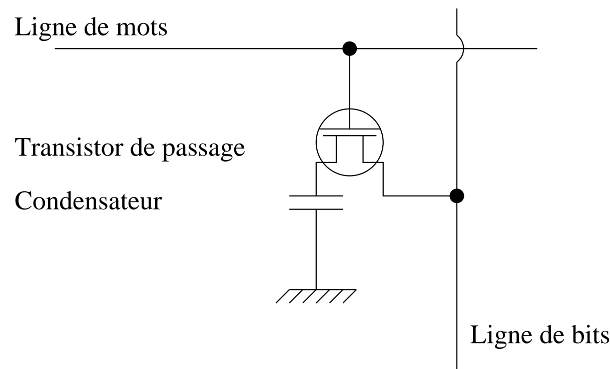



FIG. 1.9 – Schéma de principe d'une cellule de mémoire DRAM

sement automatique ne représente pas un problème important pour la performance des architectures DRAM. Dans la pratique, ces mémoires ne sont pas directement reliées aux bus du microprocesseur mais interfacées par des circuits spécifiques chargés de la gestion des accès et des rafraîchissements. La mémoire n'est indisponible que pendant un peu moins de 1% des cycles actifs de la DRAM.

Les blocs de DRAM utilisent une structure de décodage d'adresse à deux niveaux (figure 1.10). Ce découpage permet non seulement de rafraîchir une ligne entière en un cycle de lecture immédiatement suivi d'un cycle d'écriture mais aussi de ne pas trop augmenter la surface des boîtiers à cause des nombreux fils d'adresses. Lors d'un accès en lecture ou en écriture, le circuit de gestion des DRAM traduit une adresse du bus relié au processeur en adresses de « lignes » et en adresses de « colonnes ».

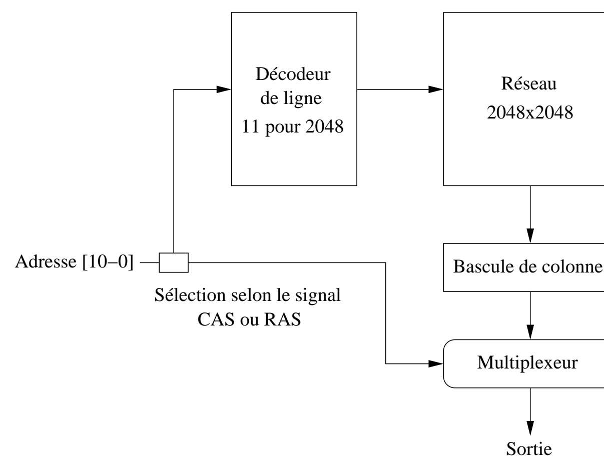



FIG. 1.10 – Conception d'une mémoire DRAM avec la structure de décodage d'adresse.

Le gestionnaire présente en entrée de la DRAM les adresses lignes et valide l'entrée *RAS* (*Row Address Strobe*), ce qui a pour effet d'activer la ligne de mots correspondante. La ligne complète est alors recopiée dans un ensemble de bascules de colonnes. Il présente ensuite en entrée de la DRAM les adresses colonnes et valide l'entrée *CAS* (*Column Address Strobe*), ce qui a pour effet de compléter la lecture en ne sélectionnant que le bit correspondant à l'adresse complète.

La lecture de l'état d'un condensateur décharge celui-ci. Une cellule doit être rechargée à chaque fois que sa ligne est accédée en lecture. Cela implique des temps d'accès aux colonnes plus longs

et des temps de latence plus importants. Le schéma d'adressage à deux niveaux et la circuiterie interne rendent les temps d'accès aux DRAM beaucoup plus longs (d'un facteur 5 à 10) que les temps d'accès aux SRAM.

Les mémoires de ce type seront appelées par la suite des « mémoires principales » ou des « mémoires externes ».

### Le futur des mémoires

Les mémoires que nous venons de décrire utilisent toutes deux des méthodes de stockage reposant sur un principe de circulation électrique. Les nouvelles technologies que nous allons présenter brièvement dans cette partie remettent en cause ce principe en utilisant le magnétisme et la lumière comme moyen de véhiculer et stocker l'information.

Les mémoires magnétorésistives (MRAM) utilisent le magnétisme et non des propriétés électriques pour stocker les données. Le développement de cette technologie provient de recherches menées par les laboratoires d'IBM sur les jonctions de tunnels magnétiques (*magnetic tunnel junction*). Les mémoires MRAM nécessitent une faible alimentation électrique uniquement pour modifier la polarité des éléments mémoire d'une puce. Alors que les mémoires SRAM ou DRAM nécessitent une alimentation électrique continue pour conserver l'information, les MRAM utilisent des propriétés de magnétisation pour indiquer la présence d'un « 1 » ou d'un « 0 ». Ces mémoires sont donc non volatiles et ne nécessitent pas de rafraîchissement lors de leur fonctionnement. Les caractéristiques de ce type de mémoire permettront d'avoir des mémoires de grandes capacités comme celles obtenues avec les mémoires DRAM tout en ayant les performances des mémoires SRAM ainsi qu'une une très faible consommation électrique. Les équipes d'IBM et d'Infineon prévoient de disposer des premiers modules de tests vers 2003 et une production en volume vers 2004.

Une autre technologie en développement utilise des propriétés optiques. Le stockage optique s'est développé depuis les années 1980 avec l'apparition des cédéroms puis des DVD (*Digital Versatil Disc*). Les premiers DVD utilisaient seulement une surface des disques pour stocker les données atteignant une capacité de 3,9 Go mais la norme était prévue dès le départ pour supporter un stockage sur deux couches réfléchissantes portant la capacité à 15,9 Go pour un support double face. L'évolution de la maîtrise de ces technologies permet maintenant de ne plus considérer seulement la surface des objets comme support et utilise tout le volume comme espace de stockage. Les mémoires *holographiques* offrent la possibilité de stocker un téra octet dans un support cristallin de la taille d'un morceau de sucre. Contrairement aux mémoires magnétiques, l'utilisation des mémoires holographiques ne propose pas de solution basse consommation. En effet, une mémoire holographique est construite grâce à un système coûteux regroupant un laser de très haute précision, une matrice LCD et un capteur CCD. Les équipes d'IBM prévoient la production des premiers systèmes de stockage holographiques personnels (*Holographic Desktop Storage System – HDSS*) avant 2010. Il reste toutefois à résoudre de nombreux problèmes technologiques avant d'atteindre les performances attendues (1 To de stockage et 1 Go par seconde de débit) et surtout une utilisation possible dans les systèmes embarqués.

### Conclusion sur les mémoires

Ce que nous appelons « mémoire conventionnelle » est généralement un espace de grande taille constitué de cellules de mémoire dynamique. Les DRAM sont utilisées pour créer des modules mé-

moire de grande capacité et sont, pour l'instant, *externes* au SoC. Les accès à cette mémoire sont effectués à l'aide de bus de communications extrêmement coûteux en consommation. Il convient donc de mettre en place une gestion des transferts afin de réduire le nombre d'accès à cette mémoire et ainsi la consommation des systèmes. L'apparition récente de mémoires DRAM embarquées devrait permettre l'intégration des DRAM sur les puces des SoC, abaissant ainsi le coût des accès. Cette intégration ne fera pour autant pas disparaître les hiérarchies de mémoires ni la nécessité de minimiser le nombre de transfert entre les niveaux. La gestion des transferts et de la taille de la mémoire nécessaire à une application constitue le sujet de cette thèse et sera développée tout au long des chapitres suivants.

### 1.3.4 Hiérarchies de mémoires et mémoires caches

On a vu que les mémoires les moins chères à produire (en coût par bit) étaient les mémoires de type DRAM. Ces mémoires sont utilisées pour avoir de grandes capacités de stockage mais ne peuvent être utilisées directement pour les échanges avec le processeur lors des calculs car elles constitueraient alors un sévère goulot d'étranglement en terme de performances. Cette différence de performances constitue un problème majeur depuis que les capacités de calculs des processeurs ont atteint des niveaux de performances dépassant celui des mémoires. Une solution apportée très tôt (à la fin des années 60 pour certaines machines prototypes de recherche et à partir des Intel 80486 et des Motorola 68040 pour les machines personnelles les plus répandues) par les concepteurs de machines a été d'introduire des mémoires caches rapides servant à masquer la différence de performances entre le processeur et la mémoire. De nombreux ouvrages traitent de la conception des hiérarchies de mémoires qui sont maintenant devenues incontournables lors de la conception d'un système [PH94] (à l'exception des systèmes temps-réels durs qui n'en font pas usage pour éviter les temps variables d'accès aux données).

Les hiérarchies de mémoires dans les systèmes embarqués peuvent être multiples et distribuées sur plusieurs bus de communication comme présenté sur la figure 1.11.

Dans une hiérarchie de mémoires, une donnée n'est copiée d'un niveau à un autre que s'ils sont adjacents [ACFS94]. La figure 1.12 présente une architecture avec seulement deux niveaux de mémoire : une mémoire cache et une mémoire principale. Le niveau *supérieur* – celui le plus proche du processeur – utilise des technologies plus coûteuses et est plus petit et plus rapide que le niveau *inférieur*.

Une mémoire cache s'interface entre le processeur et la mémoire principale comme l'illustre la figure 1.12. Chaque requête du processeur est adressée au cache et non directement à la mémoire principale. Si une copie des données est disponible dans le cache celui-ci peut renvoyer une réponse immédiate au processeur mais dans le cas contraire le cache doit accéder au niveau inférieur de la hiérarchie, copier un bloc de données depuis ce niveau et renvoyer la réponse. Le principe des caches repose sur les propriétés de localités *temporelle* et *spaciale* des accès aux données dans les programmes. En effet, les programmes contiennent pour la plupart des boucles et il est donc fortement probable que les instructions et les données sont accédées de façon répétée, ce qui génère une localité temporelle élevée. De la même manière, les données sont utilisées dans la plupart des cas de manière séquentielle exhibant ainsi une localité spatiale dans la gestion des accès. Les transferts entre deux niveaux sont effectués par *blocs* (aussi appelés *lignes de cache*) afin de prendre en compte la localité spatiale, la localité temporelle étant gérée par la *politique de remplacement* de ces blocs. Les méthodes mises en œuvre pour gérer le remplacement des blocs jouent un rôle primordial dans les

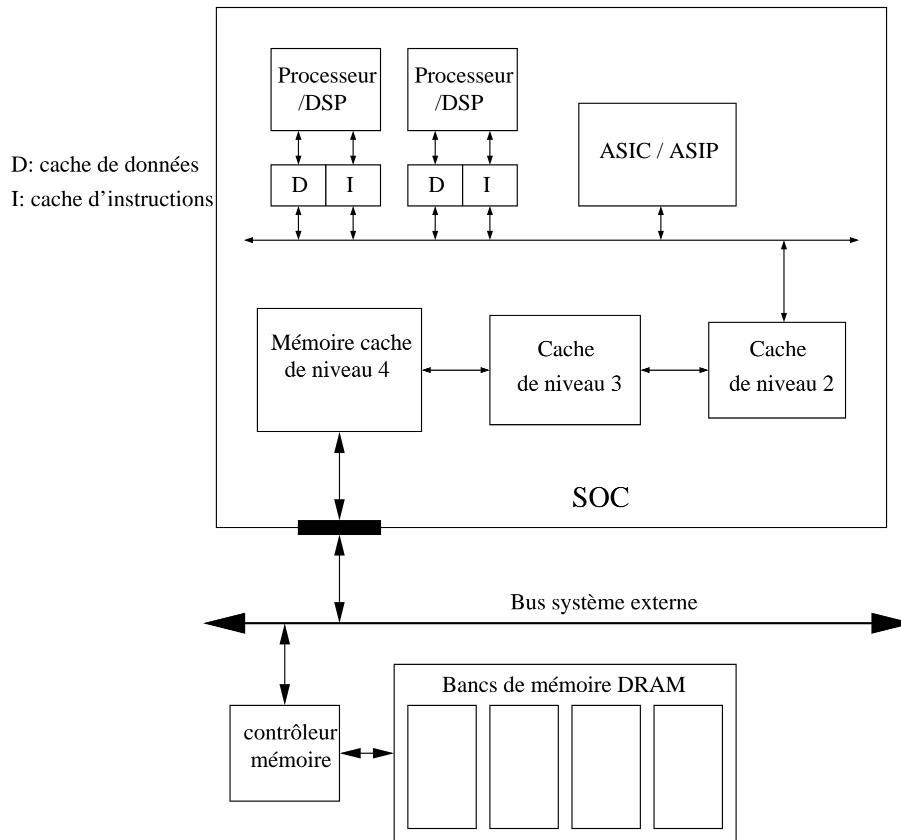



FIG. 1.11 – Architecture d'une hiérarchie mémoire

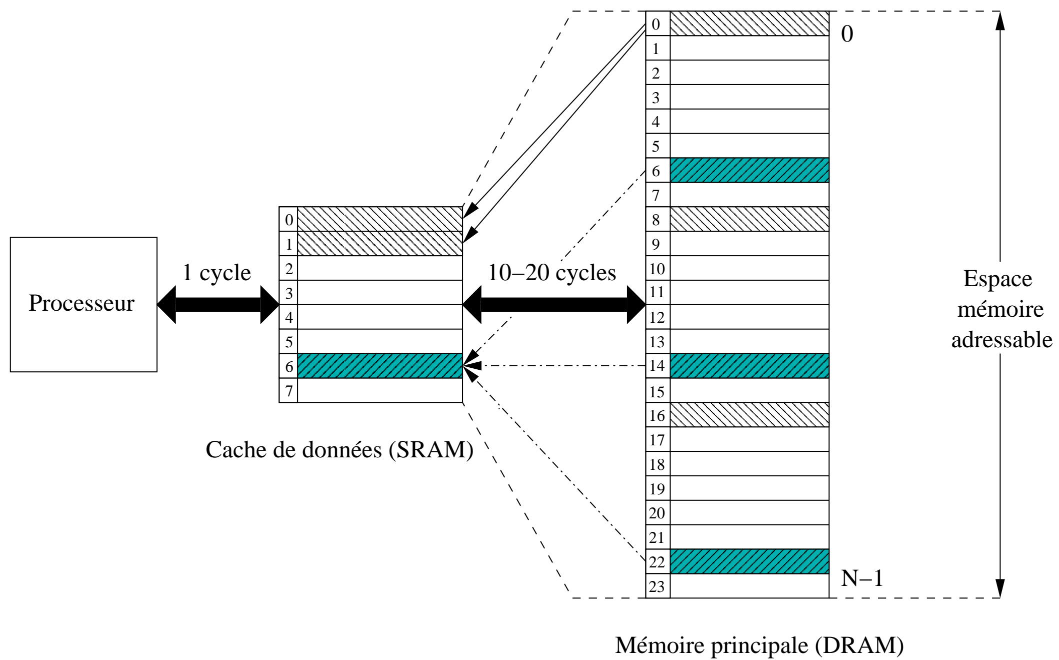



FIG. 1.12 – Architecture d'une mémoire cache

performances d'une hiérarchie. La gestion des caches fait ainsi l'objet de recherches très nombreuses aussi bien du point de vue de la conception de ceux-ci que pour le développement de méthodes et outils pour la compilation.

Le coût d'un accès à une donnée dans la hiérarchie dépend du niveau auquel on accède. Ce coût peut correspondre à deux critères : le temps nécessaire et l'énergie dépensée à récupérer la donnée. Le cas d'exemple proposé par Ko et Balsata [KBN98] utilise une hiérarchie à 4 niveaux. La consommation moyenne donnée pour les accès aux trois premiers niveaux de cache, embarqués sur la même puce que le processeur, est respectivement 150mW, 300 mW et 700mW. La quatrième niveau de la hiérarchie est une mémoire DRAM — externe — de grande capacité reliée à la puce contenant le processeur et les trois niveaux de cache par un bus de communication. La consommation moyenne d'un accès à cette mémoire est de 12,71W par transaction, soit presque 100 fois le coût d'un accès au premier niveau de mémoire cache.

Le but d'une bonne conception de hiérarchie mémoire est, ici, de minimiser la consommation moyenne des accès tout en restant en dessous des contraintes de temps d'exécution d'un programme et des contraintes de taille fixées pour la mémoire. Les approches proposées dans la littérature pour les optimisations de la mémoire peuvent être classées dans trois catégories. La première est la conception de hiérarchies matérielles, cette méthode repose sur l'utilisation de traces d'exécution des programmes et produit une hiérarchie mémoire dédiée à une application. La seconde transforme la description comportementale de l'application afin de l'adapter aux contraintes et mécanismes d'accès définis par une architecture mémoire fixe. Enfin, la troisième catégorie combine les deux premières en créant une hiérarchie mémoire dédiée ainsi que des transformations de programme permettant une adéquation optimale entre construction de la hiérarchie et utilisation de celle-ci.

Nous présentons ici les principaux types de modules de mémoire que l'on retrouve dans les architectures classiques ainsi que leur fonctionnement. Nous présentons ensuite la première approche permettant de construire une architecture dédiée. Cette première approche servira de base aux deux autres méthodes qui constituent l'objet de cette thèse et qui seront développées dans les chapitres suivants.

#### Mémoires caches gérées de façon matérielle

Ce type de mémoire représente le modèle le plus couramment utilisé dans les architectures modernes. Une mémoire cache étant plus petite que la mémoire de niveau inférieur son fonctionnement est contraint par la gestion des lignes de cache par rapport à l'espace d'adressage réel disponible dans la mémoire. On distingue trois méthodes de gestion des adresses pour les lignes de cache selon le degré d'*associativité* du cache.

- Cache à adressage direct (*Direct Mapped Cache*) : un bloc est recopié dans le cache à une adresse unique selon d'où il provient. Cette adresse est déterminée par une opération de modulo. Les blocs hachurés et grisés de la figure 1.12 sont, si on considère le cache présenté à adressage direct, assignés à la ligne de cache numéro 6. Un cache à adressage direct ne peut contenir à un moment donné qu'un seul de ces blocs.
- Cache associatif par ensembles (*Set Associative Cache*) : un bloc de mémoire peut être copié dans un ensemble de lignes du cache. Les caches associatifs par ensemble ont communément des ensembles de 2, 4 ou 8 blocs. Sur l'exemple de la figure 1.12, les blocs hachurés peuvent être placés dans les lignes 0 ou 1 du cache si celui-ci est associatif par ensemble de deux. Le cache pourra donc contenir en même temps deux blocs hachurés différents.

– Cache totalement associatif (*Fully Associative Cache*) : chaque bloc de mémoire peut être placé sur n'importe quelle ligne de cache.

La complexité de gestion du cache croît avec son degré d'associativité. En effet, une requête du processeur nécessite une recherche plus importante pour savoir si une donnée est présente dans le cache lorsque celui-ci est complètement associatif que lorsqu'il est à adressage direct. Cette complexité se traduit par une circuiterie plus importante en taille et implique aussi un accroissement de la consommation électrique.

Lorsque le processeur demande le contenu d'une adresse mémoire et que celle-ci n'est pas présente dans le cache on a alors un *défaut de cache*. On peut identifier trois types de défauts de cache :

– Défauts obligatoires de chargement : la donnée appartient à un bloc n'ayant pas encore été accédé par le programme et doit donc être chargée dans le cache une première fois en vue d'être utilisée. Ces défauts ne peuvent être évités.
– Défauts de capacité : si le cache n'est pas de taille suffisante pour contenir toutes les données dont le programme a besoin lors de son exécution, alors des blocs devront être enlevés du cache pour être remplacés par d'autres et être rechargés plus tard dans l'exécution du programme. En pratique, le nombre de défauts de capacité est négligeable par rapport aux défauts engendrés par des défauts de conflit [Kul01].
– Défauts de conflit : ce type de faute correspond à la demande du processeur d'un nombre de blocs, se projetant dans le même ensemble de lignes de cache, supérieur à la taille de cet ensemble. Ce type de conflit ne peut se produire dans un cache totalement associatif.

Les caches totalement associatifs permettent de supprimer les défauts de conflit mais ils requièrent une logique de contrôle et de recherche des lignes présentes dans le cache beaucoup plus importante ; leur utilisation n'est pas nécessairement synonyme de baisse de consommation.

Pour les caches associatifs par ensemble ou totalement associatifs, le choix des lignes à remplacer lors d'un défaut de cache peut être géré selon différents algorithmes :

– Choix de l'élément le plus ancien (*Least-recently Used – LRU*) : le cache maintient une table indiquant les dates d'accès aux blocs. On remplace ici le bloc qui a la date de dernière utilisation la plus ancienne.
– Choix de l'élément le moins utilisé (*Least-frequently used – LFU*) : le cache maintient des statistiques d'accès aux blocs et le moins utilisé est remplacé.
– Ordre d'ancienneté (*First in first out – FIFO*) : les blocs sont remplacés suivant l'ordre dans lequel ils arrivent dans le cache.
– Aléatoire : les blocs candidats sont choisis aléatoirement afin de favoriser une allocation uniforme.

Qu'en est-il des écritures ? Le cache doit assurer une réponse rapide au processeur mais doit également assurer la cohérence des données entre le cache et la mémoire de niveau inférieur. On distingue les écritures ayant pour cible un emplacement mémoire dont une copie est disponible dans le cache des écritures générant une faute de cache. Lorsque la copie est disponible, les caches peuvent avoir un des comportements suivants :

– Écriture simultanée (*Hit write Through*) : les écritures sont effectuées à la fois dans le cache et dans la mémoire principale. Cette méthode permet d'avoir une cohérence entre l'état de la mémoire cache et la mémoire principale.
– Écriture différée (*Hit write back*) : les écritures ne sont effectuées que dans le cache. Un bloc modifié doit être réécrit en mémoire principale en cas de remplacement de celui-ci dans le

cache afin d'assurer la cohérence avec la mémoire principale.

Si une copie de la donnée à modifier n'est pas disponible dans le cache alors la méthode utilisée est choisie parmi les deux suivantes :

- *Miss fetch on write* : méthode aussi connue sous le nom de *write allocate*. Le bloc est ramené dans le cache puis modifié. La pénalité pour une faute de cache en écriture est donc la même que pour une faute de lecture car il faut le temps de lire un bloc complet depuis la mémoire de niveau inférieur.
- *Miss write around* : les données sont écrites directement dans la mémoire principale sans passer par le cache. Cette méthode peut être plus rapide pour des écritures uniques mais une autre faute sera générée si le bloc est réutilisé plus tard.

Les paramètres de fabrication d'une mémoire cache gérée de façon matérielle, que nous venons de survoler ici, ont une influence importante dans la conception d'une architecture pour un système embarqué. Il convient donc de construire une architecture avec soin pour limiter les défauts de cache et réduire le nombre de transferts entre les différents niveaux. La construction d'une hiérarchie dédiée à une application est présentée page 33 et les méthodes d'optimisation comportant des modifications des applications constituent le sujet de ce travail et seront présentées dans les chapitres suivants.

#### Mémoires caches gérées de façon logicielle

Les caches matériels restent utilisés dans la majorité des cas car ils sont les plus faciles à mettre en place pour les concepteurs. Cependant, ils ne sont pas les plus performants ni les plus économiques du point de vue de la consommation, loin s'en faut. De plus, certaines classes d'applications comme les applications multimédia ou des protocoles de contrôle de réseaux évolués comme ATM, bien qu'ayant des accès réguliers, ont un comportement vis-à-vis de la gestion mémoire ne correspondant pas aux méthodes utilisées. La tendance actuelle, à l'instar du mouvement ayant provoqué l'apparition des architectures RISC voulant remplacer les machines CISC, relègue la gestion des mémoires caches au compilateur ou, à défaut d'outils, au programmeur.

Les caches logiciels ont la même structure de lignes que les caches matériels mais la gestion des chargements et des remplacements est à la charge du programme. Les processeurs doivent donc proposer des instructions spéciales permettant de pouvoir contrôler, totalement ou partiellement, la gestion des caches. Le tableau 1.2 montre les différences de caractéristiques entre les caches matériels et les caches logiciels.

| Caractéristiques | Cache matériel | Cache logiciel |
|---|---|---|
| Unités de gestion | lignes et ensembles | lignes |
| Transferts | matériel | partiellement logiciel |
| Remplacement | matériel | logiciel |
| Politique de remplacement | matériel | logiciel |

TAB. 1.2 – Différences de caractéristiques entre caches matériels et logiciels

La complexité matérielle nécessaire à la mise en œuvre des caches logiciels est bien moins im-

portante que pour les caches matériels car on élimine toute la circuiterie nécessaire au contrôle. Ils sont donc bien moins consommateurs en énergie.

De plus, les caches matériels gèrent les transferts selon l'ordre d'exécution et des mesures statistiques fixes. En comparaison, les caches logiciels utilisent des directives, provenant du compilateur ou du programmeur, pour assurer une gestion dynamique dédiée à l'application.

Enfin, une des caractéristiques les plus importantes des caches logiciels est la possibilité de contrôler la cohérence des données entre la mémoire cache et les niveaux inférieurs. Le compilateur peut décider si une donnée doit être mise à jour dans la mémoire principale et quand cette mise à jour doit avoir lieu.

La présence d'un cache géré par le logiciel présente donc de nombreux avantages ; mais cette utilisation nécessite une analyse importante du programme et des outils de compilation adaptés. Les modifications de programme utilisées pour la mise en place des mémoires caches sont présentées au prochain chapitre.

#### Mémoires auxiliaires — *Scratch Pad Memories*

A mi-chemin entre les mémoires caches matérielles et les mémoires caches logicielles, Panda, Dutt et Nicolau [Pan98, PDN99a, PDN99b] proposent la mise en place d'une hiérarchie mémoire mixte. Cette hiérarchie combine les solutions proposées précédemment sous la forme présentée figure 1.13.

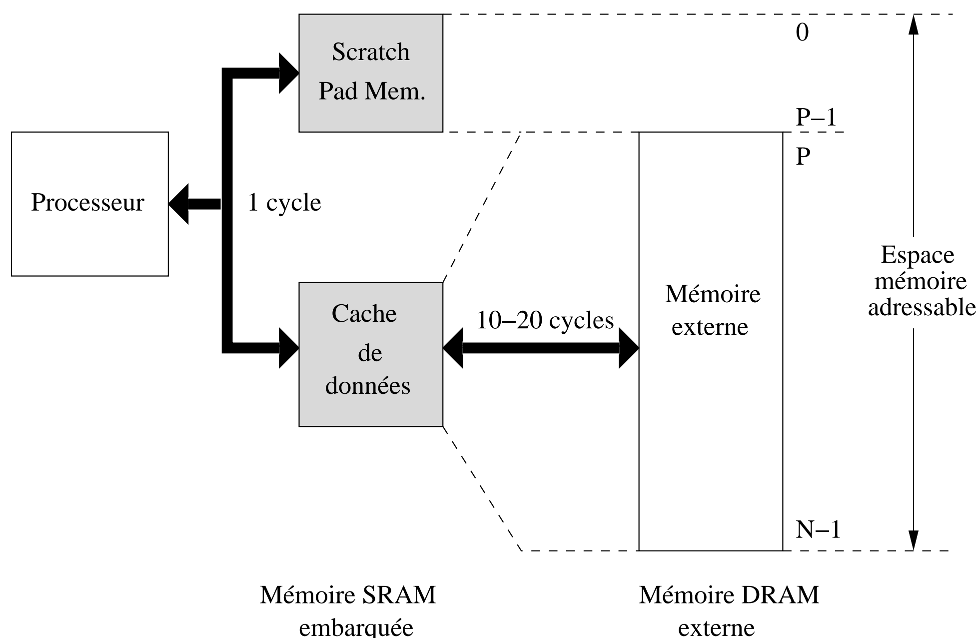



FIG. 1.13 – Mémoire auxiliaire : *Scratch Pad Memory*

Dans le système qu'ils proposent, la mémoire embarquée est composée d'une mémoire cache standard et d'une mémoire SRAM appelée *scratch-pad SRAM* faisant partie de l'espace d'adressage total du système. Cette mémoire permet de stocker dans une mémoire ayant les performances d'une mémoire cache les données dont l'utilisation provoquerait un grand nombre de fautes de cache. Il ne s'agit pas ici d'un cache logiciel mais d'une mémoire embarquée dédiée à des données utilisées

de manière intensive. Cette mémoire augmente la surface des SoC mais permet d'avoir un gain très important en performance et en consommation.

### Création d'une hiérarchie mémoire dédiée

La mise en place d'une hiérarchie de mémoire cache est un processus complexe et dépendant de manière étroite du type d'application traité. Le choix des paramètres architecturaux relève de nombreux compromis effectués entre performances, consommation et souplesse d'adaptation.

Nous présentons dans cette section les méthodes permettant de concevoir des hiérarchies dédiées aux transferts des données et des instructions d'une application.

**Mise en place de caches de données :** l'exemple illustrant les différentes consommations obtenues selon les niveaux d'accès à la hiérarchie (page 29) laisse à penser que performances et consommation vont de pair et que construire une hiérarchie permettant d'augmenter la vitesse du système réduit la consommation de celui-ci. Ceci n'est pas forcément vrai car les deux critères ne sont pas évalués selon les mêmes fonctions de coût, la performance en temps d'exécution d'un système s'évalue sur un chemin critique alors que la consommation s'évalue sur la moyenne de tous les accès mémoire.

Les expérimentations de Schiue et Chakrabarti [SC99] sur la construction d'une architecture dédiée au décodage MPEG montrent qu'un cache de petite taille (cache de 64 octets avec des lignes de 4 octets associatives par 8) permet de mettre en place une architecture proposant une décompression en 142 000 cycles pour une consommation d'énergie de $283\mu$J. L'architecture la plus performante trouvée par les auteurs utilise un cache de 512 octets avec des lignes de 16 octets associatives par 8. Cette architecture effectue les mêmes calculs en 121 000 cycles mais dépense une énergie de $1110\mu$J.

Il existe donc un compromis à traiter entre vitesse d'exécution et conception d'une hiérarchie mémoire dédiée. La performance d'une hiérarchie dépend presque exclusivement du nombre de fautes de cache subies durant l'exécution. Dans l'exemple, un cache plus important en taille totale, en taille des lignes et en nombre d'ensembles associatifs permet d'obtenir de meilleures performances ; mais dans le cas où les accès à la mémoire externe ne sont pas trop coûteux par rapport à une faute de cache (c'est le cas dans l'article cité) on peut avoir un compromis favorable entre la réduction de la taille du cache et le nombre de fautes de celui-ci.

De nombreux auteurs ont étudié l'efficacité de différentes architectures dédiées à des applications. Les principales recherches se tournent vers la sélection des modules de mémoire cache [KBN98, SC99, SD98, HKQ+98]. L'espace de recherche est discrétisé et paramétré pour permettre une exploration exhaustive, ou proche de l'exhaustif, des solutions. Ces études reposent toutes sur des profils d'exécution des applications et sont utilisées pour définir des architectures dans un contexte mono-processeur avec des hiérarchies ayant un ou plusieurs niveaux de cache. Le choix des composants s'effectue dans des ensembles prédéfinis en faisant varier les différents paramètres. Toutes les solutions sont ensuite évaluées par des simulations et celle mettant en œuvre les meilleurs compromis peut être choisie. La simulation est donc aussi la limitation de cette méthode car cela suppose que l'on puisse avoir des ensembles de tests représentatifs en nombre suffisant (ce qui peut être délicat dans le cas de programmes dont le temps de calcul dépend des données). Cette méthode de conception semble toutefois bien adaptée pour les architectures générales ayant besoin de pouvoir

s'adapter à divers types d'applications. Le support matériel de la gestion de la mémoire permet à une architecture d'avoir une capacité d'adaptation, au prix d'accès mémoires légèrement plus coûteux.

L'utilisation de systèmes mixtes utilisant les caches et les mémoires *scratch pad* n'est traité à notre connaissance que par Dutt, Panda et Nicolau [Pan98, PDN99b, PDN99a]. Les auteurs proposent en association avec ce type d'architecture un algorithme d'exploration permettant de construire une architecture optimale pour une application donnée. Cette architecture est définie par les paramètres suivants :
- La taille totale de la mémoire embarquée nécessaire ;
- le partitionnement de cette mémoire entre :
  - la mémoire *scratch-pad*, caractérisée par sa taille ;
  - le cache de données, caractérisé par sa taille ainsi que la taille des lignes de cache.

En complément de cette architecture, l'algorithme utilisé effectue également l'allocation des données en mémoire. Cette allocation détermine les variables qui seront disposées dans la mémoire embarquée et celles disposées dans la mémoire externe (et donc accédées via le mécanisme de cache). La décision de ce placement mémoire ne peut avoir lieu qu'*après l'étape de partitionnement entre matériel et logiciel* [PDN99a].

**Mise en place de caches d'instructions :** lorsque l'unité de calcul est un cœur de processeur, il faut également dimensionner la mémoire cache permettant de gérer le code des programmes. En effet, un processeur doit pouvoir disposer à chaque cycle d'horloge d'une nouvelle instruction, ou plus pour les processeurs superscalaires et VLIW. Les caches pour les instructions ne sont, en général, séparés des caches de données que pour le dernier niveau, celui le plus proche du processeur. Cette séparation permet d'utiliser toute la bande passante de chacun des deux caches et ainsi permettre l'alimentation soutenue des processeurs en instructions et en données.

Les caches d'instructions sont gérés de façon matérielle et ont des structures adaptées aux mécanismes de contrôle des programmes. On trouve, par exemple, deux types de caches utilisés couramment dans les systèmes.
- Les caches d'instructions prédécodées permettent de conserver les instructions les plus exécutées dans un format permettant de les utiliser sans avoir recours aux étapes de chargement et de décodage des processeurs.
- Les caches de boucles permettent de garder en mémoire les boucles des programmes. De tels caches sont moins efficaces que les précédents sur les programmes ayant des propriétés de localité très élevées sur certaines instructions, mais sont plus souples.

Les caches d'instructions et leur gestion ne seront pas détaillés dans les chapitres suivants car nous nous concentrons sur les transferts de données, largement dominants dans les applications multimédia. Toutefois un contrôle de la taille des programmes et un bon ordonnancement restent nécessaires lors de la compilation. Une présentation des techniques de compilation et de génération de code tenant compte de la consommation peut être trouvée dans [BM00].

### 1.3.5 Conclusion sur les architectures et les hiérarchies

Les choix proposés par les architectures pour les systèmes embarqués sont très riches et variés. Une implémentation soignée de l'architecture permet d'obtenir des gains importants en consommation. Les applications que nous considérons manipulent de grandes quantités de données et nécessitent la présence d'une hiérarchie mémoire complexe pour pouvoir être exécutées. Cependant,

comme nous l'avons déjà mentionné en introduction à ce chapitre, ne vouloir considérer les optimisations pour la consommation que du point de vue matériel est insuffisant. La présentation des composants architecturaux qui a été faite ici servira de base aux développements des autres chapitres présentant les modifications et optimisations pouvant être effectuées sur les programmes.

## 1.4 Motivations pour les transformations de programme

Les systèmes embarqués sont caractérisés par la présence d'une application logicielle spécifique tournant sur un système dédié. Dans la majorité des systèmes embarqués, comme les téléphones cellulaires ou les appareils électroniques portables, la consommation électrique est une des contraintes importantes à prendre en compte dès le début de la conception. Cependant, il n'y a que peu d'outils fournissant une aide aux concepteurs permettant d'évaluer la consommation. Dans l'état actuel des connaissances, une estimation fiable de la consommation n'est disponible que pour les niveaux de description les plus bas : niveau de conception final (transistors) d'un circuit ou éventuellement au niveau de description des portes logiques. Les simulations à ces niveaux sont trop lentes et restent impraticables (par interpolation à un haut niveau) pour des systèmes embarqués utilisant des processeurs enfouis et des hiérarchies mémoires complexes. De plus, ces techniques peuvent ne pas être du tout utilisables si l'on ne dispose pas des descriptions appropriées et suffisamment détaillées pour des éléments du système. Des recherches sont en cours pour mettre en place des estimations de consommation électrique au niveau de description architecturale. Ces estimations utilisent les instructions exécutées par le processeur comme base de stimulus et effectuent la somme des consommations des modules activés dans le système par une instruction. La consommation des modules est le plus souvent une constante et représente le coût d'activation d'un module, cette constante ne tient pas compte de la valeur des opérandes, de l'état interne des composants ni des relations d'interdépendances entre les modules. Dans le même domaine, des approches stochastiques permettent de modéliser la consommation électrique de systèmes. Cependant, le niveau d'abstraction atteint ici rend les estimations peu fiables. De plus, des détails des bas niveaux de la structure interne des éléments utilisés dans le système sont toujours nécessaires afin de modéliser le comportement de chaque module. Ce constat a lancé de grandes pistes de recherches afin d'intégrer dans la conception d'un système des critères permettant, non pas d'estimer la consommation en elle-même, mais d'intégrer le critère d'optimisation de la consommation dans la conception des systèmes embarqués.

Il est maintenant largement accepté que des modifications des programmes tournant dans les systèmes embarqués ont une grande influence sur la consommation électrique. Peu de choses ont jusqu'ici été mises en place pour mettre en œuvre des techniques traitant de ce sujet. Ce manque est imputable principalement à l'absence de retour permettant d'évaluer la consommation d'un logiciel. L'analyse au niveau des instructions permet, en partie, d'obtenir ce retour sur les transformations. L'application de ces techniques fournit l'information permettant de développer des logiciels et des systèmes performants du point de vue de la consommation électrique. Elle permet aussi d'identifier les points les plus consommateurs et ainsi de guider le concepteur dans le choix de l'architecture cible, notamment pour la conception des différents niveaux de hiérarchie mémoire.

### 1.4.1 Applications cibles

Les applications embarquées multimédia ont la particularité de nécessiter de grandes quantités de mémoire et d'en faire un usage intensif pour manipuler des flux de son, de vidéo ou d'images

fixes. Elles impliquent un parcours régulier de la mémoire associé à des contraintes temporelles. Ceci n'est réalisable que sur une architecture utilisant une mémoire hiérarchique à plusieurs niveaux. Des expérimentations de transformations sur les algorithmes de compression/décompression d'images du standard MPEG4 ont permis d'obtenir des gains substantiels en place mémoire ainsi qu'en consommation électrique pour une implémentation de ce standard [BCBM98]. Les résultats obtenus par transformations de programme à partir de la spécification du standard ISO, écrite en C, ont permis de réduire d'un facteur 65 le nombre d'accès à la mémoire principale par adaptation sur une architecture mémoire fixe.

Les applications multimédia sont très souvent directement conçues dans des langages impératifs comme C et C++. En effet, les comités de standardisation utilisent des implémentations de référence, dont la fonctionnalité a été validée, pour définir les standards. La construction et la validation de ces codes sont des étapes longues et très coûteuses. Optimiser directement les descriptions servant à définir les standards permet de se concentrer sur la partie fonctionnelle et diminue le temps nécessaire à la conception d'un système. Il est donc important de pouvoir disposer d'outils permettant de traiter ces descriptions exécutables et d'intégrer ces outils dans le flot de conception des SoC.

### 1.4.2 Optimisation mémoire et placement dans le flot de conception

Des systèmes de petite taille peuvent être conçus en effectuant le partitionnement et l'assignation décrits à la section 1.2 (page 10) de manière manuelle. Cependant, des designs de taille conséquente nécessitent impérativement la mise au point d'outils d'aide à la décision et à la conception passant par l'automatisation partielle ou complète des étapes. Des méthodes d'automatisation partielle commencent à se développer. On peut noter, par exemple, les méthodes de partitionnement comme celle décrite dans [KS98] qui permet d'étudier le codesign d'un ensemble de systèmes ayant des fonctionnalités proches et les méthodes d'assignation permettant de sélectionner parmi un ensemble prédéfini les composants architecturaux les mieux adaptés à une application [KLPMS99]. Des outils performants permettent de modéliser, simuler et évaluer à un haut niveau la mise en place d'un système combinant matériel et logiciel. Toutefois, la complexité du problème et la taille de l'espace de recherche des solutions ne permettront sans doute pas, comme pour la synthèse de haut niveau, d'avoir une méthodologie générale automatisée.

Une fois le partitionnement matériel/logiciel effectué, la mémoire est déjà divisée et les communications entre les différents modules sont *imposées* par le découpage, même si le moyen d'effectuer ces communications est encore abstrait. Il est donc très important d'optimiser la gestion des données d'une application avant cette étape de partitionnement afin de conserver une vue globale sur le trafic généré par les opérations sur la mémoire. Cette étape peut s'insérer, comme le présente la figure 1.14, dans le flot conception (figure 1.2) entre la validation d'une application et l'étape de partitionnement. Une fois l'*organisation* du système définie, les échanges entre les différents modules fonctionnels et la mémoire peuvent alors être optimisés. Ces optimisations considèrent les échanges entre une unité fonctionnelle et la hiérarchie (partagée avec les autres unités ou dédiées) et peuvent avoir lieu soit lors de la compilation de la partie logicielle, soit pendant la synthèse (ou instanciation) de la partie matérielle, soit encore de manière conjointe entre logiciel et matériel.

Le prochain chapitre présente les principes d'analyses et de transformation de programme. Cette présentation sera orientée vers les méthodes existantes pour l'optimisation de la mémoire dans les systèmes embarqués une fois le partitionnement matériel/logiciel effectué. Nous y présentons, tout d'abord, les transformations de programme, principalement de boucles, permettant d'adapter la

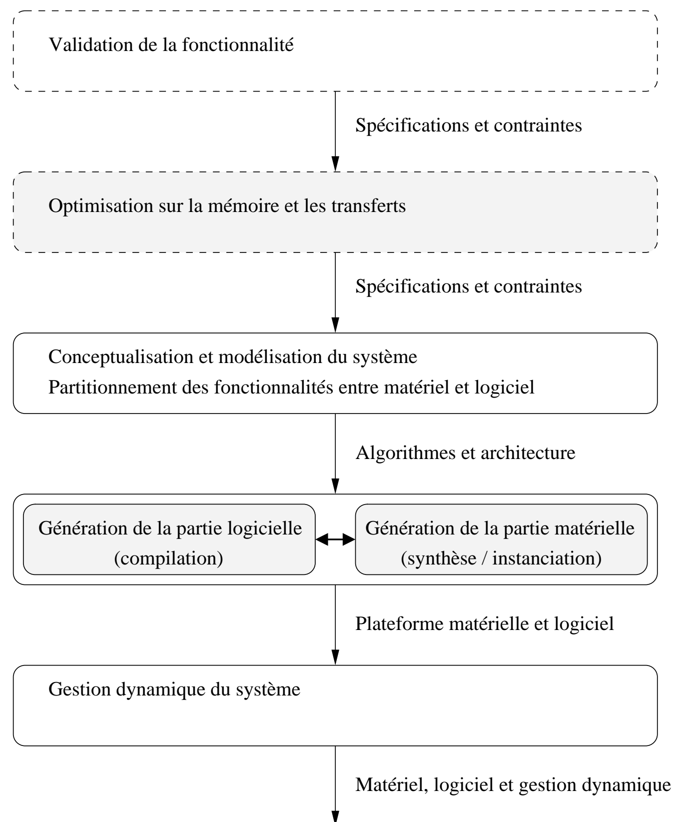



FIG. 1.14 – Optimisations dans le flot de conception

description d'une application à une hiérarchie mémoire fixée. Nous présentons également l'approche proposée par IMEC (*Interuniversitair Micro-Elektronica Centrum*) permettant d'effectuer de manière conjointe la mise en place de l'architecture et des transformations de code associées. Nous présentons plus en détail l'étape d'ordonnancement qui constitue une partie des travaux réalisés durant cette thèse.

Le chapitre 3 présente les transformations utilisables *avant* l'étape de partitionnement matériel/logiciel. Se placer à ce niveau du flot de synthèse présente le principal inconvénient de n'avoir aucun renseignement sur l'architecture de la hiérarchie mémoire qui sera utilisée. La seule supposition que l'on peut faire ici est qu'une telle hiérarchie avec au moins deux niveaux de mémoire sera instanciée dans le système. Nous présenterons les transformations possibles à ce niveau et plus particulièrement deux transformations développées dans le cadre de cette thèse : la fusion de boucles pour minimiser le nombre de tableaux temporaires et l'alignement de boucle pour la minimisation des variables temporaires inter-itérations.

Enfin, le chapitre 4 présente l'utilisation de ces méthodes au sein du flot de conception mis en place dans le logiciel VCC (*Virtual Component Codesign*) de la société Cadence. Le réalisme des fonctions objectives utilisées est évalué par rapport aux informations fournies par le logiciel et les effets des transformations sont présentés sur des exemples.

# Chapitre 2

# Transformations pour une architecture définie

Ce chapitre aborde les transformations de programmes les plus usuelles. Dans ce but nous détaillons les relations de dépendances et les formalismes de représentations de ces dépendances permettant de construire des méthodes automatiques d'optimisation. La section 2.2.2 développe les critères d'optimisation liés à la taille mémoire et à la localité des calculs disponibles pour la gestion des hiérarchies de mémoires. Ces critères sont utilisés à la section 2.3 par les transformations de boucles existantes dans la littérature et permettant de modifier un programme en tenant compte d'une hiérarchie mémoire fixée à l'avance. Ils sont également utilisés dans le chapitre suivant. La méthodologie de transformation DTSE, présentée à la section 2.4 et dont l'automatisation d'une des phases constitue une partie du travail effectué pendant cette thèse[1], permet de combiner transformations de programme et synthèse de matériel. Les transformations présentées dans ce chapitre considèrent des architectures de système n'utilisant qu'une seule unité de calcul. On se trouve donc dans le cas où le partitionnement de la synthèse de système a déjà été effectué.

## 2.1 Analyse de programme et transformations

La compilation classique d'un programme est la traduction de sa description écrite dans un premier langage — le langage source — en un programme équivalent écrit dans un autre langage — le langage cible. Nous considérons dans cette thèse un mode de compilation particulier appelé transformations source à source, pour lequel le langage cible est le même que le langage source. Le travail du compilateur est donc de transformer le programme source en un programme optimisé selon différents critères.

Le premier type de transformations que nous allons aborder considère les optimisations utilisant les paramètres de l'architecture mémoire, comme présenté figure 2.1, afin d'adapter le comportement de l'application aux spécificités de la hiérarchie mémoire présente dans le système.

[1] Une collaboration avec l'équipe de Francky Catthoor a été mise en place pour ces travaux et a pue être financée grâce à un projet d'action intégrée (PAI) Tournesol.

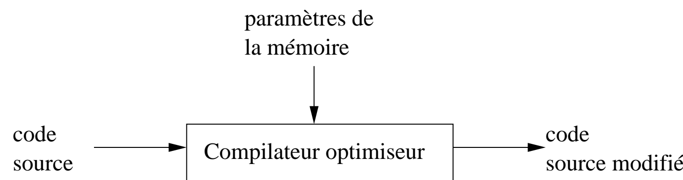



FIG. 2.1 – Transformations pour une architecture donnée

Le deuxième type de transformations modifie le comportement du programme concomitamment à la définition des paramètres de l'architecture mémoire, comme présenté figure 2.2. Le résultat de cette compilation est un couple matériel/logiciel fortement dédié à une application.

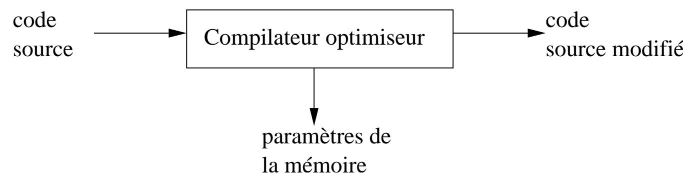



FIG. 2.2 – Transformations pour une architecture dédiée

Le lecteur pourra trouver une introduction à la compilation et à l'analyse de programme dans [ASU91]. Les transformations de programme que nous allons utiliser dans ce chapitre nécessitent toutes une analyse du programme permettant de déterminer les dépendances de données. C'est sur cette analyse des dépendances que reposent les *critères* d'optimisation ainsi que la *validité* des transformations mises en œuvre.

La prochaine section présente les relations de dépendances possibles entre des opérations ou instructions d'un programme et les représentations usuelles permettant de les manipuler sont présentées en section 2.1.2. La section 2.1.3 présente les considérations et limitations sur la forme des programmes pouvant être analysés et traités automatiquement.

### 2.1.1 Dépendances de données

Une dépendance de donnée existe entre deux opérations si elles utilisent un emplacement mémoire commun (une variable ou un élément de tableau), soit en lecture, soit en écriture. Dans les définitions qui suivent on considère que l'opération `O1` précède l'opération `O2` dans l'ordre donné par l'exécution séquentielle du programme où elles sont définies. Il existe quatre types de dépendances de données possibles entre deux opérations :

- *dépendance de flot* : l'opération `O2` est dépendante de l'opération `O1` par une dépendance de flot si `O2` utilise en lecture un emplacement mémoire écrit par `O1` et si aucune autre opération, dans l'ordre séquentiel, n'écrit à cet emplacement mémoire entre `O1` et `O2`.
- *anti-dépendance* : l'opération `O2` est dépendante de l'opération `O1` par une anti-dépendance si `O1` utilise en lecture un emplacement mémoire écrit par `O2` et si aucune autre opération, dans l'ordre séquentiel, n'écrit à cet emplacement mémoire entre `O1` et `O2`.
- *dépendance de sortie* : l'opération `O2` est dépendante de l'opération `O1` par une dépendance de sortie si `O2` et `O1` sont des écritures consécutives dans le même emplacement mémoire.

– *dépendance d'entrée* : les opérations `O1` et `O2` ont une dépendance d'entrée si `O2` et `O1` sont des lectures consécutives dans le même emplacement mémoire.

On note la relation de dépendance $\rightarrow$. La dépendance d'une opération `O2` par rapport à une opération `O1` sera notée `O1`$\rightarrow$`O2`.

Les trois premières dépendances donnent un ordre partiel, identifié par Bernstein, sur les opérations d'un programme (les dépendances d'entrée ne sont pas des dépendances directes et n'impliquent pas un ordre d'exécution sur les opérations). Un programme décrivant n'importe quel ordre d'exécution des opérations respectant l'ordre partiel défini par les dépendances de données directes est sémantiquement équivalent au programme de départ et fournit le même résultat final.

Toutes les transformations de programme utilisent donc la liberté fournie par cet ordre partiel pour réorganiser les calculs de façon à répondre aux critères d'optimisations traités. Par exemple, les techniques de parallélisation considèrent l'indépendance des opérations afin de les exécuter de manière concurrente ; les transformations utiles pour la gestion de la mémoire tentent de trouver un ordre partiel permettant d'améliorer l'utilisation des mémoires caches. Pour cela on améliore en priorité le traitement des boucles manipulant la mémoire de façon répétitive.

### Dépendances dans les boucles

Les boucles sont des structures de programmation permettant d'effectuer de manière répétitive des opérations. Nous ne donnerons ici qu'une définition intuitive des boucles et nous les représenterons sous la forme suivante correspondant à une syntaxe proche du langage C :

for(*valeur $m$ de départ pour $i$* ;
    *test de valeur finale $M$ pour $i$* ;
    *incrément $c$ pour $i$*)
{
    $S_1$
    ...
    $S_n$
}

Cette boucle est constituée de $n$ *instructions* et son indice $i$ parcourt toutes les valeurs entières allant de $m$ à $M$ par incréments de $c$ (*pas* de la boucle). Nous ne considérerons ici que des boucles sous cette forme. On considère de plus que les valeurs $m$,$M$,$c$ et $i$ ne peuvent pas être modifiées par une instruction appartenant au corps de la boucle. On appelle l'instance d'une instruction $S_k$ au pas $i$ de la boucle une *opération*, notée $(S_k, i)$. Le *domaine d'itération* d'une boucle est l'ensemble des valeurs que peut prendre la variable de boucle $i$.

$$D = \{i \in \mathbb{Z} | m \leq i \leq M \wedge \exists n \in \mathbb{N}, i = m + n * c\}.$$

Une boucle est dite *normalisée* lorsque $m = 1$ et $c = 1$. On a alors un espace d'itération de la forme :

$$D = \{i \in \mathbb{Z} | 1 \leq i \leq M\}.$$

Les boucles peuvent être récrites sous une forme normalisée par un changement de variable.

Les dépendances entre des opérations s'exécutant dans la même itération de boucle sont appelées des dépendances *indépendantes de la boucle* ; les dépendances entre des opérations s'exécutant dans différentes itérations sont appelées des dépendances *portées par la boucle*.

Une instruction présente dans une boucle peut, elle-même, être une boucle. On a alors des *nids de boucles* constitués de boucles imbriquées. Le domaine d'itération d'une instruction incluse dans un nid de boucles est défini par le produit cartésien des domaines d'itération de chaque boucle englobante. L'ordre de parcours de ce domaine est défini par un vecteur d'itération $I$ dont les composantes sont les indices de chaque boucle représentés de la boucle la plus externe vers la boucle la plus interne. Si toutes les instructions du nid de boucles sont entourées par les mêmes boucles, on dit que les boucles sont *parfaitement imbriquées* et que le nid de boucles est parfait.

Si on suppose que les bornes des boucles sont des fonctions affines des indices des boucles englobantes et des paramètres du programme, le domaine d'itération d'une instruction $S$ est un polytope (polyèdre fini) sur $\mathbb{Z}$ et peut être représenté sous forme matricielle

$$D(S) = \{x \in \mathbb{Z}^{n_S} \,|C_S.x \geq c_S\}$$

où $C_S$ et $c_S$ sont la matrice et le vecteur constant définissant les itérations du polytope et $n_S$ est la profondeur du nid de boucles englobant $S$. Dans le cas des nids de boucles parfaits, le domaine d'itération est le même pour toutes les instructions du nid de boucles.

### 2.1.2 Représentations usuelles des dépendances

Les algorithmes de transformations automatiques utilisent différents modèles de représentation des dépendances. Ces modèles servent à effectuer les transformations en utilisant des formalismes théoriques d'algèbre et/ou de théorie des graphes et permettent d'assurer la validité du code généré par le compilateur. Les définitions données ici le seront de manière informelle, le lecteur intéressé pourra trouver une étude plus formelle dans [Dar99] et [Fea91].

Le modèle le plus utilisé est une représentation sous forme de graphe pour laquelle les sommets peuvent représenter, selon la granularité choisie, les opérations, les instructions ou encore les nids de boucles d'un programme ; les arcs reliant les sommets représentent les dépendances de données ou de contrôle entre ces sommets. Bien que les applications multimédia comportent une part de contrôle importante, nous ne considérerons dans ce qui suit que les graphes de dépendances de données. En effet, nous nous concentrons sur les parties des applications manipulant de façon intensive les données par l'intermédiaire de boucles. Ces manipulations sont le plus souvent régulières et appliquées à l'ensemble des données, par exemple lors de l'application de filtres sur une image ou une donnée sonore ou encore lors de l'analyse d'un signal. Lorsque le code contient des opérations de contrôle, les dépendances de données seront surestimées en considérant que les différentes branches des tests sont prises tout le temps.

#### Graphe de dépendance développé

Un graphe de dépendance développé utilise l'ensemble des *opérations* comme ensemble de sommets. Les arcs entre les sommets représentent les dépendances entre les opérations. La représentation des dépendances est ici exacte mais nécessite la connaissance des bornes des boucles du programme et génère une description lourde à manipuler de par sa taille. En effet, les dépendances entre des

instructions d'un même nid de boucles sont représentées pour chaque instance de ces instructions. Il y a là une certaine redondance de l'information. Des représentations plus compactes sont possibles en utilisant des graphes de dépendances réduits.

```
for(i=1; i<N; i++)
  for(j=1; j<N-i+1; j++) {
    a[i][j] = a[i-1][j];
  }
```

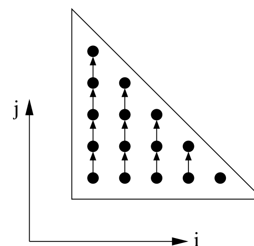



FIG. 2.3 – Graphe de dépendances développé (pour N=6)

### Graphe de dépendance réduit

Les graphes de dépendances réduits utilisent l'ensemble des *instructions* comme ensemble de sommets. Les arcs peuvent représenter les dépendances exactes lorsque cela est possible ou des approximations des dépendances entre les instructions. Ces approximations des dépendances sont le plus souvent modélisées de trois façons différentes :

**Vecteurs de distance** On appelle *distance* d'une dépendance $(S_k, i) \rightarrow (S_l, i')$ la quantité $i' - i$ d'itérations séparant les deux opérations. Lorsque cette quantité est constante et entière pour toutes les instances des opérations $S_k$ et $S_l$, on dit que la dépendance entre $S_k$ et $S_l$ est *uniforme*.

Les instructions d'un nid de boucles de profondeur $n$ sont indexées par un *vecteur d'itération* $I = (i_1, \ldots, i_n)$ où les $i_j$ représentent les compteurs de boucles en partant de la plus externe vers la plus interne. La distance entre deux opérations dans un nid de boucles est décrite par un vecteur de distance correspondant à la différence $I' - I$. Ce vecteur peut contenir des indices négatifs mais doit impérativement être lexico-positif (son premier terme non nul doit être positif) car une opération ne peut dépendre d'une opération future.

```
for(i=1; i<n; i++)
  for(j=1; j<n; j++) {
    S1: a[i,j]=c[i-1,j];
    S2: b[i,j]=a[i-1,j]+c[i-1,j];
    S3: c[i,j]=b[i,j];
  }
```

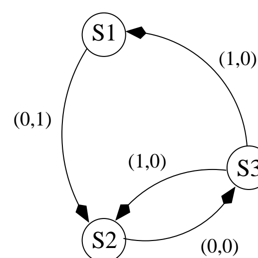



FIG. 2.4 – Graphe de dépendance réduit utilisant des vecteurs de distance

**Vecteurs de direction** Les vecteurs de direction sont une extension des vecteurs de distance. Lorsque les distances dépendent de la taille du domaine d'itération on note + les distances de

dépendances croissantes ; $-$ les distances de dépendances décroissantes (une distance peut être négative) et $*$ les distances de dépendances pouvant prendre n'importe quelle valeur.

**Approximations polyédriques** On peut aussi définir des approximations des vecteurs de distance par des polyèdres. Les relations de dépendance entre deux instructions $S$ et $T$ sont données par des relations affines de $\mathbb{Z}^{n_S} \rightarrow \mathbb{Z}^{n_T}$ où $n_S$ ($n_T$) est la dimension du domaine d'itération de $S$ ($T$). Cette représentation est exacte lorsque les domaines d'itérations sont représentés par des équations affines.

L'analyse de dépendance, dans un contexte général, est indécidable et on doit donc prendre en compte certaines restrictions sur les programmes que l'on peut analyser et donc transformer automatiquement.

### 2.1.3 Considérations sur les langages utilisés et la clarté des programmes

Les ouvrages disponibles en compilation logicielle comme [ASU91] ou [Wol96] présentent un nombre important de techniques d'optimisations de code mais celles-ci sont presque toujours exclusivement tournées vers la performance en vitesse d'exécution. Les besoins de stockage pour les tableaux de grande dimension n'étaient pas initialement perçus comme des coûts ou des problèmes. L'ordre de stockage des dimensions d'un tableau multidimensionnel n'étant même pas considéré comme un critère important pouvant être modifié par le compilateur, cet ordre est fixé dans la norme de nombreux langages comme C ou Fortran.

Le code d'origine ne convient pas toujours aux traitements automatisés et doit être modifié manuellement afin de permettre une vision à l'intérieur des fonctions. Un première phase (le *pruning*) consiste donc à modifier les descriptions en réalisant l'expansion des fonctions opérant sur des tableaux. Cette phase est aussi utile pour réorganiser les opérations de contrôle dans le code en les faire sortir au maximum du corps des boucles (induisant parfois des duplications de code) et en modifiant les opérations sur les scalaires pour les considérer comme des boites noires. Nous considérons ici que les bornes des boucles sont décrites par des relations affines ainsi que les dépendances entre les instructions.

Un autre problème rencontré dans l'optimisation statique de programme est la présence de mémoire allouée dynamiquement ainsi que la présence de pointeurs. Nous nous limiterons ici aux programmes n'utilisant pas ces deux particularités de programmation : les tableaux seront alloués statiquement (ce qui est déjà majoritairement le cas pour les systèmes embarqués) et ne seront pas accédés en utilisant de l'arithmétique de pointeurs.

## 2.2 Accès aux données et critères d'optimisation

Les accès à la mémoire sont beaucoup plus consommateurs d'énergie que les opérations de contrôle ou les opérations arithmétiques. Ainsi, des expérimentations [TMW94, NCK+96] ont, par exemple, montré qu'un accès à la mémoire externe peut coûter jusqu'à 33 fois plus en consommation qu'une addition sur 16 bits. De plus, plus l'accès mémoire est lointain, plus le coût est important : l'accès à une mémoire interne coûte 3 fois moins que l'accès à une mémoire externe au circuit.

Par ailleurs, des expérimentations manuelles de transformations, par exemple sur les algorithmes de compression/décompression vidéo du standard MPEG-4 [BCBM98], ont montré des gains non négligeables sur la mémoire. Ces transformations sur le code ont eu deux effets : réduire la taille de la mémoire et gagner en nombre de transferts de données (obtenant une réduction d'un facteur 65...). Les transformations sont principalement guidées par la localité des calculs afin de permettre une réutilisation maximale des données et éviter les transferts redondants vers la mémoire principale.

### 2.2.1 Place mémoire d'une application

Les variables scalaires et les tableaux peuvent être représentés par une taille et une durée de vie. La durée de vie d'une variable représente le temps entre la première écriture d'une donnée dans la variable et la dernière lecture de cette donnée. En dehors de sa durée de vie, une variable n'a pas de besoin d'exister en mémoire et la place qui lui sera assignée lorsqu'elle sera vivante peut être réutilisée par d'autres variables [De 98]. La détermination des durées de vie des variables est rendue possible dans le cadre de la synthèse d'architectures car les langages utilisés ne disposent pas de pointeurs. On peut donc faire l'hypothèse d'une réutilisation maximale de la mémoire. Une estimation de la taille mémoire requise par un programme peut donc être fournie par le maximum des tailles des variables vivantes à un temps $t$. Le premier problème pour cette fonction objective est donc celui de la mesure statique du temps dans un programme. Le critère de taille peut donc varier selon que l'on considère le temps écoulé par une instruction, l'itération d'une boucle ou une granularité plus élevée.

$$\text{Coût Mémoire} = \underset{\forall t}{max}\left\{ \sum_{v \in \text{Variables vivantes}(t)} taille(v) \right\}$$

La taille d'un tableau peut être calculée selon le nombre de ses dimensions, sa taille totale ou bien de manière plus précise en considérant le nombre de données effectivement stockées dans le tableau. Ces estimations donnent une borne minimale pour la quantité de mémoire nécessaire, la borne maximale étant obtenue en considérant un ordonnancement libre du programme avec un nombre infini de ressources (additionneurs, multiplieurs ...). L'ordonnancement libre d'un programme correspond à l'exécution au plus tôt (dès que les données sont disponibles) des instructions d'un programme.

### 2.2.2 Localité des accès

La localité des données peut être définie de plusieurs façons comme nous allons le voir. Cependant nous ne tiendrons pas compte de la différence entre les accès en lecture et les accès en écriture car cette différence implique des suppositions sur les choix qui devront être faits lors du partitionnement. En effet, l'utilisation de différents types de mémoires peut conduire à des résultats très éloignés allant d'un coût nul pour une opération de lecture ou d'écriture à une forte consommation d'énergie pour la même opération. Nous avons donc choisi de ne considérer la localité que sous ses formes spatiale (entrelacement) et temporelle (hiérarchie mémoire).

### Localité spatiale

La localité spatiale des données peut être définie comme la distance physique de deux accès consécutifs en mémoire. Par exemple, deux accès à un tableau `a`, un à l'adresse `a[i]` suivi d'un accès à l'adresse `a[i+k]` déterminent un enjambement (*stride*) de distance `k`. Cette localité a une influence sur la gestion des lignes de cache mémoire et sur les fautes de cache dues à des conflits pouvant survenir dans le cas des caches non totalement associatifs.

Plusieurs fonctions objectives peuvent être définies afin de prendre en compte l'ensemble $\{k_{b,t}\}$ des distances associées à chaque couple (boucle $b$, tableau $t$) d'un programme.

- On peut par exemple maximiser le nombre de distances $k$ inférieures ou égales à une borne donnée $K$ : $max \mid k_{b,t} \leq K \mid$.
- On peut aussi minimiser la valeur moyenne des distances d'un programme : $min(\sum_b k_{b,t})$.

Les modifications apportées ici ont une influence sur les méthodes de placement des données en mémoire [De 98, PNDN99]. Ces transformations seront présentées à la section 2.3.1.

### Localité temporelle

La localité temporelle est donnée par le temps écoulé entre deux accès successifs à la même adresse mémoire. Le premier problème pour cette fonction objective est donc celui de la mesure statique du temps dans un programme. Ce problème est rendu d'autant plus complexe lorsque la cible n'est pas encore générée.

L'évaluation du temps passé dans une boucle est facilité dans le contexte présent car les bornes des boucles des applications pouvant faire l'objet d'une synthèse conjointe sont souvent connues et fixées à la compilation (taille d'une image dans un programme vidéo, volume de données numériques audio dans les protocoles de transmission pour téléphones portables). Lorsque les bornes des boucles ne sont pas connues à la compilation le temps pris par une boucle doit alors être considéré comme infini.

Les fonctions objectives définissables pour ce type de problème sont similaires à celles utilisées pour la localité spatiale : étant donné un ensemble de localités temporelles, un concepteur peut avoir le choix entre

- borner les délais d'accès à un emplacement par un seuil, sachant que lorsque la contrainte ne peut pas être respectée le délai peut être dépassé.
- minimiser le temps entre deux accès successifs.

$$\sum_{(i',i) \in DoubleAccès} |i' - i|$$

où *DoubleAccès* est l'ensemble des dépendances correspondant à deux accès successifs à un même emplacement mémoire.

L'optimisation de la localité temporelle des accès est la base principale de la majorité des méthodes d'optimisation pour la gestion de la mémoire. Celles présentées à la section 2.3 mais aussi celles que nous présentons dans le prochain chapitre consacré aux optimisations pour une hiérarchie mémoire abstraite.

#### Localité de groupe

La localité de groupe représente l'union des deux critères précédents. Ce cas est intéressant lorsque l'on dispose d'assez de liberté pour manipuler simultanément les deux types de localité. On regroupe alors les accès par grappes correspondant à des transferts de blocs entre les niveaux de la hiérarchie mémoire. Les critères d'optimisation possibles sont :
- la minimisation de la différence de la taille des grappes par rapport à une valeur donnée (cette approche correspond à des tailles de blocs de cache fixes) ;
- la maximisation de la taille des grappes (correspondant à des tailles de blocs infinies).

## 2.3 Transformations pour une architecture matérielle fixe

Dans le cas d'une architecture fixe de nombreuses transformations peuvent être mises en place. Les recherches sur le sujet sont très nombreuses et une grande variété de transformations est disponible. La principale source de réduction de la consommation dans les hiérarchies mémoires fixes passe par une réduction des fautes de cache mais la modification des structures de données utilisées dans un programme [YCLV+99] est un exemple de transformation de très haut niveau permettant également de modifier les accès à la mémoire. À un niveau plus bas, les assignations de registre [ASU91, KNDK96] ainsi que la génération de code tenant compte des particularités du jeu d'instructions [TSR+98] permettent d'avoir une influence importante sur la consommation. Ces transformations ne seront pas développées dans le contexte des cette thèse et nous nous concentrerons sur les transformations de boucles et sur la gestion des tableaux en mémoire. La différence de performances entre calcul et accès à la mémoire est devenue tellement élevée que la minimisation des transferts devient un facteur parmi les plus importants. Ceci vaut pour la gestion des caches de données mais aussi pour le gestion des caches d'instructions [LMW99].

Nous nous limiterons à la gestion des mémoires caches par assignation des adresses en mémoire des tableaux [De 98] ainsi que par transformations de boucles [BGS94]. En effet, les mémoires caches fonctionnent selon un principe de localités spatiale et temporelle des accès. L'assignation des tableaux en mémoire est importante car elle joue un rôle essentiel dans les fautes de cache dues à des conflits d'adresses. La localité temporelle est, le plus souvent, générée dans les programmes par l'usage de boucles itératives. C'est le cas pour les applications multimédia dans lesquelles les données, représentées par des tableaux, sont manipulées par des boucles « for ».

### 2.3.1 Transformations pour organiser les données en mémoire

L'organisation des données en mémoire correspond à la génération des adresses réelles des éléments en mémoire. La modification du placement des variables en mémoire permet d'optimiser les transferts sur les bus de communications. Des études sur les différentes solutions disponibles pour la gestion de l'organisation des données en mémoire sont disponibles dans [PDN99b, De 98, BM00].

#### Entrelacement des données

L'idée ici est d'entrelacer les tableaux en mémoire pour minimiser les problèmes de défauts de cache survenant suite à des conflits d'adresses [De 98, PNDN97, PNDN99]. Chaque tableau est tout

d'abord découpé en blocs correspondant à une localité de groupe combinant la taille des lignes de cache et la localité spatiale des accès. Les blocs obtenus par ce découpage des tableaux sont répartis dans la mémoire de façon à minimiser le nombre de conflits d'adresses générés à l'exécution (conflits d'adresses dans un même tableau ou entre des tableaux différents). Les tableaux ne sont donc plus contigus en mémoire et cette technique complique le calcul des adresses réelles. L'entrelacement de tableaux est particulièrement adapté aux mémoires caches non totalement associatives et peut se combiner de manière très efficace avec la transformation de pavage de boucle décrite à la section 2.3.2 [PNDN99].

### Utilisation des différents modules de mémoire

L'utilisation explicite des modules mémoires [PDN99b] permet de segmenter un niveau de la hiérarchie en blocs de plus petite taille. Ceci a pour effet d'augmenter le nombre de ports d'entrées sorties disponibles pour un niveau et permet de réduire la complexité de la circuiterie et la consommation de chacun des modules. Le placement des variables ou tableaux dans des modules mémoires est plus complexe que pour un adressage linéaire simple mais permet au contrôleur de pouvoir accéder en parallèle aux différentes mémoires. On diminue ainsi le temps de latence et le nombre de requêtes que le contrôleur mémoire doit effectuer pour charger les données sur son bus externe. L'utilisation de différents modules mémoire comprend également la prise en compte des mémoires *scratch pad* présentées au chapitre 1, page 7.

## 2.3.2 Transformations de boucles

La maximisation de la localité des accès aux données par transformation de boucles s'effectue en utilisant dans la plupart des cas des fonctions de coût incluant les paramètres des mémoires caches (taille totale et taille des lignes). Les transformations utiles à une optimisation de la mémoire sont la *fusion* et la *distribution*, le découpage en sous-blocs appelé *pavage* ou *tiling*, le *décalage d'instructions*, l'*échange* (ou *permutation*) et l'*inversion*. Le lecteur pourra trouver dans [BGS94] une taxonomie détaillée des transformations de boucles.

### Fusion de boucles

La fusion est une transformation permettant de regrouper plusieurs boucles en une seule. L'augmentation de la localité temporelle est ici obtenue par rapprochement de plusieurs instructions manipulant un tableau dans le même corps de boucle [BGS94, MA97, BDSV97, MCT96]. Ceci permet d'avoir une bien meilleur réutilisation des données avec des distances de dépendances plus courtes car les emplacements mémoires sont accédés dans le même nid de boucles.

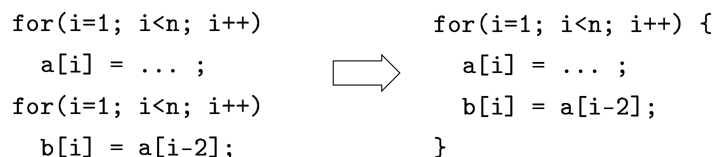



FIG. 2.5 – Exemple de fusion de boucles

La fusion de boucles est également utilisée pour permettre de réduire des tableaux à des variables scalaires (*array contraction*) [CK94, SG91, GOST92, De 98] permettant ainsi des gains en place mémoire. Le tableau $a$ de l'exemple de fusion donné figure 2.5 peut être réduit à une fenêtre de deux valeurs si le tableau n'est pas réutilisé plus loin dans le code. Cette transformation ainsi que la possible réduction de la taille des tableaux par fusion seront étudiées en détails dans le chapitre 3.

### Décalage d'instructions dans une boucle

Le décalage d'instructions est aussi connu sous le nom de *retiming* [LS91] ou de pipeline logiciel (*software pipelining*). Cette transformation consiste à définir une date d'ordonnancement (en nombre d'itérations) par rapport aux indices de la boucle. C'est donc un ordonnancement multidimensionnel dans le cas des nids de boucles.

Un exemple de décalage pour l'optimisation de la localité des calculs est représenté à la figure 2.6. Le code représenté à gauche nécessite deux variables temporaires pour stocker des valeurs réutilisées après seulement deux itérations. Le nombre de ces variables temporaires peut être minimisé si l'on aligne la boucle comme présenté à droite de l'exemple. Cette transformation est particulièrement utile pour les tableaux à plusieurs dimensions, on peut alors éviter de manipuler des sous-tableaux complets de variables temporaires.

```
for(i=1; i<n; i++) {
  a[i] = ... ;
  b[i] = a[i-2];
}
```

a b i

a b *prologue*

```
for(i=1; i<n-2; i++) {
  a[i] = ... ;
  b[i+2] = a[i];
}
```

i *épilogue*

FIG. 2.6 – Exemple de décalage de boucle

Le décalage d'instructions sera présenté plus en détails à la section 3.4 du prochain chapitre.

### Pavage de boucle

Le pavage de boucle permet de diviser l'espace d'itération d'une boucle en blocs [Car93, PNDN97, PNDN99, CM95]. Cette transformation permet d'avoir une segmentation de l'espace d'itération des nids de boucles en espaces plus petits et manipulant moins de données. La transformation augmente donc la profondeur des nids. Les possibilités de réutilisation de valeurs à l'intérieur d'un bloc sont ainsi améliorées car on peut diminuer de façon significative les conflits de capacité des caches et des bancs de registres en réduisant la quantité des données manipulées par les itérations. Le pavage est sans doute la transformation la plus dépendante des paramètres architecturaux de la hiérarchie mémoire et les principales difficultés sont les définitions des formes et des tailles des blocs en fonction de ces paramètres.

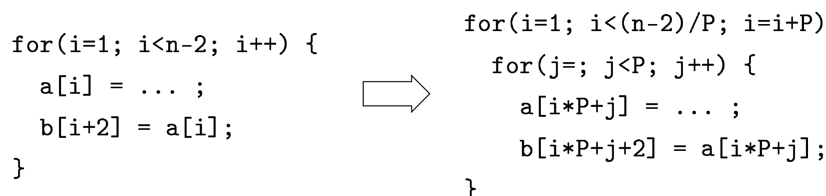



FIG. 2.7 – Exemple de pavage de boucle

### Distribution de boucle

La distribution de boucle est la transformation inverse de la fusion. Une boucle est ici séparée en plusieurs boucles afin de diminuer le nombre d'instructions et de transferts présents dans chacune d'entre elles. Cette transformation est utilisée lorsqu'une boucle manipule de manière indépendante de grandes quantités de tableaux créant des problèmes de capacité ou de conflit dans les caches.

### Échange de boucles

L'échange de boucles [Car93, MCT96] permet de changer le sens de parcours d'un tableau multidimensionnel. On peut ainsi échanger, lorsque les dépendances le permettent, une boucle externe avec une boucle plus interne. L'échange s'effectue selon le nombre de réutilisations potentielles dans une boucle et considère que les réutilisations contenues dans les boucles les plus internes d'un nid seront moins influencées par les problèmes de capacité de cache que les réutilisations effectuées dans les boucles externes. En effet, les dépendances portées par les boucles les plus externes impliquent une durée de vie plus longue pour les valeurs calculées que pour les dépendances portées par les boucles internes et donc un encombrement des niveaux supérieurs de la hiérarchie mémoire.

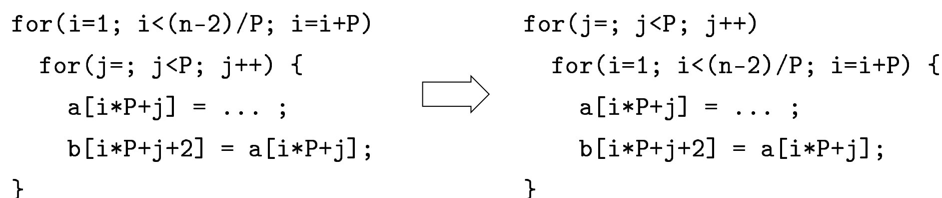



FIG. 2.8 – Exemple d'échange de boucles

### Inversion de boucle

L'inversion de boucle change le sens de parcours du domaine d'itération d'une boucle, le corps de la boucle ne doit donc pas contenir de dépendance portée par la boucle. L'inversion ne modifie pas les motifs d'accès à la mémoire et n'a que peu d'influence sur les transferts [MCT96]. Cependant, il est utile de considérer cette transformation car elle peut permettre l'utilisation d'autres transformations, notamment la fusion de boucles.

### Combinaisons de transformations de boucles

Chacune des transformations de boucles présentées ici permet, de manière individuelle, une amélioration de la localité d'un programme. Plusieurs recherches proposent des méthodes permettant de combiner ces transformations pour avoir une approche plus globale des transformations de boucles. Par exemple, la méthode proposée par Carr, McKinley et Tseng [Car93, MCT96] considère les combinaisons possibles de transformations utilisant la fusion, l'inversion, la permutation et la distribution. Les nids de boucles sont optimisés individuellement puis combinés avec les nids de boucles adjacents. La fonction utilisée pour calculer le coût d'une boucle dépend de la taille des lignes de cache et du nombre estimé de lignes manipulées lors de l'exécution d'une boucle. De même, Sarkar et Gao [SG91, GOST92], proposent une méthode permettant de combiner fusion, échange et inversion afin de pouvoir optimiser les communications par contraction de tableaux. Leur méthode est heuristique et limitée aux boucles simples ou à dimension 2.

Les méthodes de combinaisons de transformations de boucles proposées dans la littérature sont principalement conçues pour des problèmes de performances et non directement pour la localité des calculs et les transferts mémoire.

### Transformations utilisant les modèles polyédriques

La combinaison des transformations de boucles est un problème délicat à résoudre lorsqu'on utilise une modélisation des dépendances sous forme de graphe. Les méthodes utilisant les descriptions polyédriques [KP93, Len93, Fea92a, Fea92b, Fea96] se sont montrées très puissantes pour résoudre les problèmes de transformations de boucles ainsi que les combinaisons de transformations. Les problèmes sont posés sous forme d'équations linéaires et résolus par des programmes linéaires en nombres entiers (PLNE). Cependant, les transformations sont proposées, ici aussi, dans un contexte de parallélisation automatique et les formulations ne sont pas réutilisables directement aux problèmes d'optimisation des transferts et de gestion de la mémoire. Les programmes de parallélisation supposent, la plupart du temps, que la mémoire n'a pas de temps de latence ni de limite de bande passante car seule les opérations de calcul sont prises en compte pour les ordonnancements. De plus, les formulations en PLNE sont fortement contraintes par la taille des programmes à traiter [Fea92b], ce qui pose un problème pour les applications multimédia pouvant contenir des centaines de polyèdres, et ce, même après l'étape de *pruning*. La méthodologie proposée par IMEC utilise une description polyédrique pour laquelle la résolution est découpée en sous-problèmes afin de pouvoir être utilisée sur des applications de taille importante. Cette méthodologie est détaillée à la section 2.4 suivante.

## 2.4 Optimisation conjointe de l'architecture et du code : IMEC *Data Transfer and Storage Exploration – DTSE*

Il est intuitif que le meilleur moyen d'avoir de bons résultats d'optimisation sur la mémoire est d'optimiser conjointement la structure des calculs et l'architecture mémoire utilisée pour supporter ces calculs. Cette approche est activement développée par l'équipe de Francky Catthoor à IMEC (*Interuniversitair Micro-Elektronica Centrum*) en Belgique. La méthodologie proposée est découpée en étapes formant un flot appelé DTSE (*Data Transfer and Storage Exploration*) et les optimisations

sont spécialement conçues pour les applications embarquées dominées par les données comme les applications multimédia ou les protocoles réseaux évolués. Le principe de base considère que la consommation électrique globale des applications dominées par les données est principalement due aux transferts mémoire et au stockage comme nous l'avons vu tout au long des sections précédentes. La méthodologie, au départ conçue pour la création d'une architecture mémoire entièrement dédiée, a été étendue pour permettre la compilation vers une architecture prédéfinie.

La méthodologie DTSE est résumée à la figure 2.9, une description plus complète peut être trouvée dans [CWG+98]. Les optimisations mémoire présentées ici induisent le choix d'une spécification *applicative* suivant le modèle de programmation en assignation unique. Dans ce modèle, les éléments d'un tableau ne peuvent être modifiés une fois qu'une valeur leur a été assignée, les tableaux sont manipulés de façon abstraite et les pointeurs ne sont pas autorisés afin d'avoir une analyse de dépendance complète et fiable. L'entrée de la méthodologie est une spécification accédant aux tableaux multi-dimensionnels (appelés *signaux*) avec un processus unique de contrôle des calculs. Les sorties sont d'une part une spécification *exécutable* optimisée et d'autre part les paramètres complets de la hiérarchie mémoire avec la logique permettant la génération des adresses réelles des éléments de stockage.

L'étape de *pruning* ne fait pas partie de manière explicite de la méthodologie. Toutefois, cette phase est essentielle pour le bon déroulement des étapes suivantes de la méthodologie. Les modifications faites lors du pruning sont manuelles et permettent d'isoler les parties critiques des applications. Ces parties sont modifiées afin de pouvoir être transformées de manière automatique par les outils. Il s'agit d'un travail important demandant une très bonne connaissance de l'application que l'on souhaite compiler.

La première étape de la méthodologie (*Global date flow optimization*) applique des transformations globales sur le flot de données de la spécification. Ces transformations incluent les substitutions de signaux, les modifications de l'ordre des calculs en utilisant des propriétés d'associativité et de distributivité, ou la prise en compte de compromis entre espace mémoire et re-calcul de certaines valeurs. Ces transformations ne peuvent être automatisées et doivent être prises en compte par le concepteur de l'application.

Dans la deuxième étape (*Global loop and control-flow optimization*), la description est optimisée afin de réduire la durée de vie globale des signaux et d'augmenter la localité et la régularité des calculs. La représentation utilise un modèle polyédral et les transformations sont découpées en plusieurs étapes. Une première phase, le *placement*, permet de placer les polytopes dans un espace d'itération abstrait sans définition de notion de temps. La deuxième étape, l'*ordonnancement*, définit un ensemble de vecteurs d'ordonnancement permettant de séquencer les opérations. Ces deux étapes, dont celle de l'ordonnancement faisant partie des travaux effectués pendant cette thèse, seront détaillées dans la prochaine section.

Les deux premières étapes opèrent uniquement sur la spécification. La troisième étape (*Data reuse decisions*) met en place une ébauche de la hiérarchie mémoire. Les transferts mémoire générés par la spécification sont analysés, groupés et répartis en niveaux hiérarchiques. Le but principal de cette étape est de trouver un compromis entre l'espace supplémentaire généré par les duplications de données dans des variables temporaires ayant une fréquence d'accès élevée et une allocation de l'espace mémoire pour ces données dans les niveaux supérieurs de la hiérarchie.

Une fois la hiérarchie mémoire définie par son nombre de niveaux et par l'allocation des données dans les niveaux, le flot DTSE affine cette hiérarchie mémoire en minimisant son coût tout en garantissant les performances. L'analyse des contraintes de temps et la distribution des accès à

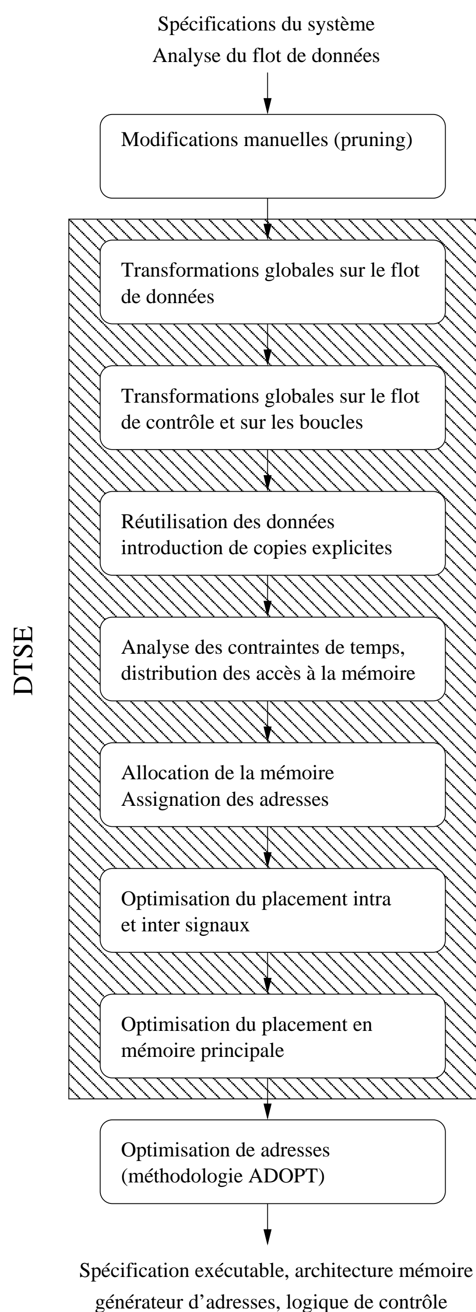



FIG. 2.9 – Flot d'optimisation simplifié de la méthodologie DTSE

la mémoire (*Storage cycle budget distribution*) permet d'ordonnancer les accès en lecture et en écriture au niveau des opérations d'une boucle afin de minimiser le nombre de modules et de ports d'entrée/sortie requis pour des accès simultanés à la mémoire tout en respectant les contraintes de temps d'exécution fixées pour l'application.

La quatrième étape d'allocation et d'assignation (*Memory allocation and assignment*) alloue un type de mémoire identifié dans une bibliothèque à chaque tableau, en fonction des contraintes de bande passante et de nombre de ports d'entrée/sortie. Cette étape détermine l'organisation de la mémoire externe d'une manière optimale. Dans le cadre de l'utilisation de processeurs programmables et d'une architecture mémoire fixée, cette étape permet d'utiliser au mieux la hiérarchie mémoire externe au processeur.

La sixième étape (*In-place optimization*) permet de trouver un placement optimal des données à l'intérieur des modules de façon à minimiser la taille mémoire requise pour chacun. Les transformations principales concernent le repliage d'un tableau sur lui-même ou le partage d'espace physique entre plusieurs tableaux. Lorsque cette étape est utilisée pour une architecture mémoire fixe, le placement permet de réduire les conflits de capacité pour les mémoires caches.

Enfin, la dernière étape de la méthodologie utilise l'entrelacement de tableaux présenté à la section 2.3.1 (page 47) pour réduire les conflits d'adresses dans les caches. Cette étape n'est utile que dans le cas d'une utilisation de processeurs programmables et de mémoires caches.

Une méthodologie séparée, appelée ADOPT (*ADdress OPTimization*) [MCJM98], également développée à IMEC, est utilisée pour simplifier les calculs d'adresses. En effet, les nombreuses étapes de la méthodologie DTSE augmentent la complexité des bornes de boucles, des indexes de tableaux et des conditions utilisées pour le contrôle. Si ces surcoûts sont négligés, le système (bien que moins consommateur d'énergie) subit une dégradation importante des performances. La simplification des adresses et des expressions peut être effectuée en utilisant également des transformations source à source de programme et/ou en générant un circuit spécifique chargé de la gestion des adresses.

Comme pour les autres modèles de transformation présentés, nous nous concentrons ici sur l'étape de transformations de boucles, appelée *Global loop and control-flow optimization* de le contexte de la méthodologie DTSE.

### 2.4.1 Transformations de boucles et de contrôle pour la DTSE

Afin d'appliquer des optimisations mémoire au niveau système, toutes les fonctions de la description initiale doivent être regroupées en une seule fonction. Le fonction ainsi obtenue comprend toutes les boucles du programme et permet d'effectuer des transformations globales ayant un large impact sur les coûts reliés aux accès mémoire.

Comme les transformations de boucles se placent au début de la méthodologie elles devraient, en principe, utiliser des estimations de haut niveau pour toutes les étapes suivantes. Il a pu heureusement être montré que certaines étapes comme la distribution temporelle des accès à la mémoire, l'allocation et l'assignation ainsi que l'organisation de la mémoire externe n'ont que des intéractions faibles avec les choix effectués ici. Ceci est principalement dû au fait qu'améliorer la localité et la régularité ne peut qu'augmenter les possibilités d'optimisation pour ces étapes ultérieures.

La méthode proposée ici pour effectuer les transformations de boucles et de contrôle utilise une représentation des dépendances polyédrique similaire à celles déjà présentées page 44. Afin de surmonter la complexité de résolution de ce genre de problème, les transformations de boucles sont

faites en plusieurs étapes. Lors de la première phase, le *placement*, les polytopes sont placés dans un espace d'itération commun sans utiliser de notion de temps et d'ordonnancement. La seconde phase, l'*ordonnancement*, permet de définir un ensemble de vecteurs d'ordonnancement permettant de séquencer les opérations de cet espace d'itération commun. La placement est également séparé en deux sous-étapes. Un des avantages majeurs de cette méthode est que chacune des phases est moins complexe que le problème de départ, un autre avantage étant la possibilité d'utiliser des critères différents dans chaque étape. Néanmoins, la segmentation de l'espace de recherche introduit le risque de ne considérer que des minima locaux et de ne pas trouver un optimum global pour le problème.

Bien que l'étape de placement constitue un travail déjà publié dans la thèse de Koen Danckaert [Dan01], nous le présentons de manière succincte afin de comprendre le contexte de l'étape d'ordonnancement réalisée pendant cette thèse et présentée à la section 2.4.3. Des expérimentations effectuées sur des applications de taille raisonnable sont présentées à la section 2.4.6.

### 2.4.2 Placement géométrique

La représentation utilisée est un graphe de dépendance polyédrique (*Polyhedral Dependency Graph – PDG*) [De 98]. Cette représentation est une extension du modèle utilisé en parallélisation automatique [Fea96]. Dans ce modèle, un nid de boucles de profondeur $n$ est représenté par un polytope de dimension $n$ dans lequel les points de coordonnées entières représentent les opérations (instances des instructions) du nid de boucles. La taille des polytopes dans chaque dimension est déterminée par les bornes des boucles. Les dépendances entre les opérations sont représentées de manière exacte si les expressions d'indexation sont affines. Bien que ce modèle soit proche de la représentation sous forme de graphe étendu, il possède une représentation mathématique compacte.

Le PDG utilise le modèle des polytopes comme base mais n'est pas limité à un seul nid de boucles. Chaque nid de boucles est représenté par un polytope et les dépendances sont représentées par des relations affines entre les polytopes. Le PDG est un graphe orienté pour lequel les nœuds représentent les nids de boucles et les arcs les dépendances de données. Durant la phase de placement, les polytopes sont placés dans un espace d'itération commun par des transformations affines. Comme aucune information sur l'ordonnancement n'est disponible durant cette étape, une fonction objective utilisant une notion de temps n'est pas utilisable : seules des transformations géométriques sont appliquées sur les polytopes. Le principal objectif de cette étape est de rendre les dépendances les plus courtes et les plus régulières possible géométriquement. Le placement selon [Dan01] est donc découpé en deux sous-étapes : l'une optimisant la régularité, l'autre la distance.

#### Critères utilisés pour les transformations de placement

La première sous-étape de placement se concentre sur la régularité des dépendances. L'optimisation de la régularité dans le modèle polyédrique se traduit par la minimisation de la variance de la direction des vecteurs de dépendance : les dépendances doivent être les plus régulières possible. La définition d'un *cône de dépendance* est utilisée pour exprimer la fonction de coût. Ce cône est celui généré par les vecteurs de dépendance. Comme un cône peut être représenté par ses rayons extrémaux, le cône des dépendances est une bonne représentation des vecteurs de dépendance extrêmes. Le cône de dépendance est une représentation suffisante pour connaître l'ensemble des vecteurs d'ordonnancement valides [AIY95].

La fonction de coût pour la régularité des calculs est exprimée en utilisant la *finesse* du cône de dépendance. Un placement avec un cône de dépendance plus fin que celui des autres placements possibles permet d'avoir une liberté plus importante pour les choix ultérieurs concernant l'ordonnancement et conduit généralement à des solutions plus efficaces. Intuitivement, plus le cône de dépendance est fin, plus le cône contenant les vecteurs possibles pour l'ordonnancement est ouvert et contient des solutions, comme le présente la figure 2.10 pour un espace à deux dimensions.

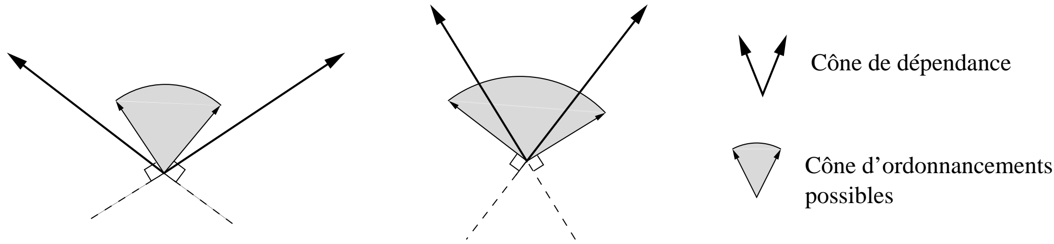



FIG. 2.10 – Cône de dépendance et cône d'ordonnancements possibles

La figure 2.11 montre un exemple simple utilisé pour illustrer cette section.

```
A: for(i=1; i<N; i++)
     for(j=1; j<N-i+1; j++)
        a[i][j] = in[i][j] + a[i-1][j];
B: for(p=1; p<N; p++)
     b[p][1] = f(a[N-p+1][p], a[N-p][p]);
C: for(k=1; k<N; k++)
     for(l=1; l<k; l++)
       b[k][l+1] = g(b[k][l]);
```

FIG. 2.11 – Code de la spécification initiale

Un placement des polytopes dans un espace d'itération commun est obtenu à la figure 2.12 pour le code de la figure 2.11. Le cône de dépendance total peut être calculé et sa finesse (l'angle d'ouverture pour un cône en deux dimensions) est égal à $\pi/2$.

Une fois la régularité optimisée, la deuxième sous-étape du placement effectue des opérations de translations sur les polytopes pour obtenir un placement final présentant un fort potentiel de localité des calculs. Cette étape est guidée par une fonction objective servant à minimiser la longueur moyenne des dépendances. Pour l'exemple présenté, il est possible de rendre le cône de dépendance nul et tous les vecteurs de dépendance uniformes comme le montre la figure 2.13.

Le placement propose donc une description optimisée pour la localité et la régularité des calculs. Les transformations sont effectuées de façon géométrique et sans notion de temps, il faut donc recréer un ordonnancement pour retrouver un code correct.

### 2.4.3 Méthode constructive pour l'ordonnancement

Une fois le placement effectué et optimisé dans un espace d'itération global, nous devons donner un ordonnancement pour toutes les opérations afin de compléter les transformations de boucles.

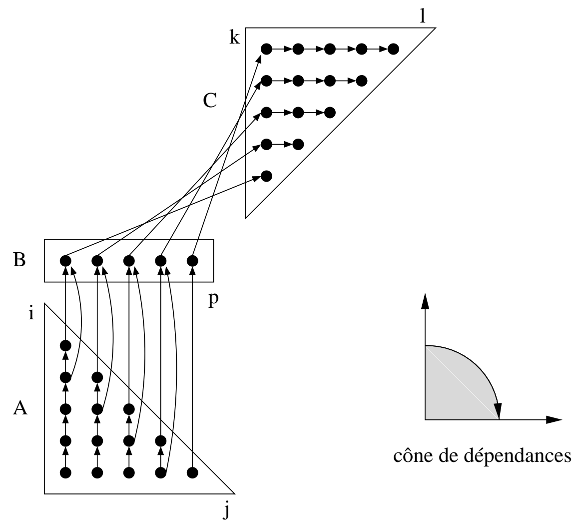



FIG. 2.12 – Espace d'itération commun initial du code de la figure 2.11 (N=5)

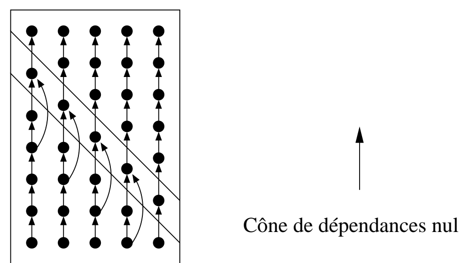



FIG. 2.13 – Espace d'itération commun final

Cet ordonnancement doit être compatible avec le placement et prendre en compte les contraintes créées par les transformations géométriques utilisées. Il détermine la date, donc l'itération, de chaque opération.

La méthode que nous proposons dans cette thèse est une méthode heuristique permettant de construire un ordonnancement de manière *incrémentale*. Au lieu de trouver un vecteur d'ordonnancement affine global, nous définissons un parcours de l'espace d'itération utilisant des *boucles imbriquées* minimisant le longueur moyenne des dépendances pour chaque niveau de boucle. Une optimisation incrémentale permet de prendre en compte différentes fonctions objectives lors de l'élaboration de l'ordonnancement et ainsi de mieux maîtriser les compromis possibles entre optimisation pour la mémoire et pour les performances de calcul. En effet, les optimisations pour la minimisation de l'utilisation de la mémoire ont la plupart du temps des effets contraires aux optimisations pour le parallélisme et la performance. Trouver un ordonnancement optimal pour la minimisation de la localité des calculs peut bloquer la détection et l'utilisation du parallélisme dans les étapes suivantes de la compilation. Nous allons uniquement traiter ici le problème d'optimisation de la localité mais la méthode est suffisamment générale pour permettre d'utiliser d'autres critères d'optimisations comme le parallélisme.

Soit $n$ la dimension de l'espace d'itération commun défini à la section 2.4.2 et $D$ le multi-ensemble comprenant *tous* les vecteurs de dépendance. L'efficacité de notre algorithme repose sur un parcours de l'espace utilisant les vecteurs de la base canonique $B = \left\{\vec{i_1}, \vec{i_2}, \ldots, \vec{i_n}\right\}$ de l'espace d'itération global comme axes de parcours. Les directions de parcours possibles sont donc données par l'ensemble de vecteurs $I$ tel que :

$$I = \left\{\vec{i_1}, \vec{i_2}, \ldots, \vec{i_n}, -\vec{i_1}, -\vec{i_2}, \ldots, -\vec{i_n}\right\}. \qquad (2.1)$$

Par exemple, pour un espace en deux dimensions nous pouvons définir dans les représentations usuelles $B = \left\{\vec{x}, \vec{y}\right\}$ comme axes et $I = \left\{\vec{x}, \vec{y}, -\vec{x}, -\vec{y}\right\}$ comme directions de parcours.

La figure 2.14 montre une solution possible au problème présenté pour le placement page 57. Cette solution est trouvée en choisissant $v_1$ dans la même direction que les dépendances. Ce choix implique le calcul des points contenus dans les rectangles pointillés par la boucle la plus interne. On peut voir sur cet exemple que nous avons besoin au maximum de $N+3$ variables temporaires entre chaque itération de la boucle externe à partir de la troisième itération.

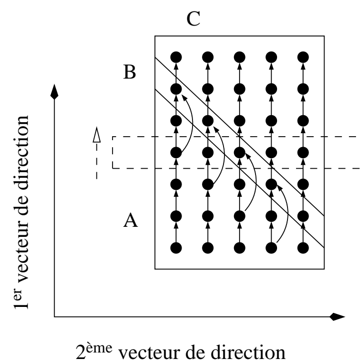



FIG. 2.14 – Mauvais choix pour l'ordre des vecteurs

**Fonction de coût pour la localité des calculs**

Les dépendances de l'ensemble $D$ portées par les boucles externes sont les plus coûteuses car elles impliquent une durée de vie plus importante pour les valeurs produites et consommées lors des différentes itérations. Nous considérons donc les boucles les plus externes en premier et construisons le nid de la boucle la plus externe vers la boucle la plus interne.

Pour chaque niveau d'imbrication $l$, nous considérons l'ensemble $D_l$ des dépendances non encore portées par les boucles englobantes (de niveau $l' < l$). Une dépendance portée par une boucle de niveau $l$ peut être enlevée de l'ensemble $D_{l+1}$ des dépendances restantes. Pour chaque niveau $l$, nous voulons minimiser la somme des longueurs des dépendances portées par la boucle $l$ de manière à faire porter les dépendances par les boucles les plus internes pour lesquelles la localité est plus élevée.

La longueur des dépendances est calculée en utilisant le produit scalaire de chaque vecteur de dépendance avec le vecteur de direction $\overrightarrow{v_l} \in I$ choisi pour une boucle $l$. Un produit scalaire $\overrightarrow{d}.\overrightarrow{v_l}$ représente la longueur de la projection du vecteur de dépendance $\overrightarrow{d}$ sur le vecteur unitaire $\overrightarrow{v_l}$. Une dépendance est portée par une boucle si ce produit scalaire est strictement supérieur à zéro.

Afin de ne sélectionner que des directions d'ordonnancement *légale* pour une boucle $l$, le vecteur $\overrightarrow{v_l}$ doit vérifier la contrainte suivante :

$$\overrightarrow{d}.\overrightarrow{v_l} \geq 0, \ \forall \overrightarrow{d} \in D_l \tag{2.2}$$

La direction d'ordonnancement que nous utilisons pour un niveau de boucle est celle minimisant la somme des longueurs de dépendance portées par cette boucle. Pour chaque niveau $l$ nous voulons trouver une direction $\overrightarrow{v_l} \in I$ minimisant la fonction suivante :

$$f(\overrightarrow{v}, l) = \sum_{\overrightarrow{d} \in D_l} \overrightarrow{d}.\overrightarrow{v} \tag{2.3}$$

Chaque pas de l'algorithme produit un hyperplan, représenté par un rectangle en pointillés dans les exemples de la section suivante, ainsi qu'une direction qui sera utilisée pour parcourir l'espace d'itération. Des opérations peuvent ne pas être ordonnancées en utilisant cet hyperplan (celles ayant une dépendance dont le vecteur a un produit scalaire égal à zéro avec le vecteur d'ordonnancement) et dans ce cas il faut augmenter la profondeur du nid de boucles et choisir un vecteur d'ordonnancement supplémentaire. Le processus de construction des vecteurs est donc itératif et nous pouvons définir l'algorithme de la façon suivante :

**Algorithme 1** (*Ordonnancement selon les axes canoniques*)

**ordonnancement** $(I,D)$
début
$I_1 \leftarrow I$
$D_1 \leftarrow D$
pour $l = 1$ à $n$ faire
début
sélectionner $\vec{v_l} \in I_l$ tel que $f(\vec{v_l}, l)$ est minimisée
$I_{l+1} \leftarrow I_l \setminus \left\{\vec{v_l}, -\vec{v_l}\right\}$
$D_{l+1} \leftarrow \left\{\vec{d} \in D_l \middle| \vec{d}.\vec{v_l} = 0\right\}$
fin
fin

**Complexité :** Chaque étape $l$ de notre algorithme doit calculer $|I_l| * |D_l|$ produits scalaires. Le nombre total d'opérations est égal à

$$\sum_{l=1}^{n}(2(n-l)) * |D_l| * n \leq 2n|D|\sum_{l=1}^{n} l.$$

La complexité globale de notre algorithme est donc $O(|D|n^3)$.

Si nous choisissons $v_1$ de façon à minimiser son produit scalaire avec les dépendances, comme indiqué sur la figure 2.15 alors nous n'avons besoin que de deux variables temporaires pour accomplir toutes les itérations de l'espace. La boucle externe calcule à chaque itération les opérations incluses dans le rectangle en pointillé. La boucle la plus interne requiert au plus 2 variables temporaires utilisées lors du passage du calcul des opérations du polytope A aux opérations du polytope B. Cet ordonnancement est clairement le meilleur ordonnancement possible pour cet exemple.

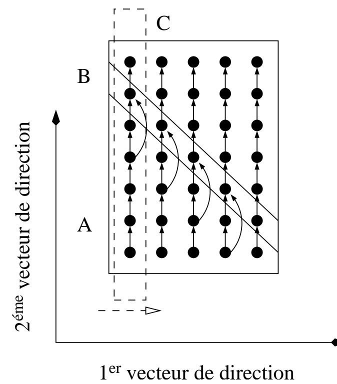



FIG. 2.15 – Meilleur choix possible d'ordonnancement

Le figure 2.16 montre le code généré par l'ordonnancement choisi à la figure 2.15. La génération de code est facilitée par la méthode utilisée pour la construction de l'ordonnancement car le parcours de l'espace est effectué selon un ordre pouvant être décrit par un nid de boucles dont les bornes peuvent être calculées à partir des équations des polytopes. L'heuristique proposée ici est polynômiale et

```
for(j=1; j<=N; j++) {
 for(i=1; i<=N-j+1; i++)
   a[i][j] = in[i][j] + a[i-1][j];
 b[j][1] = f(a[N-j+1][j], a[N-j][j]);
 for(l=1; l<=j; l++)
   b[j][l+1] = g(b[j][l]);
}
```

FIG. 2.16 – Génération du code pour la figure 2.15

permet déjà d'avoir de très bons résultats, comme le montrent les expérimentations présentées à la section 2.4.6. Cependant, elle ne permet pas toujours de trouver un ordonnancement valide et doit être généralisée comme présenté dans la prochaine section.

### 2.4.4 Extension de l'heuristique proposée

L'heuristique que nous proposons peut ne pas trouver de résultat sur certains exemples alors qu'un vecteur d'ordonnancement existe. L'exemple de la figure 2.17 montre un polytope (où seules 2 dépendances sont représentées) pour lequel aucun des vecteurs de l'ensemble $\left\{\vec{i_1}, \vec{i_2}, -\vec{i_1}, -\vec{i_2}\right\}$ ne peut être utilisé comme vecteur d'ordonnancement car l'équation 2.2 ne peut être satisfaite. Pourtant un tel vecteur existe $(\vec{c})$.

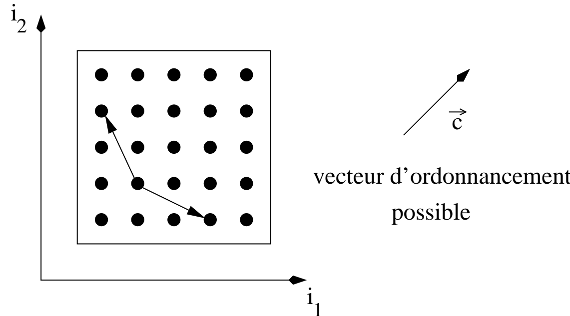



FIG. 2.17 – Exemple d'échec pour l'heuristique simple

Le problème provient de la restriction importante posée sur le choix des vecteurs d'ordonnancement. Il est possible de calculer une base de vecteurs assurant de trouver un vecteur d'ordonnancement si celui-ci existe. Ce calcul est beaucoup plus coûteux que celui de l'heuristique donnée à la section précédente.

Nous présentons ici un moyen de trouver un vecteur d'ordonnancement $\vec{v_l}$ pour chaque niveau de boucle $l$. La restriction de départ sur l'ensemble des directions possibles $I_l$ est levée. Formellement, nous voulons définir un vecteur $\vec{v_l}$ satisfaisant le système $\mathcal{S}$ :

$$\vec{d}.\vec{v_l} \geq 0 \quad \forall \vec{d} \in D_l \tag{2.4}$$

$$||\vec{v_l}|| = 1 \tag{2.5}$$

$$f(\vec{v_l}, l) \text{ est minimal.} \tag{2.6}$$

Les deux premières équations définissent le domaine dans lequel $\vec{v_l}$ doit se trouver. Elles remplacent l'ensemble $I_l$ du premier algorithme. L'équation (2.4) assure que le résultat de l'ordonnancement est valide et l'équation (2.5) assure que la norme de $\vec{v_l}$ est 1 (seule la direction de $\vec{v_l}$ est importante). La dernière équation définit la fonction de coût qui est la même que pour la première méthode.

Soit $C_l$ le cône d'ordonnancement au niveau $l$ défini par les vecteurs vérifiant l'équation (2.4) et soient $d_{e1}, \cdots, d_{ek}$ les vecteurs de dépendance extrémaux de $D_l$ (i.e., chaque vecteur de dépendance est une combinaison linéaire positive de ces vecteurs). Nous avons :

$$C_l = \left\{ \vec{\theta} \text{tel que} \vec{\theta} . \vec{d} \geq 0 \quad \forall d \in D_l \right\}$$

pour l'ensemble des vecteurs de dépendance de $D_l$ d'où

$$C_l = \left\{ \vec{\theta} \text{tel que} \vec{\theta} . \vec{d_{ei}} \geq 0 \quad \forall i \leq k \right\}$$

pour les vecteurs extrémaux uniquement tirés de la phase de placement.

Une solution du système $\mathcal{S}$ est donnée par la proposition suivante :

**Proposition 1** *Si* $\dim(\text{Vect}(D_l)) \leq n - l$ *alors n'importe quel vecteur unitaire d'une base orthogonale de* $\text{Vect}(D_l)^{\perp}$ *est une solution. Si* $\dim(\text{Vect}(D_l)) = n - l + 1$ *alors un des vecteurs unitaires extrémaux de* $C_l$ *est une solution.*

**Preuve** Si $\dim(\text{Vect}(D_l)) \leq n - l$, la proposition est trivialement vérifiée. Si $\dim(\text{Vect}(D_l)) = n - l + 1$, il suffit de montrer que si $\vec{a}$ et $\vec{b}$ sont deux vecteurs unitaires du cône d'ordonnancement $C_l$ alors le coût de tout vecteur unitaire compris dans la portion de cône décrite par $(\vec{a}, \vec{b})$ est supérieur au coût de $\vec{a}$ et au coût de $\vec{b}$ :

$$\forall \alpha \in [0; 1], f\left( \frac{\vec{a} + \alpha(\vec{b} - \vec{a})}{||\vec{a} + \alpha(\vec{b} - \vec{a})||}, l \right) \geq \min\left( f(\vec{a}, l), f(\vec{b}, l) \right)$$

où $f(\vec{x}, l)$ est de la forme $\vec{c_l} . \vec{x}$ avec $\vec{x}$ dans $C_l$ (équation 2.3).

Comme $||\vec{a}|| = ||\vec{b}|| = 1$ et par convexité de la norme, $||\vec{a} + \alpha(\vec{b} - \vec{a})|| \leq 1$. De plus, nous avons par définition du cône d'ordonnancement $\vec{c_l} \cdot \vec{b} \geq 0$ et $\vec{c_l} \cdot \vec{a} \geq 0$. Donc $\forall \alpha \in [0; 1]$

$$\begin{aligned}
f\left( \frac{\vec{a}+\alpha(\vec{b}-\vec{a})}{||\vec{a}+\alpha(\vec{b}-\vec{a})||}, l \right) &= \frac{\vec{c_l} \cdot \vec{a} + \alpha \vec{c_l} \cdot (\vec{b} - \vec{a})}{||\vec{a}+\alpha(\vec{b}-\vec{a})||} \\
&\geq \begin{cases} \vec{c_l} \cdot \vec{a} + \alpha \vec{c_l} \cdot (\vec{b} - \vec{a}) \geq f(\vec{a}, l), & \text{si } \vec{c_l} \cdot (\vec{b} - \vec{a}) \geq \vec{0} \\ \vec{c_l} \cdot \vec{b} + (1-\alpha) \vec{c_l} \cdot (\vec{a} - \vec{b}) \geq f(\vec{b}, l), & \text{sinon} \end{cases} \\
&\geq \min\left( f(\vec{a}, l), f(\vec{b}, l) \right)
\end{aligned}$$

□

Afin de trouver une solution, l'algorithme doit énumérer l'ensemble des vecteurs extrémaux de $C_l$ et en choisir un minimisant la fonction de coût.

**Proposition 2** *Si l'espace d'itération* $n-l+1$ *de* $\text{Vect}(D_l)$ *est de dimension 2 alors un sur-ensemble des vecteurs extrémaux de* $C_l$ *est trouvé en considérant les vecteurs orthogonaux aux vecteurs de dépendance. Si* $n-l+1$ *est supérieur à deux, alors un sur-ensemble des vecteurs extrémaux de* $C_l$ *est trouvé en considérant les vecteurs produits des vecteurs de dépendance, i.e. l'ensemble :*

$$P = \left\{ \vec{d}_1 \times \vec{d}_2 \times \cdots \times \vec{d}_{n-l} \text{ où } \vec{d}_1, \vec{d}_2, \ldots, \vec{d}_{n-l} \in D \right\}$$

**Preuve** La proposition est aisément vérifiée pour $n-l+1=2$. Si $n-l+1$ est supérieur à 2 alors les faces du cône d'ordonnancement $C_l$ sont constituées d'hyperplans orthogonaux aux vecteurs de dépendance (voir figure 2.10 pour un exemple). Les vecteurs extrémaux de ce cône sont des intersections de $n-l$ faces. Donc les directions des vecteurs extrémaux sont données par des produits vectoriels de vecteurs orthogonaux aux faces, i.e. les vecteurs de dépendance. □

Remarque : tous ces produits ne sont pas des vecteurs extrémaux de $C_l$, par contre tous les vecteurs extrémaux de $C_l$ sont inclus dans l'ensemble de ces produits ($P$).

Si les vecteurs extrémaux du cône de dépendance sont connus, nous pouvons remplacer $D$ dans la définition de $P$ par l'ensemble des vecteurs de dépendance extrémaux. Les vecteurs extrémaux peuvent être calculés si les dépendances sont affines.

**Algorithme 2** (*Ordonnancement étendu*)

> **ordonnancement** $(I,D)$
> début
>   Si $dim(\text{Vect}(D)) = m < n$ alors
>     sélectionner $n-m$ éléments orthogonaux de la base orthogonale de $\text{Vect}(D)^{\perp}$
>     (ces $n-m$ vecteurs donnent les premiers vecteurs de directions $v_1, \ldots, v_{n-m}$).
>   $D_{n-m+1} \leftarrow D$
>   pour $i = n-m+1$ à $n$ faire
>   début
>     sélectionner $\vec{v_i}$, un des vecteurs extrémaux du cône d'ordonnancement
>       généré par $D_i$ tel qu'il minimise $f(\vec{v_i}, i)$
>     $D_{i+1} = \left\{ \vec{d} \in D_i \middle| \vec{d}.\vec{v_i} = 0 \right\}$
>   fin
> fin

Chacune des $n$ étapes nécessite le calcul de $|D|^{n-1}$ produits vectoriels différents qui peuvent chacun être effectué en $n^3$ étapes. Comme pour l'algorithme 1, le calcul de la fonction de coût nécessite le calcul de $|D|$ produits scalaires effectués en $n$ étapes. La complexité totale de l'algorithme

est donc $O(n.(n^3.|D|^{n-1}.|D|.n)) = O(n^5|D|^n)$ où $n$ est la dimension de l'espace d'itération commun. Nous n'avons pas rencontré à ce jour d'exemple contenant plus de 8 boucles imbriquées. La prise en compte du nombre total de dépendances rend clairement cet algorithme inutilisable en pratique.

### 2.4.5 Réduction de la complexité

La complexité des deux méthodes dépend de $|D|$. Ce facteur peut être réduit en utilisant seulement des vecteurs représentatifs de chaque type de dépendance. L'information permettant d'identifier ces vecteurs doit être fournie par la phase de placement. Par l'utilisation de groupes de dépendances, la complexité de la première méthode peut être réduite à $O(|D'|n^3)$ et la complexité de la seconde peut être réduite à $O(n^5|D'|^n)$ où $D'$ est l'ensemble des dépendances différentes ou des dépendances extrémales (si elles sont connues) dans l'espace d'itération global.

Pour l'exemple de la figure 2.13 nous avons $|D| = 34$ et $|D'| = 2$. La réduction du nombre de dépendances prises en compte dans les algorithmes 1 et 2 permet d'améliorer considérablement leur complexité et rend l'algorithme 2 utilisable pour des exemples peu profonds en nombre de boucles imbriquées.

### 2.4.6 Résultats expérimentaux

La mise au point de l'ordonnancement pour la méthodologie DTSE est encore très récente et nous n'avons, pour l'instant, effectué des essais que sur le premier algorithme proposé car il permet déjà de trouver un ordonnancement correct sur tous les exemples réalistes sur lesquels nous l'avons expérimenté. Ces applications sont représentatives de différents domaines et incluent le noyau utilisé pour la détection du mouvement (*Motion Estimation – ME*) dans la compression vidéo MPEG-4 [BCBM98], un algorithme général pour le problème des chemins algébriques (*Algebraic Path Problem – APP*) tel qu'utilisé dans de nombreux problèmes scientifiques [BQR+90] et un algorithme adaptatif de décomposition en valeur singulière (*Updating Singular Value Decomposition – USVD*) utilisé dans les applications de réseaux sans fil [MDV92]. Ces expérimentations ont été effectuées avec l'aide de Sven Verdoolaege de l'université catholique de Louvain en Belgique.

#### Noyau de détection du mouvement MPEG-4

Pour le noyau de détection du mouvement nous sommes partis d'une description exécutable pour laquelle de nombreuses transformations de boucles (et donc de nombreux ordonnancements) sont possibles initialement. Au moins 6 de ces possibilités de réorganisation sont pertinentes pour une étude approfondie comme le montrent les expérimentations effectuées dans [KCA01]. Un optimum a été identifié par des recherches manuelles intensives [BCBM98] et est confirmé par la technique d'estimation présentée dans [KCA01].

En partant de ces différents codes initiaux, l'algorithme 1 proposé ici trouve automatiquement à partir du placement le meilleur ordonnancement possible offrant un optimum global minimisant les besoins de stockage mémoire. Ce meilleur ordonnancement possible, utilisant 257 valeurs stockées en mémoire, est présenté à la figure 2.18. Cet exemple étaye de façon significative la qualité de l'approche proposée. Un autre ordonnancement possible, utilisant 1260 valeurs stockées en mémoire, est montré à la figure 2.19.

```
for(i=0; i<=31; i++)
  for(j=0; j<=31; j++)
    for(k=0; k<=15; k++)
      for(l=0; l<=15; l++)
        if(l==0 && k==0)
          s[i][j][k][l]=f1(c[k][l],p[i+k][j+l]);
        else if(l==0 && k!=0)
          s[i][j][k][l]=f2(s[i][j][k-1][15],c[k][l],p[i+k][j+l]);
        else if(l==0 && k!=0)
          s[i][j][k][l]=f3(s[i][j][k][l-1],c[k][l],p[i+k][j+l]);
for(i=0; i<=31; i++)
  for(j=0; j<=31; j++)
    r[i][j]=f4(s[i][j][15][15]);
```

FIG. 2.18 – Meilleur ordonnancement pour la détection du mouvement dans MPEG-4

```
for(k=0; k<=15; k++)
  for(l=0; l<=15; l++)
    for(i=0; i<=31; i++)
      for(j=0; j<=31; j++)
        if(l==0 && k==0)
          s[i][j][k][l]=f1(c[k][l],p[i+k][j+l]);
        else if(l==0 && k!=0)
          s[i][j][k][l]=f2(s[i][j][k-1][15],c[k][l],p[i+k][j+l]);
        else if(l==0 && k!=0)
          s[i][j][k][l]=f3(s[i][j][k][l-1],c[k][l],p[i+k][j+l]);
for(i=0; i<=31; i++)
  for(j=0; j<=31; j++)
    r[i][j]=f4(s[i][j][15][15]);
```

FIG. 2.19 – Autre ordonnancement possible pour la détection du mouvement dans MPEG-4

#### Algebraic Path Problem

La description initiale de l'APP [BQR+90] n'est pas exécutable procéduralement si on la traduit directement en langage C. Une première phase de placement doit donc la rendre exécutable [FaHSCM94]. La phase d'optimisation de la régularité, utilisant le critère de finesse du cône de dépendance, a été appliquée et a montré que la description initiale était déjà régulière [DCM00]. De même, l'étape d'optimisation de la localité par translation des polytopes ne change pas la description car ce critère avait déjà été traité par les transformations permettant de rendre la description exécutable.

L'algorithme 1 a été appliqué à ce placement final. Le résultat produit est le meilleur ordonnancement minimisant la somme des longueurs des dépendances. Il s'agit aussi du meilleur ordonnancement possible pour la localité des accès dans notre contexte.

#### Algorithme *Updating SVD*

Le placement de cet exemple n'a pas encore été déterminé complètement car le prototype de l'outil pour le placement n'est pas encore complet. Nous avons donc utilisé le placement, non optimisé, fourni par la description initiale présentée dans [MDV92]. L'algorithme 1 a été appliqué à cette description, le résultat obtenu est le même ordonnancement que celui de départ car il s'agit du seul ordonnancement valide. Ceci montre que notre approche permet de trouver un tel ordonnancement même dans les cas où l'espace de recherche est très contraint (dans ce cas un seul ordonnancement possible). Des expérimentations supplémentaires doivent être effectuées, dès que l'outil de placement sera complété, sur cet exemple car la placement initial offre de large possibilités d'améliorations.

Les exemples pratiques essayés jusqu'à maintenant montrent que l'on peut obtenir un résultat optimal pour le critère de localité avec la première méthode. Toutefois, la méthode de transformations de boucles par placement et ordonnancement ainsi que les algorithmes d'ordonnancement proposés ici doivent encore être validés sur des exemples plus difficiles et de plus grande taille. Ces exemples devront permettre de mesurer les limites d'utilisation de la première méthode d'ordonnancement et guider le développement de fonctions objectives plus complexes (et plus réalistes), comme celle permettant de mesurer la réutilisation des valeurs sur des dépendances transitives [Dan01]. Le deuxième algorithme n'a pas encore été mis en œuvre ni confronté à des exemples pratiques. Son utilisation nécessite une interaction importante —qui n'a pas encore été mise en place— avec la phase de placement afin de pouvoir récupérer les informations nécessaires à l'utilisation des dépendances extrémales. Sans ces informations, la complexité de la deuxième méthode la rend inutilisable dans des cas réalistes.

## 2.5 Conclusion

Plusieurs expérimentations manuelles ont prouvé l'excellente efficacité du flot DTSE proposé par IMEC [BCBM98, NCK+96, CWG+98]. La principale originalité de la phase de transformations de boucles est la séparation des problèmes de placement et d'ordonnancement permettant le traitement de problèmes réalistes comportant de nombreux nids de boucles et tableaux. Dans ce contexte, la contribution de cette thèse permet de trouver un ensemble de vecteurs d'ordonnancement en temps polynomial pour les cas simples et en temps exponentiel, selon la dimensions de l'espace d'itération,

pour les cas plus complexes mais peu vus en pratique. De plus, ces heuristiques peuvent être adaptées à plusieurs fonctions objectives permettant de faire des compromis entre optimisation pour la mémoire et parallélisation. Un prototype a été réalisé et utilisé sur des exemples réalistes permettant ainsi de valider l'approche utilisant systématiquement la première méthode d'ordonnancement.

La méthodologie DTSE repose en partie sur l'hypothèse qu'un accès à la mémoire reste beaucoup plus coûteux qu'une opération arithmétique. Cette hypothèse pourrait être remise en cause avec l'apparition des mémoires DRAM embarquées réduisant les coûts des transferts entre les caches et la mémoire principale et ainsi la différence de consommation entre un accès mémoire et une opération arithmétique. La méthodologie ne tient pas compte de la complexité et de la taille du code généré, les transformations et le nombre d'opérations arithmétiques qu'elles introduisent devront être redéfinis en conséquence. L'utilisation de la méthodologie ADOPT [MCJM98] permet de simplifier les expressions des calculs d'adresses et d'indexation des tableaux. Toutefois la complexité des transformations mises en œuvre (aussi bien du point de vue de la représentation des dépendances que des modifications apportées au programme) rend difficile l'interaction avec le concepteur si celui-ci n'est pas expert en compilation logicielle.

De plus, toutes les étapes de la méthodologie DTSE ne sont pas actuellement entièrement automatisées. Cependant, il n'est pas certain que l'automatisation complète des transformations de programme et d'architecture dans le cadre de la conception de systèmes soit nécessaire ou même souhaitable. En effet, le concepteur doit faire face à des contraintes spécifiques de réutilisation, de lisibilité et de maintenance pour les composants du système.

C'est pour ces raisons que nous avons également développé pendant cette thèse des méthodes de transformations de boucles permettant de modifier la description d'une application de manière interactive avec le concepteur. Ces transformations étant source à source et indépendantes de l'architecture cible, le concepteur peut garder le contrôle du code et du partitionnement. Ces méthodes de transformations de boucles sont présentées dans le chapitre suivant.

# Chapitre 3

# Transformations pour une architecture abstraite

Dans ce chapitre nous présentons les transformations de programme pour la gestion de la mémoire lorsque cette dernière n'est pas encore définie. Les paramètres de configuration tels que le nombre de niveaux disponibles dans la hiérarchie, les tailles des mémoires caches ou les tailles des blocs (lignes de cache) ne sont donc pas utilisables dans les fonctions objectives servant à guider les transformations.

Nous revenons sur les transformations de boucles décrites au chapitre 2 et nous précisons leurs possibilités et limites dans le contexte présent. Ensuite nous présentons deux transformations optimisant la consommation et développées durant cette thèse : la fusion de boucles pour la minimisation des tableaux de stockage des calculs temporaires à la section 3.3 puis l'alignement de boucles pour la minimisation des dépendances entre itérations à la section 3.4. La section 3.5 discute de la combinaison possible et future de ces deux méthodes.

## 3.1 Place mémoire d'un système et localité des calculs

Les étapes de conception liées à la mémoire n'ayant pas encore été réalisées, le nombre et la taille des niveaux de la hiérarchie mémoire (caches, mémoire interne, mémoire externe) ne sont pas encore fixés à ce niveau de conception. Nous ne pouvons donc pas connaître les types de mémoires utilisés, les tailles exactes, le nombre de ports d'entrée/sortie ...

Les fonctions objectives utilisées doivent être suffisamment générales pour ne pas influencer les décisions futures. Nous considérons toutefois que l'architecture cible contient une hiérarchie mémoire, mais qu'elle n'est pas encore connue. Nous supposons également que la méthodologie de conception intègre les transformations vues aux chapitres précédents et que les optimisations possibles seront effectuées au fur et à mesure de la précision de caractéristiques architecturales.

Le premier problème est celui de la mesure statique du temps dans un programme. Ce problème est rendu d'autant plus complexe que la cible n'est pas encore générée. Nous considérons donc dans ce chapitre le temps mesuré selon un nombre d'itérations de boucles pour les transformations locales

et une mesure au niveau des nids de boucles pour les transformations globales.

Nous avons choisi de considérer certaines hypothèses pour mesurer la place mémoire occupée par une description comportementale. Les tableaux sont pris dans leur ensemble, l'espace adressable d'un tableau ne pouvant être morcelé en mémoire ni recouvert pas d'autres variables. Ces considérations sont levées par les étapes d'optimisations suivantes dans le flot de synthèse. Nous pouvons donc réutiliser ici le coût mémoire défini à la section 2.2.1 du chapitre précédent en précisant que nous considérons la taille des tableaux selon leur nombre total d'éléments (le produit des dimensions pour les tableaux multi-dimensionnels) et le temps en grains correspondant à des nids de boucles ou à des itérations.

$$\text{Coût Mémoire} = \underset{\forall t}{max} \left\{ \sum_{v \in \text{Variables vivantes}(t)} taille(v) \right\}$$

De même que pour la place mémoire, la localité des calculs ne peut être estimée de manière précise dans ce contexte. Il est tout de même possible de procéder à des optimisations permettant de l'améliorer. Le coût *temporel* définit le temps écoulé en nombre d'itérations entre deux accès consécutifs au même emplacement d'un tableau. Si ce temps est long la hiérarchie mémoire est mal exploitée. De plus, la taille de la mémoire ne pourra être réduite à son minimum dans les prochaines étapes d'optimisation car la localité à ce niveau a une influence importante sur les optimisations de contraction de tableaux.

Nous pouvons réutiliser le critère de localité temporelle défini au chapitre précédent :

$$\sum_{(i',i) \in DoubleAccès} |i' - i|$$

où *DoubleAccès* est l'ensemble des dépendances correspondant à deux accès successifs à un même emplacement mémoire. Cependant la précision de ce critère est limitée par la granularité importante que l'on utilise pour mesurer le temps.

Les transformations proposées dans les sections 3.3 et 3.4 permettent de réduire un tableau ou une dimension de tableau à une fenêtre de valeurs de taille inférieure (pouvant être dans le meilleur cas une variable scalaire). La durée de vie des valeurs y est évaluée en utilisant une notion de temps représentée par des nids de boucles complets pour la première transformation et par des itérations de boucles pour la seconde.

## 3.2 Possibilités offertes par les transformations

Les transformations de boucles présentées à la section 2.3.2 du chapitre précédent restent présentes à ce niveau de conception. Certaines sont toutefois limitées. D'autres doivent être modifiées selon la précision des critères.

**Le pavage :** L'application des techniques de pavage est extrêmement limitée à ce niveau car on ne dispose pas d'informations suffisantes sur les particularités matérielles de l'architecture, en particulier la taille du cache et sa méthode de gestion. Toutefois, il est certain qu'un nid de boucles sera alloué à une même unité de calcul dans la suite de la conception. Les prochaines étapes de

compilation pourront, et devront donc utiliser le pavage en bénéficiant des transformations effectuées avant le partitionnement matériel/logiciel.

**La distribution :** La distribution de boucle est utilisée pour diminuer la quantité de données manipulées dans une même boucle. Comme on ne connaît pas encore la taille des mémoires, les seuls cas de distribution que l'on puisse obtenir sont ceux pour lesquels il n'y a pas de dépendance entre les données. La distribution simplifiée à ce niveau peut donc être effectuée sur une simple analyse de dépendance.

**L'échange :** L'échange est une transformation pouvant être utilisée pour diminuer la longueur des dépendances les plus externes et optimiser le critère de localité. C'est une transformation intéressante à ce niveau de conception. Nous ne l'avons pas étudiée plus en détail faute de temps mais il y a fort à parier que son potentiel est important.

**L'inversion :** De même que pour le chapitre précédent, l'inversion ne modifie pas à ce niveau les accès à la mémoire. Cependant cette transformation devra être considérée pour la mise en place de combinaisons de transformations globales de nids de boucles. En effet, l'application d'une inversion de boucle peut permettre d'effectuer une autre transformation qui n'était pas possible sans le renversement des dépendances (ce cas peut arriver par exemple lors de la fusion de deux boucles).

La fusion et le décalage d'instructions (appelé alignement dans le contexte de cette thèse) constituent les principaux travaux de ce chapitre et sont développés dans les prochaines sections. La fusion permet de minimiser la taille mémoire utilisée par les tableaux de calculs temporaires et d'augmenter la localité des calculs. L'alignement minimise le nombre de mémoires de premier plan requis pour stocker les valeurs temporaires calculées et utilisées dans une même boucle. La conclusion de ce chapitre discutera de la combinaison des méthodes applicables dans le contexte présent.

## 3.3 Fusion de boucles pour la minimisation des tableaux temporaires

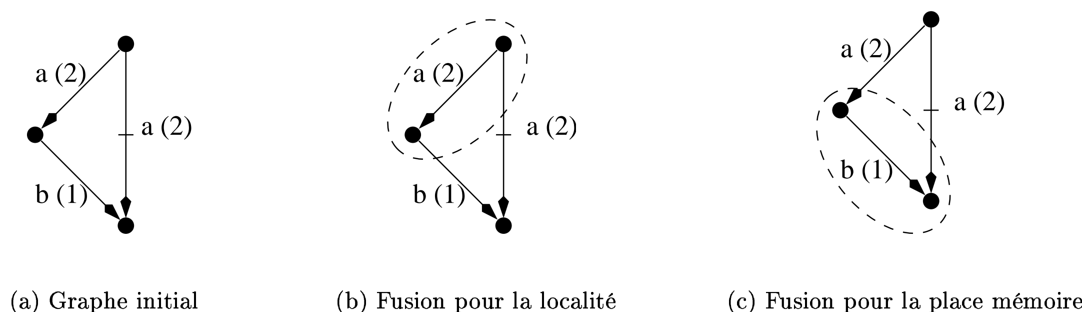
L'algorithme que nous présentons dans cette section propose une solution optimale pour la minimisation de la taille des tableaux servant de stockages intermédiaires dans les programmes composés de nids de boucles. En effet, les applications multimédia utilisent de nombreux nids de boucles pour appliquer des traitements sur les flux (application de filtres gaussiens, sous ou sur échantillonnage, calculs de maxima locaux, ...). Ces traitements sont appliqués les uns après les autres sur les données de départ dans un souci de clarté. Les descriptions comportementales utilisent des nids de boucles distincts pour chaque traitement. Des stockages temporaires sont donc nécessaires pour assurer les communications entre les traitements. L'algorithme de fusion présenté dans cette section permet de réorganiser les calculs effectués pour rapprocher la création d'un tableau de son utilisation.

### 3.3.1 Travaux existants sur la fusion

La fusion de boucles à déjà été étudiée dans de nombreux articles. Les travaux se rapprochant le plus du problème posé ici sont ceux de Manjikian et Abdelrahman [MA97], McKinley, Carr et Kennedy [MK93, CK94, MCT96] et Gao et Sarkar [SG91, GOST92]. Les autres travaux traitent de l'utilisation de la fusion dans un contexte de parallélisation, par exemple pour la minimisation des barrières de synchronisation entre boucles séquentielles et boucles parallèles ou pour la fusion maximale des nids de boucles parallèles [MK93, RK98, KM94, MS97].

Manjikian et Abdelrahman [MA97] proposent un algorithme de fusion pour la localité des calculs et le parallélisme. Leur algorithme permet d'obtenir une fusion totale des boucles en considérant des décalages d'itérations lorsque des dépendances de données interdisent la fusion directe de deux boucles. Le nid de boucles ainsi obtenu est ensuite découpé en blocs parallèles. Leur technique requiert des dépendances uniformes entre les nids de boucles et ne propose pas de compromis pour contrôler le nombre de décalages effectués sur une boucle. L'algorithme que nous proposons n'utilise pas la technique des décalages pour obtenir une fusion totale mais permet de fusionner un graphe sans la contrainte d'avoir des dépendances uniformes. La section 3.5 traite de la combinaison de la fusion et du décalage dans notre contexte.

L'algorithme de réutilisation maximale proposé par McKinley et Kennedy [MK93] permet d'effectuer la fusion d'un graphe pour maximiser la réutilisation des valeurs (représentée par des dépendances de flot ou d'entrée). La figure 3.1 montre un exemple ayant une solution différente pour la minimisation des tableaux temporaires —que nous proposons ici— et pour la maximisation de la localité. Le graphe de dépendances de départ (figure 3.1(a)) est composé de trois nœuds représentant des boucles. Les arcs du graphe sont étiquetés par les tableaux portant les dépendances et pondérés par la taille des tableaux. La dépendance portée par l'arc barré et étiqueté par $a$ ne peut être fusionnée (soit à cause de problèmes de dépendances, soit par incompatibilité des en-têtes de boucles dans le cas de l'algorithme de réutilisation maximale). La maximisation de la réutilisation produit le graphe de la figure 3.1(b) pour lequel l'arc étiqueté par $a$ est préféré en raison de son poids plus élevé. Cependant cette fusion ne permet pas de diminuer le nombre et la taille des tableaux présents en mémoire car le stockage des tableaux $a$ et $b$ reste nécessaire. La minimisation de la place mémoire représentée sur le graphe de la figure 3.1(c) est obtenue par la fusion de l'arc étiqueté par $b$, qui permet de réduire sa taille ou de le remplacer par une variable scalaire. McKinley et Kennedy démontrent que leur problème est NP-Complet.




(a) Graphe initial (b) Fusion pour la localité (c) Fusion pour la place mémoire

FIG. 3.1 – Comparaison sur un cas simple de la maximisation de la localité proposée par McKinley et Kennedy et de la minimisation des tableaux de calculs temporaires.

Gao et Sarkar [SG91, GOST92] proposent de transformer un ensemble de boucles afin de maximiser le nombre de tableaux pouvant être contractés. Les transformations utilisées sont la fusion, l'échange et l'inversion. Il s'agit donc du même problème que celui présenté dans cette section. Toutefois, leur approche est limitée sur de nombreux aspects. Le code source considéré doit utiliser une programmation en assignation unique et les indices de boucles ne doivent pas apparaître plusieurs fois dans les fonctions d'accès aux tableaux. C'est une hypothèse irréaliste dans le cas des applications multimédia embarquées. De plus, l'algorithme proposé pour la fusion des boucles simples est une heuristique. Une extension pour les nids de boucles à deux dimensions est proposée mais n'est pas extensible pour des dimensions plus élevées.

La fusion est un problème considéré par la littérature comme étant difficile à résoudre. Le lecteur pourra trouver dans [Dar00] une étude de complexité sur les différents problèmes de fusion. Nous proposons un algorithme exponentiel permettant de trouver une solution optimale au problème posé dans cette thèse. Nous considérons une classe plus large de problèmes pour lesquels les boucles sélectionnées pour la fusion n'ont pas nécessairement des en-têtes semblables. Ces boucles peuvent être fusionnées en utilisant des prédicats de contrôle dans le corps des boucles. Les techniques utilisées pour effectuer les optimisations de stockage des tableaux ou le remplacement par des variables scalaires peuvent manipuler des flots de contrôle conditionnels [De 98, CK94]. Enfin, nous montrerons l'efficacité en pratique de notre algorithme.

### 3.3.2 Modélisation du problème

Nous utilisons un graphe de flot de données $G = (V, E, A)$ afin de représenter les dépendances du problème. Les nœuds ($V$) représentent les nids de boucles du programme et les arcs ($E$) représentent les dépendances de données entre ces boucles. L'ensemble $A$ représente les tableaux créant les dépendances. Chaque arc du graphe représente une dépendance portant sur un seul tableau $a_i$ et est étiqueté par le nom de ce tableau. Il est de plus pondéré par la taille, connue à la compilation, de ce tableau $taille(a_i)$. Un arc n'est étiqueté que par un unique tableau. S'il existe de multiples dépendances entre deux nœuds alors le problème est représenté par un multi-graphe.

Le graphe de dépendance est généré à partir d'un programme exécutable. Il est orienté et acyclique, par définition.

Une dépendance de donnée entre deux références à un même tableau est représentée par des vecteurs mixtes de distances et directions $\overrightarrow{\delta} = \{\delta_1, \ldots, \delta_n\}$ décrits avec l'information la plus précise pour chaque composante (on choisira une représentation par distance plutôt que par direction si cette première est disponible).

La figure 3.2 représente le graphe de dépendance calculé à partir du code source donné en exemple. Les boucles `L2` et `L3` ne peuvent être fusionnées à cause de la dépendance portée par le tableau `a2`. En effet, si les boucles étaient fusionnées, le programme utiliserait la valeur `a2[i+1]` (instruction provenant de `L3`) avant son calcul `a2[i]` (instruction provenant de `L2`) dans la même itération.

Deux nœuds peuvent être fusionnés si et seulement si aucune des dépendances n'est renversée dans la boucle fusionnée par rapport au programme de départ. Un arc portant une dépendance interdisant la fusion de sa source et de sa destination est appelé un *arc empêchant la fusion – AEF*. Il est barré sur la représentation.

Les instructions isolées (hors des boucles) sont considérées comme des boucles ayant un domaine

```
L1: for(i=1; i<n; i++)
      a1[i] = ... ;
L2: for(i=1; i<n; i++)
      a2[i] = f2(a1[i]);
L3: for(i=1; i<n; i++)
      a3[i] = f3(a2[i+1]);
L4: for(i=1; i<n; i++)
      a4[i] = f4(a1[i-1],a3[i]);
L5: for(i=1; i<n; i++)
      ... = f5(a2[i],a3[i],a4[i]);
```

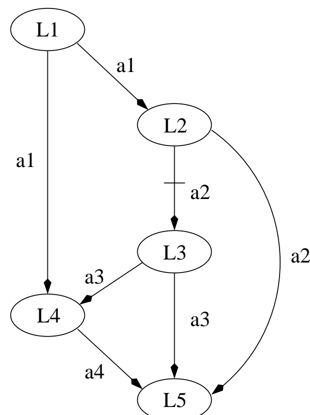



FIG. 3.2 – Exemple de programme et son graphe de dépendances associé

d'itération réduit à un point. Les dépendances entre les instructions et les boucles sont préservées lors de la transformation de programme. Une instruction isolée sera représentée comme un nœud du graphe ; tous ses arcs entrants et sortants seront marqués comme empêchant la fusion.

Un tableau peut être « supprimé » de la mémoire, ou au moins réduit en taille, si l'on peut fusionner entre elles toutes les boucles procédant à des écritures et des lectures de ce tableau. Un tableau de ce type est marqué par une étoile dans la représentation (voir figure 3.3). On appelle par extension « arc étoilé » un arc étiqueté par un tableau pouvant être enlevé. La prochaine section décrit l'algorithme qui permet d'identifier ce type de tableaux.

### 3.3.3 Détection des tableaux supprimables

Afin de pouvoir enlever par fusion un tableau $a$ du programme, nous devons pouvoir fusionner tous les nœuds connectés par des arcs étiquetés par ce tableau dans le graphe de dépendances. Par conséquent, un tableau $a$ porté par un arc $e = (u, v)$ ne peut pas être enlevé si $e$ est un AEF ou s'il existe dans le graphe un chemin de $u$ à $v$ contenant un AEF.

Afin de détecter et de représenter les chemins qui empêchent la fusion nous effectuons une clôture transitive sur le graphe. La clôture transitive effectuée sur le graphe d'exemple est montrée sur la figure 3.3.

L'étiquette `a2` ne peut être enlevée car il existe un AEF entre les nœuds `L2` et `L3`. De même pour l'étiquette `a1` pour laquelle il existe un chemin entre `L1` et `L4` passant par un AEF (`L2`,`L3`). En effet, enlever l'étiquette `a1` requiert la fusion des nœuds `L1`, `L2` et `L4` créant ainsi un cycle entre les nœuds `L124` et `L3` dans le graphe de dépendance. Un tel cycle n'est pas légal car il ne préserve pas les contraintes de précédences imposées par les dépendances de données (la modélisation des dépendances au niveau des boucles ne crée pas de dépendance d'un nœud vers lui-même).

Les tableaux `a3` et `a4` sont marqués d'une étoile car ils peuvent être enlevés par fusion des nœuds `L3`, `L4` et `L5`. La détection de tels tableaux se fait par parcours du graphe fermé : il est inutile de fusionner les arcs $u \xrightarrow{a} v$ s'il existe un chemin entre $u$ et $v$ contenant un AEF $u' \nrightarrow v'$ dans le graphe fermé. On peut ainsi construire une liste des tableaux pouvant être supprimés en mémoire. Nous appelons, par extension, un *arc étoilé* un arc étiqueté par un label étoilé.

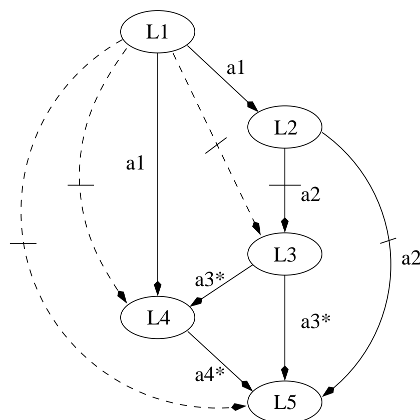



FIG. 3.3 – Clôture transitive du graphe de la figure 3.2 pour la détection des tableaux supprimables

### 3.3.4 Conflits entre tableaux

L'étape précédente permet de détecter l'ensemble des étiquettes (tableaux) pouvant être enlevées du graphe par fusion. Cependant, cette détection est locale. Elle permet d'identifier une fusion seule. Or toutes les fusions ne sont pas indépendantes comme nous allons le voir avec les deux exemples suivants :

**Conflits directs** La figure 3.4 présente un exemple de conflit entre deux tableaux. Les arcs étiquetés par les tableaux `a` et `b` peuvent être fusionnés si on les considère de manière indépendante mais ils ne peuvent pas être fusionnés en même temps.

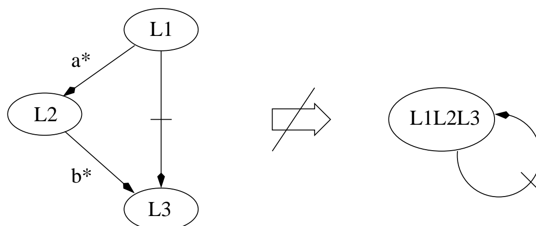



FIG. 3.4 – Conflit direct de fusion

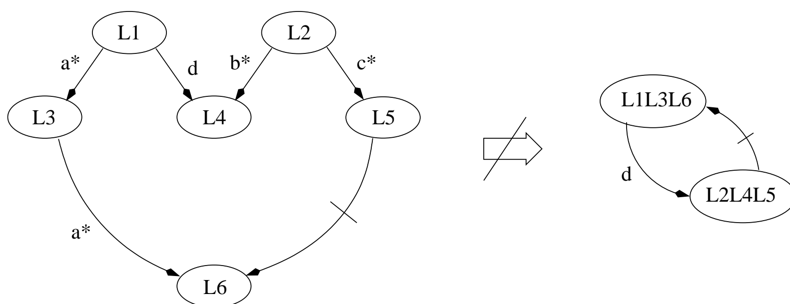



FIG. 3.5 – Conflit sur un cycle de fusion

**Conflits sur un cycle non orienté** La situation peut être plus compliquée dans le cas de conflits présents sur un cycle non orienté. La figure 3.5 présente un exemple de conflit sur un cycle. La détection des tableaux supprimables permet de déterminer que les tableaux `a`, `b` et `c` sont supprimables. Cependant la fusion des nœuds du graphe en deux groupes (`L1`,`L3`,`L6`) et (`L2`,`L4`,`L5`) est illégale car le graphe devient alors cyclique.

Afin de compléter le processus de fusion, nous devons donc résoudre les conflits possibles. Cette résolution est faite en deux étapes : la première identifie l'ensemble des conflits et la seconde résout de manière optimale ces conflits en utilisant un algorithme de programmation linéaire en nombres entiers.

### Détection des conflits

Les conflits sont générés par la présence d'un AEF sur un cycle composé en partie d'arcs étoilés. Soit $\mu$ un cycle élémentaire du graphe. On définit arbitrairement un sens de parcours donné pour ce cycle. On note par $\mu^+$ l'ensemble des arcs du cycle orientés dans le sens de parcours et par $\mu^-$ les arcs orientés dans le sens inverse [GM85]. On peut associer à $\mu$ un vecteur $\vec{\mu} = (\mu_1, \mu_2, \ldots, \mu_{|E|})$ tel que :

$$\mu_u = \begin{cases} +1 & \text{si } u \in \mu^+ \\ -1 & \text{si } u \in \mu^- \\ 0 & \text{si } u \notin \mu^+ \cup \mu^- \end{cases}$$

Le vecteur $-\vec{\mu}$ est également un vecteur associé au cycle $\mu$ mais avec le sens de parcours opposé.

**Proposition 3** *Afin de résoudre tous les conflits pouvant créer un cycle illégal après la fusion nous devons détecter et couper tous les cycles non orientés $\mu$ du graphe tels que $\vec{\mu}$ soit composé de la manière suivante :*

- *tous les $+1$ (resp. $-1$) correspondent à des arcs étoilés ;*
- *au moins un des $-1$ (resp. $+1$) est un AEF.*

**Preuve** Supposons que le graphe $G$, une fois fusionné, soit illégal. Nous voulons montrer que le graphe $G$ avant la fusion est composé d'au moins un cycle tel que défini à la proposition 3.

Un graphe fusionné est illégal s'il contient un cycle de dépendances orienté $\mu_d$. De plus, ce cycle est composé d'AEF ou d'arcs non étoilés car tous les arcs étoilés ont été fusionnés.

Comme il n'y avait pas de cycle de dépendance orienté avant la fusion (le graphe de dépendance d'un programme est par définition un graphe orienté acyclique), le dernier arc $e_s$ fusionné est étoilé. De plus, il est orienté dans une direction empêchant le graphe de départ d'être cyclique. Donc, le vecteur $\vec{\mu}'$ associé au cycle non orienté composé de $\mu_d \cup e_s$ correspond à la définition donnée à la proposition 3.

Nous voulons maintenant montrer que, s'il existe un cycle non orienté tel que défini à la proposition 3 dans le graphe de départ, alors le graphe résultant de la fusion de tous les arcs étoilés est illégal. Supposons que nous ayons fusionné tous les arcs étoilés d'un cycle non orienté $\mu$ produisant un graphe $G'$. Le vecteur $\vec{\mu}'$ de $G'$ contient un seul élément $+1$ (respectivement $-1$) correspondant à l'arc étoilé restant à fusionner et au moins un $-1$ (respectivement $+1$) correspondant à un AEF.

Si nous fusionnons le dernier arc étoilé alors le graphe résultant de la fusion contient un cycle orienté contenant au moins un AEF. □

### Algorithme de détection des cycles

La détection des conflits est effectuée par une exploration du graphe $G$ de départ. Une liste $C$ contenant tous les cycles de conflits est construite en cherchant pour chaque AEF $(u, v)$ les chemins entre $u$ et $v$ correspondants à la proposition 3.

La fonction **chemins**($G = (V, E)$,k,u,v) effectue une recherche exhaustive par progression dans le graphe en marquant tous les sommets $k$ qu'elle rencontre lors de la recherche d'un chemin entre $u$ et $v$ (différent de l'AEF $(u, v)$). Nous cherchons donc tous les chemins $\mu$ de $u$ à $v$ pour lesquels $\overrightarrow{\mu}$ contient seulement des arcs étoilés dans $\mu^+$. La progression dans le graphe à partir d'un sommet $k$ déjà atteint peut donc être limitée aux arcs étoilés dirigés dans le sens du parcours et aux arcs quelconques dirigés en sens inverse. Le parcours sur un chemin est stoppé dans les autres cas (arc non étoilé ou AEF partant de $k$ et orienté dans le sens du parcours).

**Algorithme 3** (*Exploration du graphe*)

> **chemins**($G = (V, E), k, u, v$)
> début
>   si $k = v$
>     $C \leftarrow C \cup$ le chemin composé des sommets $x$ tels que marque[$x$]=vrai
>   sinon
>     marque[$k$] $\leftarrow$ vrai
>     pour tout $t \in V$ faire
>         si $e = (k, t) \in E$ est étoilé et marque[$t$] $=$ faux
>           chemins(G,t,u,v)
>         sinon si $e = (t, k) \in E$ et marque[$t$] $=$ faux
>           chemins($G, t, u, v$)
>     marque[$k$] $\leftarrow$ faux
> fin
>
> **exploration** ($G = (V, E)$)
> début
>   $C \leftarrow \emptyset$
>   marque[k] $\leftarrow$ faux, $\forall k \in V$
>   pour tout AEF $e = (u, v) \in V$
>     **chemins**(G,u,u,v)
> fin

Le nombre de cycles d'un graphe est exponentiel par rapport au nombre de nœuds. Ce problème peut rendre la résolution complexe si l'on ne prend pas garde à utiliser une représentation compacte pour les cycles. Cependant, nous verrons à la section 3.3.6 que les temps de calculs pour des graphes de tailles représentatives du type d'application traité sont tout à fait acceptables.

**Résolution des conflits par programmation linéaire en nombres entiers**

Une fois l'ensemble des conflits détecté, nous devons résoudre le problème de façon globale afin de déterminer l'ensemble des tableaux fusionnables qui ne seront pas considérés par la fusion.

On définit une variable binaire $x_{a_i}$ pour chaque tableau étoilé $a_i$ pouvant être enlevé mais étant inclus dans un des conflits détectés à l'étape précédente. Si $x_{a_i} = 0$ alors le tableau $a_i$ reste considéré pour la fusion, dans le cas contraire ($x_{a_i} = 1$) le tableau ne sera plus étoilé et ne sera pas concerné par la fusion.

Si le graphe est un multi-graphe, ayant par exemple $k$ arcs étiquetés $a_1, a_2, \ldots, a_k$ entre deux nœuds $u$ et $v$, on définit une variable $x_{uv}$ inférieure ou égale à chacune des variables $x_{a_1} \ldots x_{a_k}$ associées aux tableaux portés par les arcs entre $u$ et $v$. Si $x_{uv}$ est égal à 1 (tous les arcs sont enlevés d'un chemin) alors toutes les variables associées seront égales à 1 et les tableaux étiquetant ces arcs ne seront plus étoilés. Une variable $x_{a_i}$ peut être égale à 1 sans interférer avec les autres tableaux portés par les arcs entre $u$ et $v$ (équation 3.3).

Pour chaque cycle $\mu$ de $C$ nous devons décider quels arcs de $\mu^+$ ne seront par fusionnés et donc quels tableaux ne pourront être enlevés de la mémoire. Ceci est assuré par l'équation 3.2 indiquant que la somme des variables $x_{uv}$ de l'ensemble $\mu^+$ d'un cycle doit être supérieur ou égal à 1 (au moins un arc doit être supprimé).

La fonction objective (équation 3.1) du programme est de minimiser la somme globale des tailles des tableaux enlevés de l'ensemble des tableaux fusionnables détectés en conflit lors de l'étape décrite dans la section 3.3.3.

$$\min \left( \sum_{a_i \in A} taille_{a_i} * x_{a_i} \right) \tag{3.1}$$

$$\sum_{(u,v) \in \mu^+(c)} x_{uv} \geq 1, \qquad \forall \mu \in C \tag{3.2}$$

$$x_{uv} \leq x_{a_i}, \quad \forall a_i \in (u,v), \forall (u,v) \in E \tag{3.3}$$

$$x_{a_i} \in \{0,1\}, \qquad \forall a \in A \tag{3.4}$$

$$x_{uv} \in \{0,1\}, \qquad \forall (u,v) \in E \tag{3.5}$$

Les valeurs $x_{a_i} = 1, \forall i$ sont toujours une solution possible pour le problème. De plus, toutes les solutions possibles ont un coût

$$\sum_{a_i \in A} taille_{a_i} * x_{a_i} \geq 0.$$

Ce qui assure qu'une solution optimale existe toujours.

### 3.3.5 Fusion et réécriture du code

Les arcs restant étoilés après la résolution des conflits peuvent maintenant être tous fusionnés. Pour les groupes de nœuds qui vont constituer les nœuds du nouveau graphe nous devons assurer que si deux nœuds $u$ et $v$ appartiennent au même groupe alors tous les nœuds situés sur des chemins dirigés de $u$ vers $v$ seront dans le même groupe. Cette étape est effectuée par une clôture transitive

modifiée pour laquelle, s'il existe un chemin $u \xrightarrow{*} v$, un chemin $u \to w$ et un $w \to v$ ; alors $u$, $v$ et $w$ seront fusionnés dans le même groupe.

Cette fusion doit cependant être faite dans un ordre permettant de respecter la sémantique du programme. On doit donc trouver un ordre sur les sommets permettant d'effectuer les fusions comme on peut le voir à la figure 3.6. Sur cet exemple, la fusion des nœuds `L1` et `L3` doit être précédée de la fusion de `L1` et de `L2` sous peine d'introduire un cycle de dépendances entre les boucles `(L1,L3)` et `L2`.

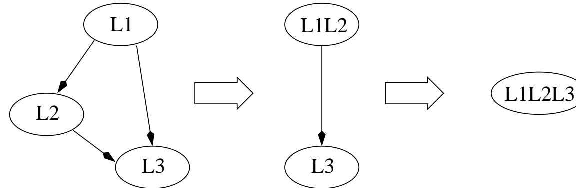
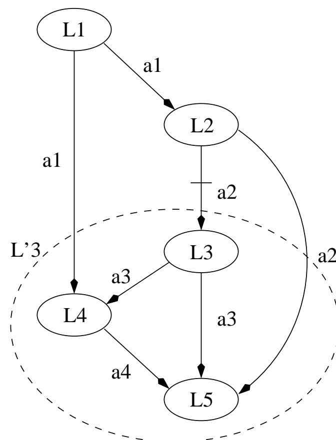



FIG. 3.6 – Importance de l'ordre des fusions

La génération du code est effectuée en récrivant le corps des boucles selon un ordre topologique, donné par exemple par la hauteur des nœuds dans le graphe :

$$h(x) = \begin{cases} 0 \text{ si } d^{-}(x) = 0 \\ \max \{h(y), y \to x \in E\} + 1 \text{ sinon} \end{cases}$$




```
L1: for(i=1; i<n; i++)
      a1[i] = ... ;
L2: for(i=1; i<n; i++)
      a2[i] = f2(a1[i]);
L3L4L5:
    for(i=1; i<n; i++) {
      a3[i] = f3(a2[i+1]);
      a4[i] = f4(a1[i-1],a3[i]);
      ... = f5(a2[i],a3[i],a4[i]);
    }
```

FIG. 3.7 – Graphe de dépendances groupé et code modifié

La figure 3.7 représente le graphe de dépendances modifié et le programme résultant de la fusion de boucles. Les tableaux `a3` et `a4` peuvent être remplacés par des variables scalaires dans la nouvelle boucle `L3L4L5`.

### 3.3.6 Expérimentations sur des graphes aléatoires

Bien qu'exponentiel en nombre de cycles, cet algorithme de fusion est très efficace en pratique. Nous avons effectué des tests de résolution sur des graphes générés aléatoirement. Les graphes considérés sont d'une complexité équivalente, sinon supérieure, aux graphes de flots de données que

l'on peut trouver dans une application multimédia de grande taille. Les caractéristiques des graphes utilisés sont résumées dans le tableau 3.1.

| Caractéristiques des graphes | |
|---|---|
| nombre de nœuds $\|V\|$ | 10–30 |
| nombre d'arcs $\|E\|$ | $3 * \|V\|$ |
| arcs entrant sur un nœud | 1–10 |
| arcs sortant d'un nœud | 1–10 |
| probabilité pour un arc d'empêcher une fusion | 1/3 |
| taille des tableaux | 1–100 |

TAB. 3.1 – Caractéristiques des graphes générés aléatoirement pour la fusion.

Les tests ont été réalisés sur 21 000 graphes différents (une série de 1000 graphes a été générée pour chaque nombre de nœuds entre 10 et 30) et nous avons utilisé le solveur `LP_SOLVE` disponible gratuitement depuis son site `ftp` [Ber]. Bien que l'étape d'énumération des cycles ainsi que la résolution du PLNE soient exponentielles, les temps de calculs moyens sont de 0,05s par graphe (ce temps comprend la construction des cycles et la résolution) sur une machine Intel Pentium II à 450MHz.

Ces bons résultats proviennent en grande partie de la redondance entre les cycles de conflits construits à partir d'un graphe. Certains exemples aléatoires peuvent générer plus de 100 000 cycles mais la résolution de ceux-ci par le programme linéaire est efficace car tous les cycles sont en partie inclus les uns dans les autres. La programmation de l'algorithme de fusion pose donc le problème de la place mémoire nécessaire au calcul et au stockage de l'ensemble des cycles du graphe. Toutefois, il est peu probable de rencontrer des graphes générant un aussi grand nombre de cycles dans un contexte réel. En effet, le nombre d'arcs entrants ou sortants d'un nœud est en pratique très largement inférieure à 10, valeur prise délibérément élevée pour les présentes expérimentations.

### 3.3.7 Conclusion sur la fusion

Nous avons défini une méthode de transformation permettant une diminution efficace de la taille mémoire en un temps raisonnable en pratique.

La transformation proposée améliore également la localité des calculs, sans toutefois l'optimiser. Il est donc important de combiner l'algorithme présenté ici avec celui pour la maximisation de la localité proposé par McKinley et Kennedy [MK93]. Les deux transformations doivent être effectuées dans un ordre précis comme le montre l'exemple de la figure 3.1 : d'abord la minimisation des tableaux temporaires puis la maximisation de la localité. En effet, l'application de la maximisation de la localité construit une fusion maximale du graphe, bloquant ainsi toute possibilité de transformation pour notre algorithme. Comme la maximisation de la localité ne sélectionne pas les fusions

en fonction du critère de taille mémoire, nous devons utiliser notre algorithme en premier. De plus, notre algorithme laisse des possibilités de fusion comme le montre l'exemple de la figure 3.7 pour laquelle les nœuds `L1` et `L2` peuvent être fusionnés afin d'augmenter la localité des calculs portant sur le tableau `a1`. La section 3.5 discute plus largement des combinaisons de transformations de boucles.

La fusion est une transformation globale opérant sur un ensemble de nids de boucles sans considérer les transactions générées par l'exécution à l'intérieur des boucles. La prochaine section présente une transformation locale permettant d'optimiser la gestion de la mémoire dans un nid de boucles de manière indépendante de la structure de la hiérarchie mémoire.

## 3.4 Alignement de boucles pour la minimisation des dépendances entre itérations

L'algorithme que nous présentons ici minimise le nombre de mémoires de premier plan requis pour stocker les valeurs temporaires calculées et utilisées dans une même boucle comme illustré à la section 2.3.2 (page 49). Cette minimisation correspond à l'optimisation de la distance moyenne entre un accès en lecture et un accès en écriture dans la même boucle. Cette technique est non seulement utile pour garder les valeurs servant au calcul dans le haut de le hiérarchie mémoire (dans des registres ou une mémoire cache), elle permet également de réduire la quantité de mémoire nécessaire lorsqu'on la combine avec les méthodes de remplacement de tableaux par des scalaires ou de réduction de taille des tableaux.

Nous présentons tout d'abord le problème dans le cas des boucles simples, pour lequel un algorithme polynomial a pu être développé. L'optimisation des nids de boucles par alignement est étudié ensuite à la section 3.4.4 et une heuristique est proposée dans ce cas.

### 3.4.1 Modélisation du problème pour les boucles simples

Nous utilisons un graphe de dépendance réduit $G = (V, E, w)$. Les nœuds $V$ du graphe représentent les instructions du corps de la boucle et les arcs $E$ représentent les dépendances de données. Chaque dépendance est pondérée par une distance $w$ qui correspond au nombre d'itérations séparant la production d'une cellule mémoire de sa consommation. Ces distances correspondent donc aux dépendances de flot et sont nécessairement positives car un programme ne peut pas utiliser une valeur avant de l'avoir produite.

Nous restreignons volontairement le problème au cas des distances de dépendances uniformes (constantes) [BGS94] dans la boucle afin de pouvoir utiliser les techniques de *retiming* [LS91, DH98]. Cette restriction limite donc le nombre de boucles que peut traiter notre algorithme dans un contexte réel. Le domaine d'application considéré ici utilise toutefois dans un grand nombre de cas des dépendances uniformes lors des traitements appliqués sur les signaux.

La figure 3.8 représente le code d'une boucle et son graphe de dépendance associé. Nous pouvons voir qu'il comporte deux dépendances de distance 2 : celle entre les nœuds `I1` et `I3` et entre les nœuds `I1` et `I2`. Il y a aussi une dépendance de distance 1 entre le nœud `I4` et les nœuds `I1` et `I2`. La valeur produite par l'instruction `I3` (`c[i]`) est utilisée dans la même itération par l'instruction `I4` et la distance de la dépendance entre ces 2 instructions est donc de 0.

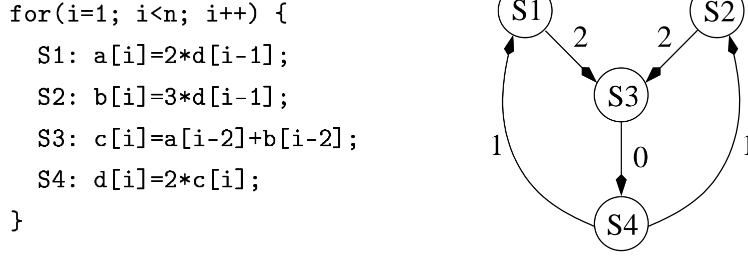



FIG. 3.8 – Modélisation des dépendances dans une boucle

On note $\Gamma^+(x)$ l'ensemble des arcs sortants et $\Gamma^-(x)$ l'ensemble des arcs entrants d'un nœud $x$. Le nombre de variables temporaires utilisées par une instruction, représentée par un nœud $u$, dépend de la distance $w(e)$ portée par les arcs $e$ sortants. Il est défini par

$$C(u) = \max_{e \in \Gamma^+(u)} w(e) \tag{3.6}$$

Le nombre total de variables temporaires utilisées dans le graphe $Coût(G)$ est donc

$$Coût(G) = \sum_{u \in V} C(u).$$

La minimisation de $Coût(G)$ peut être effectuée en temps polynomial, comme nous le verrons à la section 3.4.3, en utilisant un retiming du graphe.

Une valeur entière $r(u)$ de retiming est associée à chaque nœud $u$. Cette valeur représente le décalage, en nombre d'itérations, qui sera effectué sur l'instruction représentée par $u$. Appliquer un retiming sur un graphe modifie donc les distances de dépendance. Le graphe, après retiming, peut être récrit sous forme de programme fonctionnellement équivalent au programme source mais avec des distances de dépendance données par la relation suivante :

$$w_r(e) = w(e) + r(v) - r(u), \qquad (u \xrightarrow{e} v)$$

Nous devons également définir des contraintes afin d'assurer que le retiming est légal. En effet, on doit assurer que les distances de dépendances après retiming restes positives.

$$w_r(e) \geq 0, \ \forall e \in E$$

### 3.4.2 Formulation du programme linéaire en nombres entiers (PLNE)

Cette partie présente la formulation du PLNE permettant de minimiser la fonction $Coût(G)$.

$$\min \sum_{u \in V} C(u) \tag{3.7}$$

$$w(e) + r(v) - r(u) \geq 0, \quad \forall e = (u, v) \in E \tag{3.8}$$

$$C(u) \geq w(e) + r(v) - r(u), \quad \forall e = (u, v) \in E \tag{3.9}$$

La fonction objective de notre PLNE est donnée par la relation 3.7. Les contraintes 3.8 assurent que nous avons un retiming légal. Le coût d'un nœud, après retiming, est défini par l'équation 3.9. Comme nous ne pouvons pas utiliser une fonction max, le programme ne serait pas linéaire standard, nous devons définir le coût d'un nœud $u$ comme étant supérieur ou égal au coût de chacun de ses arcs sortants. La minimisation de 3.7 assure que la valeur maximale est atteinte par $C(u)$, donnant ainsi le coût attendu pour chaque nœud.

La formulation en PLNE donnée par (3.7), (3.8) et (3.9) minimise le nombre de variables temporaires utilisées entre les itérations de la boucle. Ce problème est très proche du problème de maximisation du partage de registres résolu par Leiserson et Saxe dans [LS91], cependant ils le résolvent par un algorithme de flot en nombres rationnels et leur solution permet difficilement de modifier la quantité à minimiser ou les contraintes de légalité. Nous allons voir que la formulation que nous proposons peut être résolue par un algorithme de flot en nombres entiers. Les algorithmes de flot ont une meilleure complexité en nombres entiers qu'en rationnels. Ceci est dû à la diminution du nombre d'étapes nécessaires pour obtenir la convergence. De plus, notre formulation permet d'imposer des contraintes de légalité et un coût dont la formulation est plus générale que celle proposée dans [LS91]. L'algorithme présenté dans cette section a été développé en collaboration avec Guillaume Huard et le lecteur intéressé trouvera dans [Hua01] une étude sur les techniques de décalage d'instructions.

Les valeurs $r(u) = 0$ et $C(u) = \max_{e \in \Gamma^+(u)} w(e)$ sont toujours des solutions possibles pour le problème. De plus, chaque solution a un coût

$$\sum_{u \in V} C(u) \geq 0$$

assurant l'existence d'une solution optimale grâce à cette borne inférieure.

### 3.4.3 Polynomialité du problème et algorithme

La représentation matricielle de la formulation linéaire précédente est donnée par la relation 3.10.

$$\min \left\{ \begin{pmatrix} r & C \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} \middle| \begin{pmatrix} r & C \end{pmatrix} \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix} \geq \begin{pmatrix} -w & w \end{pmatrix} \right\}, \tag{3.10}$$

où la matrice $A$ est une matrice d'incidence nœuds-arcs de taille $|V|*|E|$ (chaque colonne possède un et un seul $+1$ et $-1$, voir [GM85]) du graphe de dépendances réduit $G = (V, E, w)$ et la matrice $A^+$ est définie comme suit :

$$\begin{cases} a^+_{i,j} = 1 \text{ si } a_{i,j} = 1 \text{ (où } a_{i,j} \text{ est un élément de } A) \\ a^+_{i,j} = 0 \text{ sinon} \end{cases} \tag{3.11}$$

La matrice $A^+$ est de même dimension que $A$ mais n'en conserve que les valeurs positives. Cela correspond au fait que nous définissons le coût $C(u)$ seulement pour les arcs sortants d'un nœud et non pour les arcs entrants.

**Proposition 4** *Le problème défini par la relation 3.10 a une solution optimale entière qui peut être trouvée en temps polynomial.*

**Preuve** Le problème admet une solution entière optimale dans les entiers si la matrice

$$M = \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix}$$

est totalement unimodulaire [Sch86]. Une matrice est totalement unimodulaire si et seulement si toutes ses sous-matrices carrées sont unimodulaires (de déterminant 1, $-1$ ou 0). Considérons une sous-matrice carrée $M'$ de $M$, si elle contient des lignes de la sous-matrice $\begin{pmatrix} O & A^+ \end{pmatrix}$, alors pour chacune de ses lignes $l$ :

- soit la ligne correspondante $l'$ de $\begin{pmatrix} -A & A \end{pmatrix}$ est présente dans $M'$ (chacune des lignes de $A^+$ correspond à une ligne de $A$ comportant les mêmes entrées positives), dans ce cas nous pouvons soustraire $l$ de $l'$, ce qui est une transformation unimodulaire de $M'$ et ne change pas son déterminant. Comme $A$ contient au plus un seul 1 par colonne, après la soustraction toutes les entrées positives de $l$ sont les seules dans leur colonne.
- soit la ligne correspondante de $\begin{pmatrix} -A & A \end{pmatrix}$ n'est pas présente dans $M'$ et dans ce cas toutes les entrées positives de $l$ sont les seules dans leur colonne (car ce sont aussi les entrées positives d'une ligne de $A$ qui n'est pas dans $M'$ et par définition d'une matrice de connexion il n'y en a pas d'autre).

Dans tous les cas nous aboutissons finalement à une matrice $M'$ et une éventuelle transformation unimodulaire telle que chaque colonne contient au plus un seul 1 et un seul $-1$ (et toutes les autres entrées à 0). Cette dernière matrice est unimodulaire car c'est une sous matrice d'une matrice de connexion (les matrices de connexion sont totalement unimodulaires, voir [Sch86]). □

Bien que ce problème puisse être résolu de manière très efficace par un solveur de PLNE, nous présentons ici la forme duale (3.10) du problème permettant de se ramener à un algorithme de flot à coût minimal. La complexité des algorithmes de flot est généralement moins élevée que celle des algorithmes de résolution de programmes linéaires.

### Interprétation de la forme duale du problème

La forme duale du problème 3.10 est donnée par le problème 3.12 [dW90].

$$\max\left\{ \begin{pmatrix} -w & w \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} \middle| \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}, \begin{pmatrix} x & y \end{pmatrix} \geq 0 \right\} \tag{3.12}$$

Nous transformons tout d'abord le problème pour avoir un problème de minimisation plutôt que de maximisation. Le problème transformé est donné par l'équation 3.13.

$$\min\left\{ \begin{pmatrix} w & -w \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} \middle| \begin{pmatrix} -A & A \\ 0 & A^+ \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}, \begin{pmatrix} x & y \end{pmatrix} \geq 0 \right\} \tag{3.13}$$

La fonction de coût des variables $x$ et $y$ à minimiser est $w(x-y)$. Cette minimisation est contrôlée par deux ensembles de contraintes donnés par les équations 3.14 et 3.15.

$$A.(x-y) = 0,\ \begin{pmatrix} x & y \end{pmatrix} \geq 0 \tag{3.14}$$

$$A^+.y = 1,\ y \geq 0 \tag{3.15}$$

Le premier ensemble 3.14 de contraintes impose à la valeur $(x-y)$ de représenter un flot sur le graphe [GM85]. Les contraintes représentées par l'équation 3.15 indiquent que la partie $y$ du flot

sur un nœud doit être dirigée vers un unique arc sortant de ce nœud. Ces contraintes peuvent être prises en compte en construisant un nouveau graphe $G'(V', E', w')$ à partir du graphe $G(V, E, w)$ de la manière suivante :

- Les modifications pour un nœud n'ayant qu'un seul arc de sortie sont montrées sur la figure 3.9. L'arc $e = (u, v)$ est conservé avec son poids $w(e)$ et nous ajoutons un arc $e' = (v, u)$ de $v$ à $u$ de poids $-w(e)$.

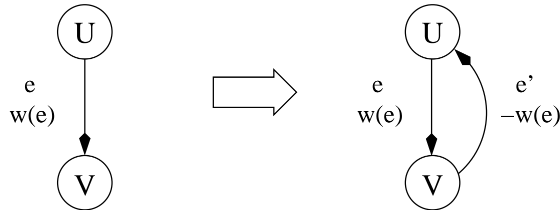



FIG. 3.9 – Transformation du graphe pour la résolution à l'aide d'un algorithme de flot

- Les transformations pour un nœud avec plusieurs arcs sortants sont plus complexes. Un exemple de transformation est montré sur la figure 3.10. Les arcs sortants $\{e_i\}$ du graphe initial sont conservés avec leur poids respectif dans le graphe transformé. Nous introduisons un *nœud virtuel* $v_u$. Pour chaque arc $e_i = (u, v)$, nous construisons un arc $e'_i = (v, v_u)$. L'arc $e'_i$ est pondéré par $-w(e_i)$ et a une capacité maximale de flot $c$ fixée à 1. Un autre arc $e_u$ est ajouté du nœud virtuel $v_u$ au nœud $u$. Cet arc, de poids nul, a une capacité minimal $l$ et une capacité maximale $c$ fixées à 1.

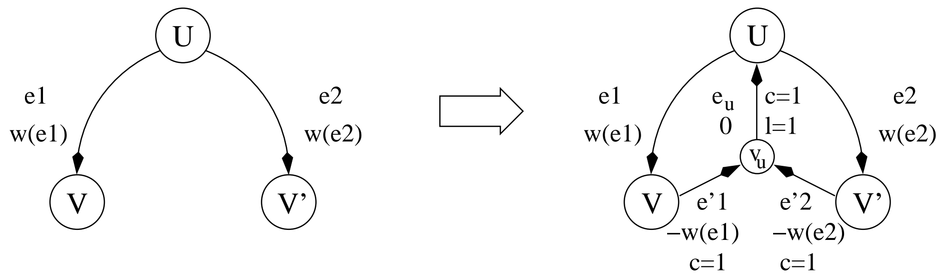



FIG. 3.10 – Transformation du graphe pour la résolution à l'aide d'un algorithme de flot

Soit $f$ un flot de $G'$, nous définissons pour chaque arc $e$ un couple $(x(e), y(e))$ de la façon suivante :

$$x(e) = f(e) \tag{3.16}$$

$$y(e) = f(e') \tag{3.17}$$

**Proposition 5** *Il y a une bijection entre les flots de $G'$ et l'ensemble des solutions de la forme duale du problème. De plus, les flots à coût minimal de $G'$ correspondent à des solutions optimales pour le problème dual.*

**Preuve** Pour chaque nœud $u$ de $G$ nous avons l'équation suivante sur $f$ :

$$\sum_{e \in \Gamma^+_E(u)} f(e) + \sum_{e' \in \Gamma^+_{E' \setminus E}(u)} f(e') = \sum_{e \in \Gamma^-_E(u)} f(e) + \sum_{e' \in \Gamma^-_{E' \setminus E}(u)} f(e') \tag{3.18}$$

À l'aide de 3.16 nous pouvons récrire cette équation sous la forme suivante :

$$\sum_{e \in \Gamma^+_E(u)} x(e) - \sum_{e' \in \Gamma^-_{E' \setminus E}(u)} f(e') = \sum_{e \in \Gamma^-_E(u)} x(e) - \sum_{e' \in \Gamma^+_{E' \setminus E}(u)} f(e'). \tag{3.19}$$

Par construction nous avons :

- $\forall e \in \Gamma^-_E(u),\ \exists!\ e' \in \Gamma^+_{E' \setminus E}(u)$ et $f(e') = y(e)$ donc

$$\sum_{e' \in \Gamma^+_{E' \setminus E}(u)} f(e') = \sum_{e \in \Gamma^-_E(u)} y(e)$$

- $\forall e \in \Gamma^-_{E' \setminus E}(u),\ e = (v_u, u)$ et

$$\sum_{e' \in \Gamma^-_{E' \setminus E}(v_u)} f(e') = f(e_u) = 1,$$

  ce qui est équivalent à la conservation du flot sur les nœuds virtuels.
  De plus, nous avons

$$\forall e' \in \Gamma^-_{E'}(v_u),\ \exists!\ e \in \Gamma^+_E(u) \text{ avec } f(e') = y(e) \text{ et } \sum_{e \in \Gamma^+_E(u)} y(e) = 1$$

  qui est la contrainte que doit satisfaire toute solution du problème dual. Donc

$$\sum_{e' \in \Gamma^-_{E' \setminus E}(u)} f(e') = \sum_{e \in \Gamma^+_E(u)} y(e).$$

L'équation 3.18 est équivalente à l'équation 3.20.

$$\sum_{e \in \Gamma^+_E(u)} x(e) - \sum_{e \in \Gamma^+_E(u)} y(e) = \sum_{e \in \Gamma^-_E(u)} x(e) - \sum_{e \in \Gamma^-_E(u)} y(e) \tag{3.20}$$

Nous pouvons conclure de cette dernière équivalence que le flot $f$ sur $G'$ est en bijection avec une possible solution $(x - y)$ pour le problème dual selon les relations 3.16 et 3.17.

De plus, par construction des poids sur les arcs du graphe $G'$ et grâce aux relations 3.16 et 3.17, les deux flots ont le même coût.

Nous avons démontré qu'un flot $f$ pour $G'$ est en bijection avec une possible solution $\begin{pmatrix} x & y \end{pmatrix}$ de $G$ et que ces deux solutions ont le même coût. Les flots $f$ à coût minimal pour $G'$ sont donc en bijection avec les solutions optimales $\begin{pmatrix} x & y \end{pmatrix}$ pour le problème dual. □

### Calcul d'un flot minimum $f$ pour le graphe $G'$

Nous utilisons un algorithme standard pour calculer les flots caractérisés par des capacités et des bornes inférieures à coût minimal (voir [dW90]).

Nous construisons trivialement un flot admissible pour $G'$ afin de pouvoir utiliser l'algorithme. Ce flot satisfait les contraintes de capacités sur les arcs $(v_u, u)$ en choisissant arbitrairement pour chaque nœud $u$ un arc de sortie $(u, v)$ et en plaçant un flot $\{(u, v), (v, v_u), (v_u, u)\}$. L'algorithme de flot à coût minimal introduit un graphe $R^*(f)$ construit à partir du flot $f$ qui sera utilisé dans la suite de la construction.

### Construction des solutions du problème primal à partir de celles du problème dual

Une fois que les solutions optimales pour le problème de flot ont été trouvées, nous devons construire le retiming équivalent pour le problème primal.

La construction de ce retiming est effectuée de la façon suivante : nous considérons le flot optimal $f$ et son graphe associé $R^*(f)$ Nous ajoutons une source $S$ reliant tous les nœuds du graphe $R^*(f)$ avec des arcs de poids nul et nous calculons les plus courts chemins $\pi(u)$ reliant $S$ à tout les nœuds $u' \in V'$ à l'aide de l'algorithme de Bellman-Ford [GM85].

On associe à chaque nœud $u$ la valeur de retiming $r(u)$ et le coût $C(u)$ suivants :

$$r(u) = -\pi(u)$$

$$C(u) = \max_{e=(u,v)\in E} w(e) + r(v) - r(u).$$

Par définition des plus courts chemins, les valeurs $\pi$ trouvées par l'algorithme de Bellman-Ford vérifient les contraintes suivantes :

$$\forall e' = (u, v) \in E',\ \pi(v) \leq \pi(u) + w(e).$$

Ces contraintes se développent sous la forme

$$\forall e' = (u, v) \in E',\ (-\pi(v)) - (-\pi(u)) + w(e) \geq 0.$$

Comme chaque arc $e$ de $E$ appartient aussi à $E'$ avec le même poids, les contraintes 3.8 sont également satisfaites. Nous avons donc

$$\forall e = (u, v) \in E,\ r(v) - r(u) + w(e) \geq 0$$

Les contraintes 3.9 sont vérifiées par définition de $C$. Donc $\left(r \quad C\right)$ est une solution du problème primal.

**Proposition 6** *La solution proposée est optimale pour le problème primal.*

**Preuve** Puisque $(r \quad C)$ et $(x \quad y)$ sont des solutions acceptables pour le problème primal et le problème dual, le théorème des écarts complémentaires [dW90] implique qu'elles sont optimales si et seulement si :

$$r(0 - (-Ax + Ay)) = 0 \tag{3.21}$$
$$C(1 - A^+y) = 0 \tag{3.22}$$
$$(-Ar + w)x = 0 \tag{3.23}$$
$$(Ar + A^+C - w)y = 0 \tag{3.24}$$

Comme $(x - y)$ est un flot pour le graphe $G$ nous avons (équations 3.14 et 3.15) :

$$\begin{aligned} A.(x-y) &= 0 \\ A^+.y &= 1 \end{aligned}$$

Donc les contraintes 3.21 et 3.22 sont vérifiées.
Pour chaque arc $e = (u, v) \in E$, nous avons les deux possibilités suivantes :
- si $f(e) = 0$ alors $x(e) = 0$ et $(r(v) - r(u) + w(e)).x(e) = 0$ ;
- sinon, il existe un arc $(u, v)$ de poids $-w(e)$ et un arc $(v, u)$ de poids $w(e)$ dans $R^*(f)$ car $f(e)$ n'est ni à sa capacité minimale ni à sa capacité maximale. L'algorithme de Bellman-Ford donne donc les relations
$$\begin{cases} \pi(v) \leq \pi(u) + w(e) \\ \pi(u) \leq \pi(v) - w(e) \end{cases}$$
donc $(-\pi(v)) - (-\pi(u)) + w(e) = 0$ et $r(v) - r(u) + w(e) = 0$.

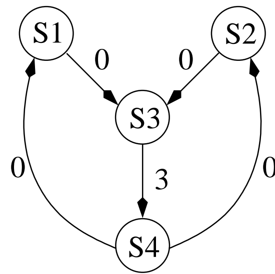
Donc $\forall e = (u, v) \in E, (r(v) - r(u) + w(e)).x(e) = 0$ et l'ensemble de contraintes 3.23 est vérifié.
Pour chaque arc $e = (u, v) \in E$, nous avons les deux possibilités suivantes :
- si $y(e) = 0$ alors $(C(u) - (r(v) - r(u) + w(e))).y(e) = 0$ ;
- sinon il existe un arc de $v_u$ à $v$ dans $R^*(f)$ de poids $w(e)$ par construction de $R^*(f)$. De plus, pour chaque nœud $v' \neq v$ tel que $e' = (u, v') \in E$, nous avons $y(e') = 0$ de la relation 3.15. Nous avons donc un arc de $v'$ à $v_u$ de poids $-w(e')$ dans $R^*(f)$. Ceci implique que l'algorithme de Bellman-Ford trouve les relations
$$\begin{cases} \pi(v) \leq \pi(v_u) + w(e) \\ \pi(v_u) \leq \pi(v') - w(e') \end{cases}$$
Ceci conduit au résultat
$$\pi(v) \leq \pi(v') + w(e) - w(e')$$
qui se transforme en
$$(-\pi(v')) - (-\pi(u))) + w(e') \leq (-\pi(v)) - (-\pi(u)) + w(e),$$
ce qui est équivalent à
$$r(v') - r(u) + w(e') \leq r(v) - r(u) + w(e)$$

Donc, par définition de $C(u)$, nous avons $C(u) = w(e) + r(v) - r(u)$ qui vérifie la relation $(C(u) - (r(v) - r(u) + w(e))).y(e) = 0$ et les contraintes 3.24 sont donc vérifiées.

Nous avons donc montré que $(r \quad C)$ et $(x \quad y)$ sont des solutions pour le problème primal et que l'algorithme du problème dual vérifie le théorème des écarts complémentaires. Les solutions trouvées sont donc optimales pour les deux problèmes. □

La complexité de l'algorithme proposé est $O(|V||E|.(\sum_{e \in E} w(e)))$.

L'exemple que nous avons présenté à la figure 3.8 nécessitait 5 variables temporaires. Sa version optimisée, présentée à la figure 3.11, n'en utilise plus que 3.




*prologue*

```
for(i=2; i<n-2; i++) {
  S4: d[i-1]=2*c[i-1];
  S1: a[i]=2*d[i-1];
  S2: b[i]=3*d[i-1];
  S3: c[i+2]=a[i]+b[i];
}
```

*épilogue*

FIG. 3.11 – Exemple de la figure 3.8 après minimisation des variables temporaires d'itération

L'alignement présenté dans cette section permet d'optimiser une boucle simple pour le critère de localité des calculs défini ici. Les références traitant du décalage d'instructions pour le parallélisme ou la gestion des registres ne proposent pas, ou très rarement, d'extension pour les cas multidimensionnels car les transformations sont appliquées sur les boucles les plus internes des nids. Cependant, le contexte présent des optimisations mémoire *nécessite* une telle extension comme nous allons le voir dans la section suivante.

### 3.4.4 Alignement multidimensionnel

Dans cette partie nous étendons le problème aux nids de boucles. Dans le cas multi-dimensionnel, les dépendances sont représentées par des vecteurs. Une composante $w_i$ d'un vecteur de dépendance correspond à la distance portée par la $i^{ème}$ boucle du nid, en commençant par la boucle la plus externe. Un nid composé de $n$ boucles aura donc des vecteurs de dépendances de dimension $n$.

```
for(i=1; i<N; i++)
  for(j=1; j<M; j++) {
    S1: a[i][j]=c[i][j-1];
    S2: b[i][j]=a[i-1][j]+c[i-1][j];
    S3: c[i][j]=b[i][j-2];
  }
```

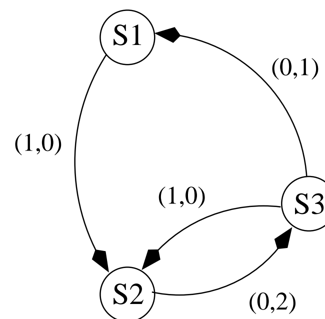



FIG. 3.12 – Modélisation des dépendances pour un nid de boucles

La figure 3.12 montre un exemple de code source ainsi que la représentation de ses dépendances par un graphe.

Les dépendances portant sur les boucles externes sont plus coûteuses que celles portées par les boucles internes car elles représentent une durée de vie plus importante pour les valeurs produites et consommées lors des différentes itérations. Sur l'exemple de la figure 3.12, la dépendance entre `S3` et `S1` est portée par la boucle la plus interne et une seule itération sépare la production de la consommation d'une valeur `c[i,j]`. La dépendance entre `S3` et `S2` est portée par la boucle externe. Le nombre d'itérations séparant la production de `c[i,j]` de sa réutilisation correspond à l'exécution entière de la boucle interne. La dépendance entre `S3` et `S2` coûte donc $M$ (dimension de la boucle interne) fois plus cher que celle entre `S3` et `S1`. Le coût total pour le programme est de $2M + 3$ valeurs.

Étant donné la représentation sous forme de graphe $G = (V, E, w)$ d'un nid de boucles, nous pouvons connaître la taille du stockage temporaire utilisé pour son exécution. Soit $I = (i_1, i_2, \ldots, i_n)$ l'espace d'itération du nid de boucles :

$$i_k = \begin{cases} \prod_{l=k+1}^{n} \text{dimensions}(\text{boucle}_l) \text{ si } 0 < i < n \\ 1 \text{ sinon} \end{cases}$$

Le coût d'un nœud, défini par l'équation 3.6 pour le cas mono-dimensionnel, devient pour le cas multi-dimensionnel :

$$C_m(u) = \max_{e=(u,v) \in E} I.w(e) \tag{3.25}$$

Dans ce cas, pour être correctes les dépendances doivent être positives lexicalement. Nous notons $\geq_{lex}$ l'ordre lexicographique. L'équation 3.8 prise du cas mono-dimensionnel devient maintenant la suivante :

$$w(e) + r(v) - r(u) \geq_{lex} 0, \ \forall e = (u, v) \in E$$

Ces contraintes ne sont pas linéaires et ne peuvent pas être linéarisées, à notre connaissance, sans perdre la totale unimodularité de la matrice et donc la polynomialité du problème.

Nous proposons donc une heuristique efficace pour le problème multi-dimensionnel utilisant l'algorithme trouvé pour le cas mono-dimensionnel. Les transformations sont effectuées en appliquant successivement l'algorithme simple sur chacune des boucles du nid. Comme les dépendances portées par les boucles externes sont les plus coûteuses nous les considérons donc en premier.

### Heuristique pour le cas multi-dimensionnel

L'heuristique proposée pour les accès mémoire dans les nids de boucles transforme les nids boucle par boucle en commençant par la boucle la plus externe. À chaque étape $k$ ($1 \leq k \leq n$), on dispose d'un graphe $G_k = (V, E_k, w_k)$ utilisant des vecteurs de dimension $n-k+1$ et nous appliquons notre algorithme dans la première dimension afin de trouver un retiming $r_k$ permettant de l'optimiser. Nous définissons alors un graphe $G_{k+1} = (V, E_{k+1}, w_{k+1})$ à partir de $G_k^{r_k}$ (graphe $G_k$ auquel nous avons appliqué le retiming $r_k$) de la manière suivante :

$$E_{k+1} = \{e \in E_k \,|w_k(e) \leq_{lex} (0, +\infty, \ldots, +\infty)\}.$$

$w_{k+1}$ est défini à partir de $w_k$ en prenant uniquement les $n-k$ dernières composantes. Nous procédons à l'étape suivante à l'aide de $G_{k+1}$.

**Proposition 7** *Cette heuristique produit un code correct après retiming.*

**Preuve** Supposons que le graphe $G$ soit tel qu'il existe un retiming multidimensionnel légal $r$. Nous voulons prouver que nous pouvons produire un retiming multidimensionnel légal $r_1$ optimisant la première dimension.

Le graphe après retiming a la première composante de toutes ses dépendances positive ou nulle (car il est légal). Donc le retiming $r$ restreint à ses premières composantes et le coût associé constituent une solution possible pour le programme linéaire du cas mono-dimensionnel. Nous pouvons donc appliquer notre algorithme afin de trouver un retiming $r_1$ de $G$ dans la première dimension permettant de minimiser les variables temporaires par rapport à la première boucle.

Nous voulons prouver que ce retiming peut être complété dans les $n-1$ dimensions restantes pour obtenir un retiming multidimensionnel complet de $G$ qui est légal et qui optimise la première boucle. Comme $G^r$ est légal, il ne contient pas de circuit de poids lexico-négatif, et par conservation du poids d'un circuit par retiming, $G$, $G^{r_1}$ et par construction $G' = (V, E', w')$ (graphe de dimension $n-1$ construit à partir de $G^{r_1}$) n'ont pas de circuit de poids lexico-négatif. Étant donné l'ordre lexicographique et l'addition des vecteurs, nous pouvons appliquer un algorithme de type Bellman-Ford [GM85] sur $G'^{r_1}$ permettant de trouver les vecteurs $\pi(u)$ qui vérifient :

$$\forall e = (u, v) \in E', \pi(v) \leq \pi(u) + w'(e)$$

et ainsi

$$w'(e) + (-\pi(v)) - (-\pi(u)) \geq 0,$$

ce qui veut dire que $-\pi$ est un retiming multidimensionnel légal de $G'$ (et puisque, par construction, les arcs non présents dans $G'$ sont portés par la première dimension dans $G^{r_1}$, la composition de $r_1$ et $-\pi$ est légale pour $G$).

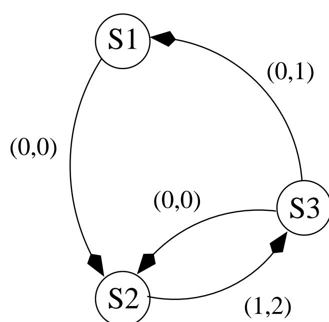
Nous pouvons continuer la procédure sur $G'$ qui est de dimension $n-1$. Comme le graphe de départ est légal car il provient d'un code exécutable, la procédure complète est correcte et fournit un retiming légal optimisé. □

L'application de l'algorithme de Bellman-Ford inclus dans la preuve n'est pas nécessaire car seule l'existence d'un retiming multi-dimensionnel légal est requise, si la procédure d'optimisation n'est pas effectuée jusqu'à la boucle la plus interne, nous devons l'appliquer sur les dimensions restantes afin d'obtenir un retiming légal.

La figure 3.13 montre le code optimisé pour l'exemple de la figure 3.12 pour lequel la première boucle a été alignée. Le coût du graphe transformé est de $M+3$, soit un gain de $M$ éléments par rapport au programme de départ. Les optimisations incrémentales, comme celles que nous proposons, permettent de stopper l'optimisation des accès mémoire selon un compromis pour considérer éventuellement un autre problème d'optimisation. Par exemple, la minimisation de la mémoire est souvent contraire à la maximisation de la parallélisation et trouver un ordonnancement optimal pour la localité des accès mémoire peut conduire à une très mauvaise détection du parallélisme dans les prochaines étapes de la compilation.

### 3.4.5 Conclusion sur l'alignement

Nous avons défini une méthode de transformation efficace permettant de minimiser l'espace de stockage requis pour conserver les valeurs produites et réutilisées entre les itérations d'un nid de boucles. L'optimisation des boucles simples est effectuée en temps polynomial et nous proposons une heuristique pour les nids de boucles permettant de les transformer de manière incrémentale.




```
for(j=1; j<M; j++) {
  b[1][j]=a[0][j]+c[0][j];
}
for(i=1; i<N-1; i++)
  for(j=1; j<M; j++) {
    S1: a[i][j]=c[i][j-1];
    S3: c[i][j]=b[i][j-2];
    S2: b[i+1][j]=a[i][j]+c[i][j];
  }
for(j=1; j<M; j++)
  a[n][j]=c[n][j];
  c[n][j]=b[n][j-2];
```

FIG. 3.13 – Exemple de la figure 3.12 après minimisation des variables temporaires

La transformation de boucles par alignement permet d'augmenter la localité des calculs et a donc une influence sur la taille de la mémoire et la gestion des transactions entre les différents niveaux de la hiérarchie. En effet, comme pour la fusion, cette transformation permet d'améliorer les résultats des techniques de contraction de tableaux et de gestion des mémoires caches. L'alignement modifie les distances des dépendances de données et ainsi permet de réduire la taille des sous-tableaux de valeurs qui seront générés après la contraction. Dans le cas d'une distance de dépendance rendue nulle par alignement, la contraction permet de réduire la dimension des tableaux manipulés.

La génération de code d'une boucle après alignement peut être effectuée soit en créant un prologue et un épilogue, soit en augmentant l'espace d'itération de la boucle et en contrôlant l'exécution des instructions. Ces deux méthodes introduisent un surcoût dans la taille et la complexité du code. Des expérimentations doivent être effectuées afin de mettre en place des métriques permettant d'évaluer ce surcoût de contrôle et mieux pondérer l'alignement.

## 3.5 Combinaison des transformations et conclusion

Les transformations de fusion et d'alignement abordées dans ce chapitre montrent qu'il est possible d'optimiser le code d'une application pour en améliorer la gestion de la mémoire sans connaître a priori la structure exacte de la hiérarchie mémoire. Ces transformations sont présentées, et peuvent être utilisées, de manière indépendantes. Toutefois, une optimisation cohérente et complète de la structure d'un système et de la gestion de la mémoire nécessite une approche globale des transformations de boucles. Il est donc primordial de construire une approche permettant de combiner les transformations de boucles, aussi bien celles présentées dans ce chapitre que celles du chapitre précédent.

Certaines combinaisons sont évidentes, par exemple la combinaison de la fusion pour la minimisation des tableaux temporaires et la fusion pour la maximisation de la localité comme présenté à la section 3.3.7.

La distribution est également une transformation dont on peut intuitivement deviner la com-

binaison avec les autres : tout d'abord effectuer une distribution maximale de toutes les boucles afin d'exhiber les transferts de données entre les nids. On peut alors appliquer les transformations globales et locales sur les boucles, notamment une fusion maximale combinant les critères de taille et de localité. On obtient alors à nouveau des nids de boucles mais dans une organisation différente de celle donnée avant la distribution maximale.

L'échange et l'alignement peuvent aussi être combinés de manière intuitive : une boucle pouvant faire l'objet d'un échange sera alignée de la même manière avant ou après cet échange, l'échange n'a pas d'influence sur les transformations possibles d'alignement (un échange modifie le niveau de la boucle portant les dépendances mais ne modifie pas les bornes des boucles ni les dépendances). L'inverse n'étant toutefois pas vrai, un alignement pouvant bloquer les possibilités d'échange de boucles dans un nid par modification des dépendances et aussi en partie à cause de la création des prologues et épilogues (le nid de boucles n'est plus parfait après un alignement). Il convient donc d'effectuer l'échange avant l'alignement lorsque les deux méthodes doivent être utilisées.

Toutefois, les combinaisons entre certaines transformations, par exemple la fusion et l'échange ou la fusion et l'alignement ou encore les trois ensembles, restent un problème ouvert pour lequel les compromis à réaliser et l'ordre d'application des transformations sont obscurs. Par exemple, l'alignement proposé à la figure 3.13 modifie la dépendance sur le tableau `b` en l'avançant et en calculant `b[i+1]` pour une itération `i`. Cette modification empêche une fusion avec une autre boucle si celle-ci utilise la valeur `b[i]` dans ses itérations créant ainsi un arc de dépendance de flot inversé. Ce type de problème peut être résolu en utilisant une fusion décalée où l'on décale toutes les itérations d'une boucle jusqu'à ce que la fusion soit rendue légale. Cependant, nous ne pouvons contrôler la valeur maximale du décalage. De plus, les compromis et les critères de choix entre la fusion décalée et la taille des prologues et épilogues est un problème ouvert. D'un autre côté, la fusion de deux boucles ne crée pas de cycle de dépendance au niveau des instructions dans le corps de la nouvelle boucle. Les alignements qui étaient possibles de manière indépendante sur les boucles avant la fusion restent donc valides et peuvent être effectués sur la nouvelle boucle. Néanmoins, dans le cas de la fusion de nids de boucles, les prologues et épilogues sont complexes et de grandes tailles. Il est alors difficile de préciser si la transformation obtenue en considérant la fusion avant l'alignement est d'un gain supérieur à celle obtenue en utilisant l'ordre inverse. On voit là que pour mieux maîtriser ces deux transformations il faut mettre en place des critères provenant de cas pratiques au moyen d'expérimentations.

Lorsque les paramètres de la hiérarchie mémoire sont un peu plus détaillés (notamment la taille des caches et des lignes de cache) l'utilisation de fonctions de coût plus complexes permet de mettre en œuvre les transformations décrites dans le chapitre 2 (notamment le pavage) et de guider plus facilement les combinaisons [Car93, MCT96]

Néanmoins, l'optimisation de la conception d'un système avec une approche globale des transformations de boucles constitue un problème extrêmement complexe. Ceci vient en partie de la taille gigantesque de l'espace de recherche offert par les combinaisons possibles entre matériel et logiciel que seul le concepteur est à même de pouvoir orienter. Dans ce contexte, il apparaît difficile pour l'approche de transformations de boucles proposée par IMEC —présentée à la section 2.4— de pouvoir s'adapter à un contexte de codesign et de partitionnement. En effet, l'approche est à la fois trop et trop peu automatisée. Trop peu automatisée car elle nécessite des étapes manuelles (le *pruning*, les transformations globales sur le flot de données et la réécriture en assignation unique) dont l'automatisation semble être un problème extrêmement complexe à résoudre. Trop automatisée car les représentations des données et les transformations mises en œuvre sont inaccessibles pour un concepteur qui n'est pas forcément un expert en compilation et qui pourtant doit pouvoir

relire et manipuler le code lors du partitionnement. En effet, le concepteur est le seul à pouvoir, à ce niveau, considérer des contraintes non quantifiables de manière formelle comme la lisibilité, la réutilisation de portions de code ou de composants matériels ou encore l'évolution et la maintenance des programmes. Cependant, dans un contexte de compilation hors partitionnement, les transformations apportées par la méthodologie sont efficaces et peuvent être utilisées avec profit dans un outil de CAO. Les modèles et critères définis dans la méthodologie pour les transformations de boucles sont une base de travaux futurs sur l'automatisation des transformations dans un environnement interactif.

Pour définir plus précisément les critères de combinaison des méthodes, pour valider les transformations proposées et pour percevoir une méthodologie d'aide à la transformations de boucles, il faut réaliser des expérimentations de code d'applications réelles. Une collaboration avec les équipes de recherche sur la conception de systèmes de la société Cadence, à Sophia Antipolis et à Chelmsford dans le Massachussets, nous a permis d'utiliser le logiciel VCC et d'effectuer des tests d'intégration dans la méthodologie proposée par cet outil. Elles sont proposées au prochain chapitre.

# Chapitre 4

# Mise en pratique et intégration des transformations

Les critères et transformations que nous avons présentés dans les chapitres précédents sont conçus pour optimiser la gestion et la consommation d'un système. Le présent chapitre concerne l'application de ces méthodes formelles dans un cadre pratique.

Plusieurs problèmes sont encore ouverts pour l'application des transformations de boucles dans les méthodologies de codesign. Un premier problème concerne le réalisme des fonctions de coût et la prise en compte de l'environnement dans lequel on effectue les transformations : « comment définir des critères et des contraintes adaptés et pouvant être pris en compte dans les outils ? » ; « comment influencer et diriger les étapes de codesign en fonction des transformations effectuées ? » sont autant de questions qu'il est important de poser avant l'intégration des méthodes de transformation dans un outil. Comme on l'a vu en conclusion du chapitre 3, une autre question importante concerne l'organisation des transformations et leurs combinaisons dans une approche globale d'optimisation.

Deux types d'outils étaient disponibles pour effectuer cette démarche d'intégration : Acropolis d'IMEC et VCC de Cadence.

L'outil Acropolis [IME], développé à IMEC, propose un environnement de compilation automatique pour la synthèse d'architectures dédiées ou sur plateformes fixes. Cet outil, bien qu'ayant un bon nombre d'étapes de la méthodologie DTSE implémentées et fonctionnelles, n'est pas encore prêt pour ce type d'intégration car la phase de transformations de boucles, décrite au chapitre 2 (page 51) n'est pas encore liée aux autres phases de la méthodologie DTSE.

Une collaboration avec les équipes de recherche de la société *Cadence Design Systems* [Cad], grâce à Laurence Just-Meunier, nous a permis de confronter les critères et les optimisations décrits dans cette thèse avec la méthodologie utilisée dans l'outil de conception conjointe matériel/logiciel *VCC*. Cet outil propose un atelier logiciel permettant de modéliser, de simuler et d'estimer le coût d'un système intégré. Il n'intègre cependant pas d'outils d'optimisation automatique ou d'aide à la prise de décision. Les choix sont donc pour l'instant laissés au concepteur.

La prochaine section présente l'outil de conception VCC et l'approche méthodologique choisie pour la conception de systèmes dans ce contexte, nous y validons les fonctions objectives proposées dans cette thèse. La section 4.2 concerne les interactions entre les transformations automatiques de

code et les interactions entre les modifications d'un comportement et les étapes de partitionnement et d'assignation dans le contexte de VCC.

## 4.1 Cadence *VCC : Virtual Component Codesign*

Le logiciel VCC suit la tendance actuelle de développement des SoC et propose un environnement intégré permettant une conception au niveau système. La construction d'un système est faite selon une approche dite *Platform Based Design – PBD* [CCH+99] utilisant la réutilisation de composants complexes de grande taille. L'environnement permet d'intégrer des modèles de propriété intellectuelle (*Intellectual Property – IP*) et de simuler, évaluer et sélectionner les composants virtuels appropriés pour des systèmes multimédia, de communication, ou encore des systèmes embarqués dans des engins mobiles. L'environnement différencie le comportement (devant déterminer ce que fait le système) de l'architecture (devant déterminer comment il fonctionne).

La méthodologie de *codesign* mise en place est similaire à celle présentée au chapitre 1 (page 11) et permet de mettre au point de manière conjointe les parties logicielles et matérielles d'un système afin de satisfaire à la fois les contraintes de coût et de performances comme le présente la figure 4.1. Une implémentation uniquement logicielle peut être effectuée à un coût (financier ou en temps de conception) minime mais n'offrira pas les performances souhaitées (en vitesse, consommation ou place). D'un autre côté, une implémentation uniquement matérielle permet d'obtenir de très bonnes performances au prix d'un coût de revient très élevé. La partie grisée de la figure représente la zone pour laquelle les contraintes de coût et de performances sont satisfaites au moyen d'un partitionnement efficace entre matériel et logiciel.

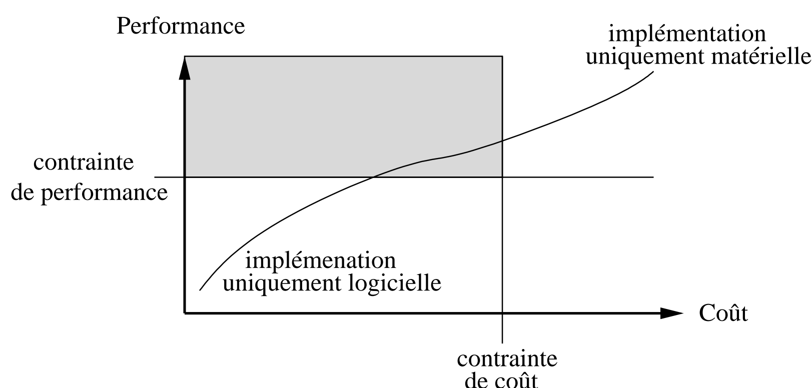



FIG. 4.1 – Espace de recherche entre coût et performances

La construction d'un système se fait à partir de blocs pouvant être prédéfinis. La sélection et la mise en place des constituants d'un système se découpe en quatre étapes :
- construction d'un prototype fonctionnel du système ;
- définition de la plateforme architecturale ;
- liens entre fonctions et architecture ;
- simulation et analyse des performances dans un environnement graphique.

### 4.1.1 Définition des blocs comportementaux

L'éditeur de comportement permet de capturer de façon non ambiguë les spécifications à un haut niveau d'abstraction en utilisant des modèles provenant de bibliothèques, exportés d'autres outils de conception ou définis par l'utilisateur. La spécification d'un comportement est indépendante de son implémentation matérielle éventuelle.

La composition d'un système est effectuée par un assemblage de blocs comportementaux. Ces blocs peuvent être décrits en utilisant différents modèles : langages C, C++ ou SDL, diagrammes d'états (*State Transition Diagram – STD* ou *Control Finite State Machine – CFSM*), ou des outils d'analyse et de traitement du signal comme SPW.

Les modèles comportementaux représentent les blocs fonctionnels et les tests. Un comportement ne peut être assigné qu'à une seule unité architecturale. Un comportement peut être composé de manière hiérarchique afin de pouvoir être assigné sur un seul composant architectural ou de manière répartie sur différentes ressources. Les blocs sont connectés par des ports dont le comportement est spécifié au moyen de primitives de communication (*communication pattern*).

La figure 4.2 montre la composition comportementale d'un client de messagerie vocale (*voice mail pager*). Les blocs sont répartis selon les fonctionnalités principales : les parties encadrées sur fond grisé sont des périphériques externes (interface utilisateur, haut parleur, antenne et système de réception) ; les blocs comportementaux restants représentent la gestion de la couche physique, la pile de protocole utilisée et un module de traitement de la voix. Ce sont eux qui font l'objet de la conception.

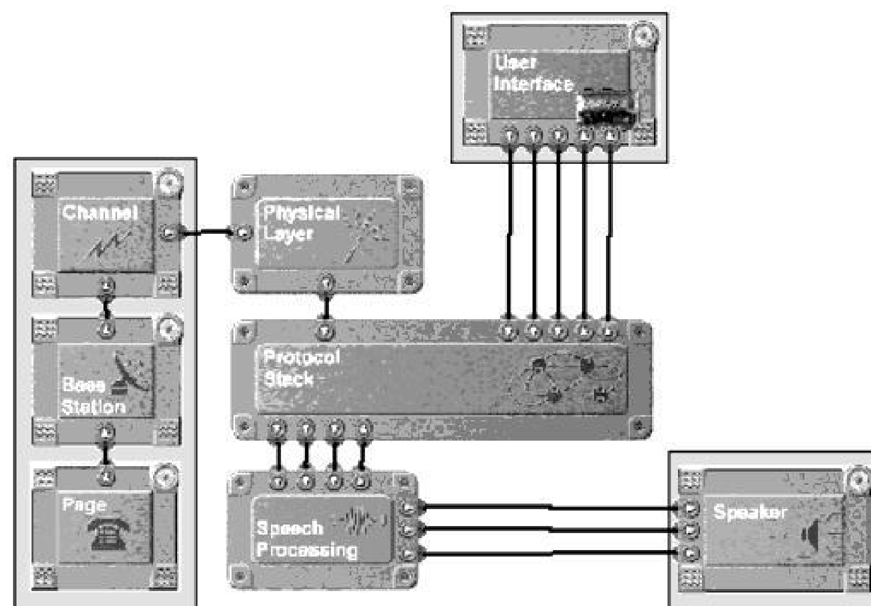



FIG. 4.2 – Diagramme comportemental d'un système de messagerie vocale

Le langage utilisé pour modéliser les comportements dépend du type d'application et de la provenance du code :

- importés depuis SPW pour des comportements de traitement de signaux numériques.
- importés ou crées dans l'outil pour les machines à états finis (CFSM ou STD).

- C pour les applications embarquées (*white box C*)
- C++ pour les blocs de fonctionnement généraux (*black box C++*)

Dans la suite de ce chapitre, nous utilisons les modèles décrits par des boîtes blanches écrites en C. Ce type de description a été choisi en raison des possibilités d'analyses offertes par l'outil permettant d'avoir une simulation complète des accès à la mémoire, un calcul de la taille mémoire des programmes et une gestion des différents segments (code, pile d'exécution, segments de données).

### 4.1.2 Définition des blocs architecturaux

L'éditeur de diagrammes architecturaux permet de mettre en place une architecture cible abstraite. Ce mode de spécification permet de construire la plateforme cible à l'aide de composants matériels et logiciels de haut niveau et de chemins de communication provenant de vendeurs externes ou de bibliothèques de composants. Seule la topologie du système est importante à ce niveau et ne nécessite pas la définition détaillée de chaque port ou signal matériel.

La variété des modèles supportés permet de considérer tous les types de ressources utilisables dans une architecture tels que des unités de calculs (cœurs de processeurs génériques, de DSP ou d'ASIP), des bus de communication, des mémoires, des parties matérielles dédiées (ASIC), et des noyaux de systèmes d'exploitation temps-réel. Ces blocs sont connectés par des ports de communication sur les bus appropriés et sont définis par un modèle de performances et des caractéristiques d'implémentation. Un modèle de performances doit être défini pour chaque type de composant. Par exemple, un processeur est caractérisé par les délais des instructions exécutées. Afin de permettre une évaluation rapide, les composants sont décrits dans des modèles de niveau d'abstraction supérieurs aux niveaux d'implémentation tels que C ou VHDL et Verilog. Un modèle architectural représente un modèle de performances pouvant être utilisé pour analyser l'implémentation d'un comportement. Les modèles de performances peuvent être précaractérisés à partir d'une implémentation réelle ou bien définis par l'utilisateur et caractérisés par des délais représentant les performances attendues définies à partir des contraintes.

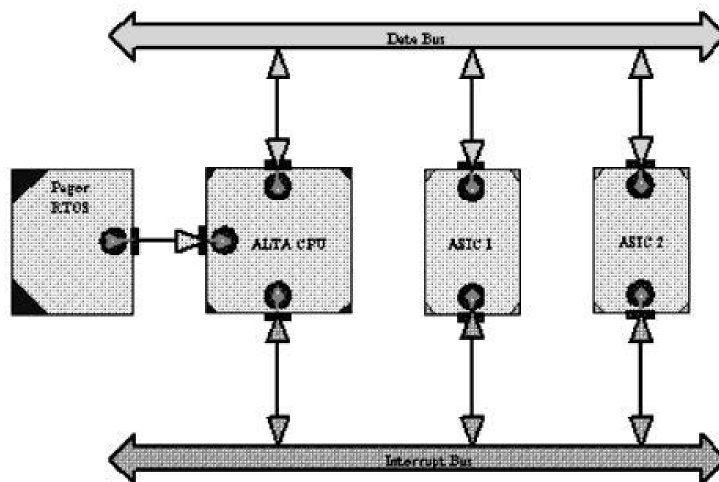



FIG. 4.3 – Architecture cible d'un système de messagerie vocale

La figure 4.3 représente l'architecture retenue pour le système de messagerie vocale. Il est composé d'un ordonnanceur (RTOS), d'un processeur générique (Alta CPU) et de deux ASIC reliés par

un bus de données (Data Bus) et un bus d'interruption (Interrupt Bus). Un système de gestion est modélisé par un ordonnanceur simple.

### 4.1.3 Assignation ou *Mapping*

L'éditeur de diagrammes de *mapping* permet au concepteur d'associer une fonctionnalité à une ressource de l'architecture cible et d'identifier les chemins de communication requis. L'assignation définit le partitionnement entre matériel et logiciel. Par exemple, un comportement assigné à un ordonnanceur et un processeur sera implémenté en logiciel alors qu'un ASIC correspond à une implémentation matérielle.

La figure 4.4 présente l'assignation du comportement du système de messagerie vocale sur son architecture. Les modules présentés à la figure 4.2 sont des blocs hiérarchiques. Le mapping est effectué à partir des vues de détail de ces blocs. On peut voir que la gestion de la couche physique a été assignée au premier ASIC, le traitement du signal vocal est réparti sur les deux ASIC du système et la pile de protocole est gérée de façon logicielle par le processeur. Certaines communications entre les blocs sont précisées et sont assignées au bus de données ou au bus d'interruption.

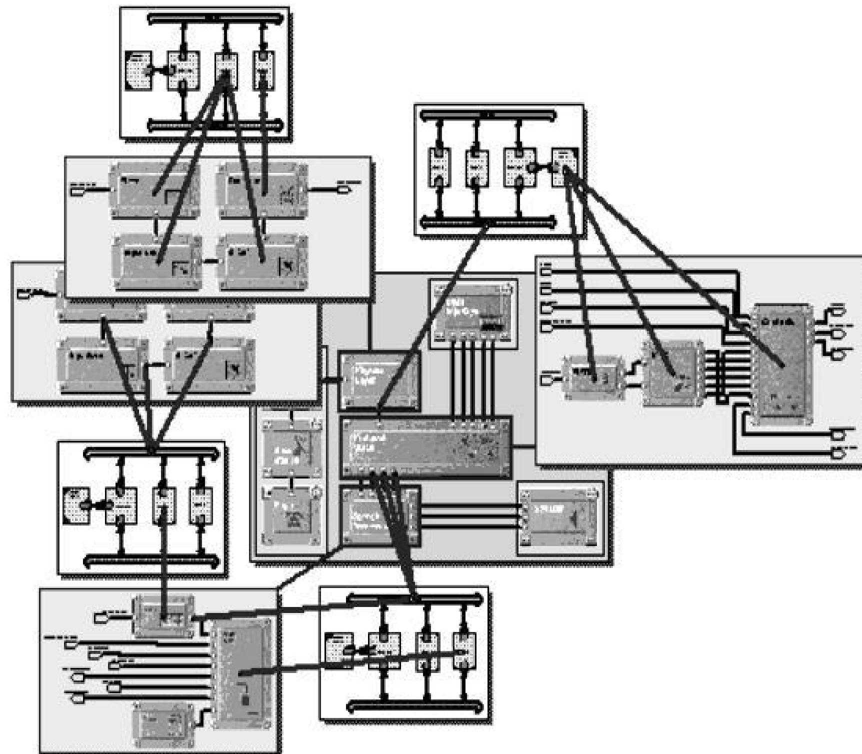


FIG. 4.4 – Assignation du comportement pour le système de messagerie vocale, l'architecture cible est répétée pour plus de clarté

L'assignation définit le partitionnement entre matériel et logiciel. Elle spécifie aussi les modèles de performances permettant de simuler et d'estimer les délais pour le comportement exécuté sur l'architecture.

### 4.1.4 Simulations d'un système

Une fois l'assignation complétée, le concepteur peut simuler et évaluer le système complet. Un simulateur permet de quantifier le tout en simulant les performances de chacun des comportements de manière asynchrone. La modélisation des performances est fournie par des délais. Le trafic des données est simulé de façon complète ou à l'aide de modèles (*memory access patterns*) sur les bus et les différents modules de mémoire présents dans le système. Les mémoires et les caches sont instanciés à partir de la bibliothèque de composants de VCC. Les transferts entre les différents modules sont modélisés grâce aux services architecturaux (*architecture services*) définissant le fonctionnement des modules.

#### Simulation fonctionnelle

La simulation fonctionnelle permet de vérifier de manière rapide la bonne mise en place du diagramme comportemental. Elle permet de déboguer un système avant de commencer des estimations de performances beaucoup plus gourmandes en temps de calcul. Les communications entre modules dans ce type de simulation sont effectuées par passage de messages.

#### Simulation de performances

La simulation de performances est le mode que nous allons utiliser pour les estimations des sections suivantes de ce chapitre. Ce mode de simulation permet de générer un système complet et de le simuler par instrumentation de code ou avec l'aide de simulateurs externes.

Lorsqu'un comportement est assigné à un processeur, et donc traité comme une partie logicielle, son estimation de performance est obtenue grâce à une instrumentation automatique du code source (disponible seulement pour les modèles décrits en C) permettant une analyse de celui-ci. Le calcul de la taille mémoire est effectué en fonction des variables déclarées de manière statique dans le code et correspond au critère que nous avons défini pour les transformations de boucles dans les sections 2.2.1 (page 2.2.1) et 3.1 (page 69).

L'instrumentation automatique rajoute des annotations au programme permettant de simuler son exécution sur une machine abstraite ayant un jeu d'instructions réduit. Une table, définie selon les modèles de processeurs, permet de faire une estimation de la durée d'exécution d'un programme en fonction du délai estimé de chaque instruction du jeu réduit. La granularité des transactions vers la mémoire correspond aux accès en lecture et en écriture détaillés au niveau des instructions. Les hypothèses utilisées pour les fonctions objectives sur la localité temporelle des calculs définie à la section 3.1 (page 69) sont donc utilisables dans ce contexte d'évaluation de haut niveau car elles prennent en compte toutes les informations proposées par l'outil et correspondent bien au niveau d'abstraction.

De plus, le compilateur génère des segments différents pour le code, les données initialisées (*bss*), la pile d'exécution (*heap*) et les données non initialisées (*data*). Ces segments peuvent être placés dans des mémoires architecturales différentes (par exemple sur une ROM pour le segment de code et sur un module de mémoire standard pour les autres segments). Des segments supplémentaires peuvent être définis et des directives permettent d'associer une variable à un segment particulier. Cette gestion de l'espace d'adressage permet de différencier les transactions reportées sur les bus

et les mémoires caches selon la nature des données transportées. On peut ainsi avoir une analyse séparée pour le code du programme et les transferts de données.

Pendant la simulation, VCC fournit le routage des transferts depuis une unité de calcul vers la mémoire référencée. Chaque transaction mémoire est enregistrée et simulée de bout en bout dans la hiérarchie mémoire.

Les mémoires caches sont également simulées de manière exacte. Leur fonctionnement est paramétré au niveau architectural selon les caractéristiques présentées à la section 1.3.4 (page 29). Ces paramètres sont :
- la taille totale ;
- la taille et le nombre de lignes ;
- le degré d'associativité ;
- les algorithmes de gestion des remplacements et des écritures.

La simulation permet donc d'avoir une estimation fiable des transferts mémoire d'un système et de comparer les différentes solutions comportementales ou architecturales du point de vue de la gestion de la mémoire et des transactions d'un système.

### 4.1.5 Analyse des simulations

L'environnement de VCC offre des moyens de visualisation des résultats des simulations de comportement ou de performances. L'analyse permet de mesurer les critères suivants :
- temps d'exécution ;
- taille des mémoires générées ;
- courbes de charges et d'activation des bus ;
- transactions sur la hiérarchie mémoire.

Les transactions mémoire sont détaillées selon leur taille, le type de segment utilisé et la direction (lecture ou écriture). Dans le cas d'un cache mémoire, les résultats permettent de connaître également, selon les segments utilisés, le nombre de succès et le nombre de fautes qui ont eu lieu durant la simulation.

En utilisant les informations fournies par la simulation, un concepteur peut modifier le système jusqu'à trouver un compromis satisfaisant toutes les contraintes.

Nous nous intéresserons donc aux simulations de performances pour caractériser le fonctionnement d'un comportement en fonction du mapping. Les techniques développées pour les estimations de performances dans VCC correspondent aux critères d'optimisation que nous avons définis de manière théorique et confirment donc que les fonctions objectives pour les transformations sont utilisables dans le contexte de l'outil. Bien que les critères utilisés pour VCC, pour la méthodologie DTSE d'IMEC et dans cette thèse aient été développés de manière indépendante, ils semblent converger et correspondent au même niveau d'abstraction utilisé pour les transformations.

## 4.2 Expérimentations avec VCC : estimations et optimisations pour la mémoire

Cette section présente l'utilisation des transformations pour l'optimisation de la mémoire d'un bloc comportemental. Nous nous intéressons aux interactions entre l'outil et les transformations

automatiques à ce niveau de conception et à leur influence possible sur l'architecture et le mapping du système.

L'exemple du client de messagerie vocale (figure 4.4) propose une décomposition du bloc de gestion de la couche physique en quatre sous-modules (un filtre, un égaliseur, un échantillonneur et un décodeur) utilisés les uns à la suite des autres sur le signal d'entrée. Une fois un bloc fonctionnel hiérarchisé, les transferts entre chaque sous-fonction (module) sont figés et ne pourront être réduits. La description d'un comportement doit donc être transformée tant que l'on dispose d'une vue globale sur l'organisation des transferts générés. Le modèle de conception de système proposé par VCC privilégie la réutilisation de composants. Cette réutilisation, pour être faite efficacement, doit assurer une communication facile entre les modules et utilise des interfaces bien définies. Il n'est donc pas possible dans ce fonctionnement d'optimiser les transferts mémoire entre les blocs fonctionnels une fois que ceux-ci sont définis. L'utilisation des transformations d'optimisations de boucles est donc limité aux cas où le concepteur doit créer un bloc, éventuellement hiérarchique, à partir de sa description comportementale. Une fois les optimisations pour la mémoire effectuées, le bloc peut être hiérarchisé afin de pouvoir être éventuellement réparti sur plusieurs unités architecturales.

La figure 4.5 représente le diagramme comportemental que nous utilisons pour les transformations. Ce diagramme est composé d'un bloc principal contenant le code à optimiser et deux blocs de tests : un bloc d'initialisation (*Init*) et un bloc activé à la fin de la simulation (*Sink*).

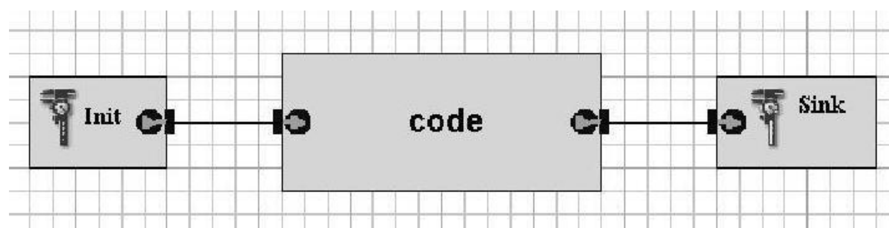



FIG. 4.5 – Diagramme comportemental utilisé pour les tests

```
#define N 100
#define L 10
int a[N][N];
int b[N][N];
```

```
void compute(void) {
  int i,j,k;
  for(i=0; i<N; i++)
    for(j=0; j<=N-L; j++)
      b[i][j] = 0;
  for(i=0; i<N; i++)
    for(j=0; j<=N-L; j++) {
      for(k=0; k<L; k++)
        b[i][j]+=a[i][j+k];
    }
}
```

FIG. 4.6 – Exemple de départ pour les transformations

Nous considérons un exemple simple de code C, donné à la figure 4.6 pour modéliser le comportement du bloc principal. La prochaine section présente la mise en place des optimisations indépendantes de l'architecture cible. La section 4.2.2 suivante présente les transformations conjointes

entre comportement et architecture dans l'outil.

### 4.2.1 Optimisations indépendantes de l'architecture

Un premier mapping est réalisé sur une architecture, représentée à la figure 4.7, comportant un processeur générique, une mémoire cache, une mémoire RAM et deux bus de communication. Les paramètres de l'architecture sont fixés pour les besoins de la simulation mais les transformations utilisées sur le code sont faites indépendamment de ceux-ci.

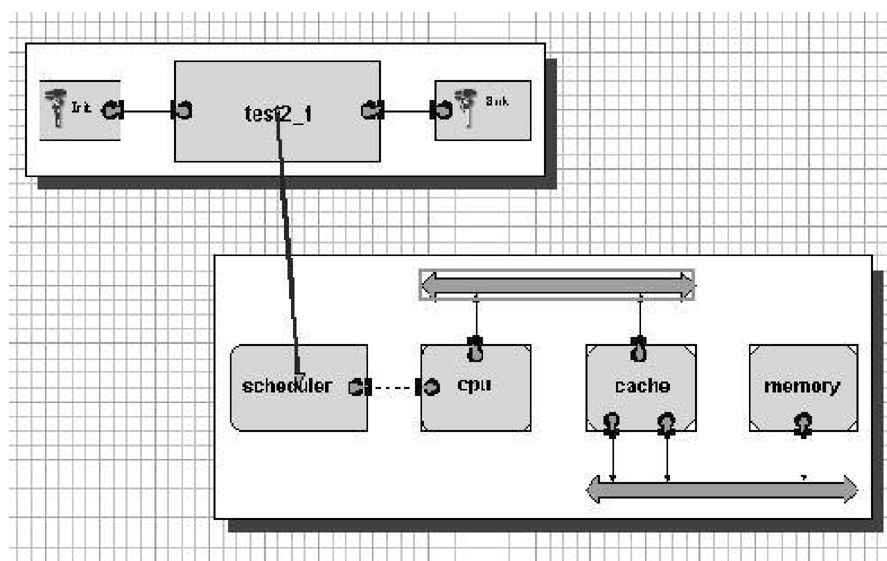



FIG. 4.7 – Diagramme de l'architecture cible et du mapping utilisé

La place mémoire des données estimée pour ce programme est de 20 000 entiers. Tous les segments (code, pile et données) sont assignés à la mémoire principale. Le code de départ, une fois simulé dans sa totalité génère 373 100 lectures et 100 100 écritures (données uniquement) sur le bus entre le processeur et la mémoire cache. Le nombre de transactions entre la mémoire cache et la mémoire principale est lui de 4700 lectures et 4600 écritures (la taille des lignes de cache est fixée à 4 mots et ce sont donc 188 000 et 184 000 mots qui transitent réellement sur le bus).

La figure 4.8 présente le code de la figure 4.6 une fois transformé par fusion de boucles. L'activité générée par le code sur le bus entre le processeur et la mémoire cache est inchangée. En effet, seule la localité temporelle est modifiée dans le programme, le nombre d'opérations reste le même qu'avant la transformation. La transformation a une influence sur le nombre de transferts entre la mémoire cache et la mémoire principale. L'activité sur ce bus y est réduite par un facteur 2 (2400 lectures et 2300 écritures) sans aucun changement de paramètre dans la hiérarchie. L'opération de fusion n'est donc bénéfique que si l'architecture est hiérarchisée comme nous l'avons supposé dans la définition des critères présentés à la section 3.1.

```
void compute(void) {
  int i,j,k;
  for(i=0; i<N; i++)
    for(j=0; j<=N-L; j++) {
      b[i][j] = 0;
      for(k=0; k<L; k++)
        b[i][j]+=a[i][j+k];
    }
}
```

FIG. 4.8 – Exemple de départ après fusion de boucles

### 4.2.2 Modification de l'architecture et du comportement

Une fois les transformations indépendantes de l'architecture effectuées, les transformations de code peuvent faire apparaître de manière explicite la gestion de la hiérarchie mémoire et influencer ainsi l'architecture et les paramètres choisis.

La méthodologie DTSE d'IMEC propose des étapes de transformations ayant une influence sur l'architecture. Nous présentons ici l'étape de réutilisation des données et son influence sur les possibilités offertes au concepteur pour modifier la structure de l'architecture cible ainsi que les paramètres des mémoires caches. Le code de la figure 4.9 présente le code de la figure 4.8 une fois l'étape de réutilisation des données (*Data Reuse*) effectuée.

```
#pragma segment bss
int a[N][N];
int b[N][N];
#pragma endsegment
#pragma segment data
int a_buf[L];
int b_buf;
#pragma endsegment
```

```
void compute(void) {
  int i,j,k;
  for(i=0; i<N; i++)
    for(j=0; j<=N-L; j++) {
      b_buf = 0;
      a_buf[(j+L-1)%L]=a[i][j+L-1];
      for(k=0; k<L; k++)
        b_buf+=a_buf[(j+k)%L];
      b[i][j] = b_buf;
    }
}
```

FIG. 4.9 – Exemple de départ après fusion et réutilisation de la mémoire

Cette étape de transformation introduit des variables de stockage supplémentaires (`a_buf` et `b_buf`) correspondant à des copies locales des parties des données ayant une fréquence d'utilisation élevée. Le code est donc transformé pour y faire apparaître explicitement l'utilisation d'une hiérarchie mémoire. L'utilisation de copies explicites augmente la quantité de mémoire utilisée par le programme (44 mots dans l'exemple) ainsi que la taille du code généré. Les variables supplémentaires sont destinées à être placées sur des niveaux élevés de la hiérarchie mémoire. Le nombre de transferts supplémentaires visibles au niveau du code source par les copies dans les tableaux `a_buf`

et b_buf correspond aux transferts entre la mémoire principale et les niveaux dans lesquels seront placés ces tableaux. Ces transferts étaient également nécessaires dans le code de la figure 4.8 mais n'étaient pas détaillés de manière explicite. Une simulation du code de la figure 4.9 sur la même architecture que celle utilisée pour les autres exemples montre en effet que le nombre de transactions sur les différents bus de données est le même que celui obtenu avant la transformation.

Le contrôle des transferts à partir du code permet d'avoir deux approches pour modifier l'architecture du système. La première permet de construire une hiérarchie mémoire dédiée à l'application. Une deuxième méthode conserve l'architecture mais permet d'ajuster ses paramètres, notamment les paramètres des mémoires caches.

### Modifications de l'architecture

Une mémoire locale supplémentaire (voir section 1.3.4, page 32) est allouée hors du cache et connectée directement au bus entre le processeur et la mémoire cache comme présenté à la figure 4.10. Les transactions effectuées sur les bus de données sont les mêmes que pour la cas précédent mais l'utilisation d'une petite mémoire (44 mots dans le cas présent) permet de réduire l'activité de la mémoire cache de 92%. Cette réduction d'activité offre un gain très important en consommation car la gestion des requêtes effectuées dans une mémoire cache est beaucoup consommatrice d'énergie que l'utilisation d'une « petite » mémoire statique. Il existe de manière évidente des compromis entre la taille de la mémoire rajoutée dans le système et la complexité de la gestion de la mémoire cache, ces compromis appartiennent au concepteur qui, à ce niveau de conception, est le seul à pouvoir maîtriser ces informations. Le lecteur intéressé par les copies explicites et l'utilisation de mémoires auxiliaires pourra consulter [CWG+98] et [PDN99b].

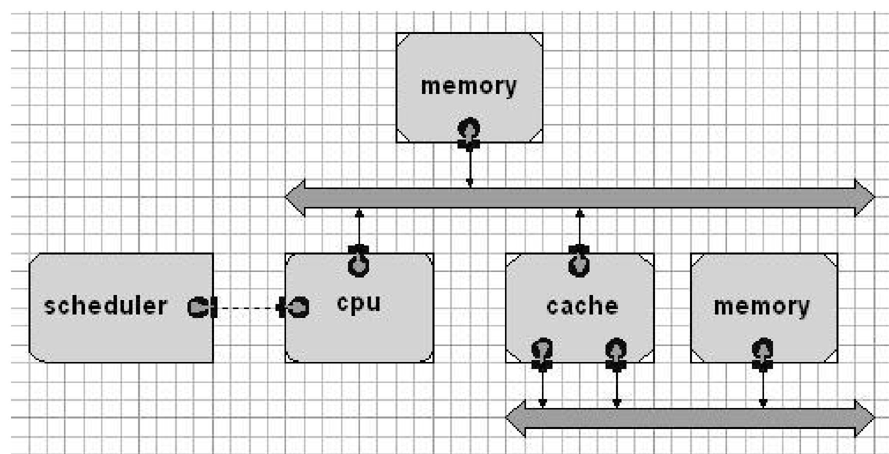



FIG. 4.10 – Diagramme d'une architecture possible utilisant une mémoire statique locale

### Modifications des paramètres architecturaux

Une autre utilisation des copies explicites permet de simplifier la gestion des caches de données. En effet, dans le cas de l'utilisation d'un cache logiciel, les copies explicites représentent les données qui devront être chargées dans le cache puis libérées après utilisation. Dans le cas de l'utilisation

d'un cache matériel, l'utilisation de variables positionnées à des emplacements fixes de la mémoire (adresses réelles) permet de limiter les effets des conflits d'adressages pour les caches non totalement associatifs. Dans l'exemple présenté, les paramètres de la mémoire cache utilisée pour l'architecture cible ont pu être modifiés en réduisant sa taille par un facteur 4 (passage de 256 lignes de caches à 64 lignes) et en utilisant une associativité directe (au lieu d'un associativité par voies de 2 lignes) sans modifier le nombre de fautes de caches générées pendant la simulation.

La simplification des caches peut également être faite en utilisant des politiques de remplacement et de réécriture plus simples. La réduction du nombre de lignes du cache permet d'obtenir un gain en taille et en consommation. La simplification du fonctionnement des caches permet de diminuer la consommation électrique générée par les transactions.

## 4.3 Conclusion

Des expérimentations de plus grande taille n'ont pas, pour l'instant, pu être effectuées à cause de problèmes sur la gestion des transactions existant dans l'outil VCC : le modèle de cache génère des transactions erronées sur son interface de communication avec la mémoire. Il faut donc quelques modifications de l'outil avant de pouvoir valider les méthodes présentées dans cette thèse et mettre en place des métriques permettant de combiner les transformations sur des exemples d'applications.

Cependant, les différents critères et fonctions objectives, définis et utilisés dans la méthodologie DTSE, l'outil VCC et cette thèse, convergent vers un même niveau d'abstraction.

Ils s'associent naturellement pour former un outil de conception de systèmes intégrés semi-automatisé pour l'optimisation de la mémoire. Les transformations indépendantes de l'architecture permettent de mieux cerner l'organisation du système et gardent tout le potentiel des transformations de la DTSE lorsqu'on l'applique sur un comportement matériel ou logiciel.

L'intégration des transformations automatiques dans l'outil VCC peut être effectuée par des ajouts de composants logiciels externes sans changer le principe de la méthodologie. En effet, VCC propose des interfaces de programmation permettant de manipuler la description d'un système, d'effectuer des simulations utilisant des jeux de paramètres différents et de récupérer les résultats obtenus. La mise en place des modifications des comportements et des architectures peut donc être faite sans modifier l'outil déjà existant et s'intègre naturellement dans la philosophie de VCC.

Pour l'instant, cet outil fournit une aide précieuse pour la mise en place des composants, l'estimation des performances et le dimensionnement des modules comportementaux et architecturaux d'un système. Il ne propose pas encore de transformations ou d'optimisations automatiques pour la compilation ou la validation. Cependant, la méthodologie est définie de manière précise et des recherches futures permettront de compléter l'outil.

# Chapitre 5

# Conclusion et perspectives pour la synthèse de systèmes

Les transformations présentées dans cette thèse permettent d'avoir une influence importante sur la mémoire d'un système sur silicium. La méthodologie proposée par IMEC, dont nous avons développé une étape de transformation au chapitre 2, propose une compilation efficace d'un programme à la fois pour une architecture cible fixe et pour une architecture mémoire dédiée. Les transformations présentées au chapitre 3 peuvent être utilisées dans le cadre d'un outil semi-automatique afin de mettre en évidence et d'optimiser un système par rapport aux besoins relatifs à la mémoire. Ces transformations sont guidées par des fonctions objectives indépendantes des paramètres de l'architecture mémoire qui est uniquement supposée hiérarchisée. On peut ainsi mettre en place des optimisations du comportement d'une application de manière indépendante de la cible architecturale sur laquelle elle sera assignée. Ces optimisations ont une influence sur l'organisation et les paramètres de l'architecture mémoire, les communications et l'assignation des mémoires dans la hiérarchie. De nombreuses questions sont encore ouvertes concernant les transformations de programme. Quelles autres transformations sont possibles lorsque l'on ne dispose d'aucune information sur l'architecture cible et comment doit-on (peut-on) les combiner ? Quels sont les compromis à faire entre lisibilité et maintenance du code généré et efficacité des optimisations ? Ces questions devront être étudiées au moyen d'expérimentations sur des exemples d'applications réelles et en collaboration avec des concepteurs.

Les techniques et les outils sont, en général, définis autour de langages de spécifications et de leurs méthodologies sous-jacentes. Les transformations de boucles présentées dans cette thèse utilisent des graphes de flot de données synchrones comme représentation des comportements. Cette représentation permet d'utiliser des formalismes mathématiques pour assurer la validité des transformations effectuées. Cependant, comme c'est fréquemment le cas en compilation, on ne prend pas en compte les notions de multitâche, d'interruption ou d'asynchronisme, pourtant présentes dans les systèmes. Le logiciel VCC utilise un langage de description graphique pour effectuer une modélisation sous forme de diagramme de communications. Les parties logicielles peuvent utiliser des langages de description plus éloignés de l'implémentation comme SDL. D'un autre côté l'architecture cible est décrite de façon relativement précise en utilisant une représentation explicite des objets architecturaux qui seront générés. Ce mode de représentation provient sans doute de l'attachement

des concepteurs aux parties matérielles d'un système. L'architecture —comme le comportement— peut pourtant être modélisée en utilisant des langages orientés objets comme UML. La synthèse de système manipule des concepts et des systèmes de calculs (ou programmation) variés. Cependant si l'on veut pouvoir les assembler de manière cohérente et optimale, il convient d'unifier également les langages de description.

Une représentation et des modèles adaptés sont nécessaires pour pouvoir mettre en place des outils intelligents proposant une aide au concepteur. Toutefois, une automatisation complète des transformations de programme, du partitionnement et de l'assignation semble peu réaliste. En effet, la conception conjointe de systèmes souffre en partie des mêmes critiques que celles qui avaient été formulées lors de l'apparition de la synthèse de haut niveau. L'espace de recherche est trop important et une exploration de toutes les solutions possibles ne pourra sans doute jamais exister. Faute de pouvoir mettre en place une automatisation complète des transformations, il faut donc s'appuyer sur des méthodologies intéractives simples d'utilisation, mais néanmoins puissantes. Des outils d'aide peuvent proposer efficacement des modifications ponctuelles ou locales de comportement, des réorganisations globales du code ainsi que garantir la justesse des transformations demandées par le concepteur. De tels outils doivent permettre de transformer et optimiser de manière interactive un système sans qu'il soit nécessaire de refaire des étapes de validation ou de cosimulation très coûteuses. Il faut transformer les outils de cAO en outils de Cao afin que les concepteurs puissent gérer la complexité des futurs systèmes.

Cette complexité provient en grande partie de la richesse des connaissances qui doivent être combinées de manière efficace. La conception de systèmes se situe aux frontières de plusieurs domaines qui jusqu'ici n'étaient que rarement associés sur de mêmes projets de recherche. Les outils disponibles dans les domaines de la compilation logicielle et de la synthèse de matériel doivent pouvoir être rapprochés et confrontés. En effet, les systèmes embarqués utilisent de plus en plus de composants logiciels qui jusqu'alors étaient perçus comme des entités disjointes. C'est par exemple le cas pour les noyaux de systèmes d'exploitation temps-réel (*Real Time Operating System – RTOS*) se retrouvant maintenant intégrés directement dans un SoC. Cette intégration ne peut se faire efficacement que si le RTOS est modifiable pour être adapté et dédié au système. Une telle mise en place nécessite une bonne connaissance à la fois des ressources architecturales et du fonctionnement détaillé des systèmes d'exploitation. Ce constat est valable pour de nombreux domaines et une interaction forte doit donc être mise en place entre des personnes venant de domaines variés comme les systèmes d'exploitation, la compilation logicielle, le parallélisme, la synthèse de matériel, le traitement du signal, les réseaux ou de nombreux autres domaines présents dans la conception de systèmes embarqués. Des thèmes de recherche tels que la réutilisation de composants, la validation temporelle ou comportementale, la cosimulation de systèmes distribués et la vérification doivent être confrontés afin de mettre en commun les techniques et les outils utiles dans les différents domaines.

Quelle doit être l'interface présentée au concepteur ? Quelle représentation utiliser ? Comment mettre en place des transformations dans un processus semi-automatique ? Ces questions devront trouver une réponse si l'on veut pouvoir créer des outils efficaces permettant de concevoir des systèmes complexes de grande taille.

# Bibliographie

[ACFS94] Alpern (Bowen), Carter (Larry), Feig (Ephraim) et Selker (Ted). – The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 1994, vol. 12, p. 72–109.

[AIY95] Ancourt (C.), Irigoin (F.) et Yang (Y.). – Minimal data dependency abstraction for loop transformations. *International. Journal of Parallel Programming*, 1995, vol. 23, n4, p. 359–362.

[ARK99] Aditya (Shail), Rau (B. Ramakrishna) et Kathail (Vinod). – Automatic architectural synthesis of VLIW and EPIC processors. *In : IEEE International Symposium on System Synthesis*. – 1999. p. 107–113.

[ASU91] Aho (Alfred), Sethi (Ravi) et Ullman (Jeffrey). – *Compilateurs : principes, techniques et outils*. – Paris : InterEditions, 1991. 875 p. ISBN 2–7296–0295–X.

[BB96] Burd (Thomas D.) et Brodersen (Robert W.). – Processor design for portable systems. *Journal of VLSI Signal Processing*, août 1996, vol. 13, n2, p. 203–222.

[BBPM99] Benini (Luca), Bogliolo (Alessandro), Paleologo (Giuseppe A.) et Micheli (Giovanni De). – Policy Optimization for Dynamic Power Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, juin 1999, vol. 18, n6, p. 813–833.

[BCBM98] Brockmeyer (Erik), Catthoor (Francky), Bormans (Jan) et Man (Hugo De). – Code Transformations for Reduced Data Transfer and Storage in Low Power Realisation of MPEG-4 full-pel Motion Estimation. *In : IEEE International Conference on Image Processing, ICIP*. – Chicago, octobre 1998. p. 985–989.

[BDSV97] Boulet (Pierre), Darte (Alain), Silber (Georges-André) et Vivien (Frédéric). – *Loop Parallélization algorithms : from parallelism extraction to code generation*. – Laboratoire de l'informatique du parallélisme, École Normale Supérieure de Lyon, juin 1997. Rapport de recherche n97-17.

[Ber] Berkelaar (Michel). – Lp_solve Mixed Integer Linear Programming solver 3.2. – On line, archive FTP disponible sur `ftp://ftp.es.ele.tue.nl/pub/lp_solve/`. consulté le 23 septembre 2001.

[BGS94] Bacon (David F.), Graham (Susan L.) et Sharp (Oliver J.). – Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, décembre 1994, vol. 26, n4, p. 345–420.

[BJN99] Berkel (C. H. (Kees) van), Josephs (Mark B.) et Nowick (Steven M.). – Scanning the technology : Applications of asynchronous circuits. *Proceedings of the IEEE*, 1999, vol. 87, n2, p. 223–233.

[BM00] Benini (Luca) et Micheli (Giovanni De). – System-Level Power Optimization Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, avril 2000, vol. 5, n2, p. 115–192.

[BMM+98] Benini (Luca), Micheli (Giovanni De), Macii (Enrico), Sciuto (Donatela) et Silvano (C.). – Address bus encoding techniques for system-level power optimisation. *In : Design, Automation and Test in Europe*. – Paris, France, février 1998. p. 861–866.

[BQR+90] Benaini (A.), Quinton (P.), Robert (Y.), Saouter (Y.) et Tourancheau (B.). – Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 1990, vol. 15, p. 135–158.

[Cad] Cadence Design Systems. – On-line, disponible sur `http://www.cadence.com/`. consulté le 23 septembre 2001.

[CAP99] Centoducatte (Paulo), Araujo (Guido) et Pannain (Ricardo). – Compressed Code Execution on DSP Architectures. *In : IEEE International Symposium on System Synthesis*. – 1999. p. 56–61.

[Car93] Carr (Steve). – *Memory-Hierarchy Management*. – Thèse de Doctorat, Rice University, février 1993. 93 p.

[CCH+99] Chang (Henry), Cooke (Larry), Hunt (Merril), Martin (Grant), McNelly (Andrew) et Todd (Lee). – *Surviving the SOC Revolution : A Guide to Platform-Based Design*. – Boston : Kluwer Academic Publisher, 1999. 235 p. ISBN 0–7923–8679–5.

[CK94] Carr (Steven) et Kennedy (Ken). – Scalar Replacement in the Presence of Conditional Control Flow. *Software Practice & Experience*, janvier 1994, vol. 24, n1, p. 51–.

[CM95] Coleman (Stephanie) et McKinley (Kathryn S.). – Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices*, 1995, vol. 30, n6, p. 279–290.

[CMST00] Conte (Thomas), Menezes (Kishore N.), Sathaye (Sumedh W.) et Toburen (Mark C.). – System-Level Power Consumption Modeling and Tradeoff Analysis Techniques for Superscalar Processor Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, avril 2000, vol. 8, n2, p. 129–137.

[CWG+98] Catthoor (Francky), Wuytack (Sven), Greef (Eddy De), Balasa (Florin), Nachtergaele (Lode) et Vandecapelle (Arnout). – *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. – Boston : Kluwer Academic Publisher, 1998. 344 p. ISBN 0–7923–8288–9.

[Dan01] Danckaert (Koen). – *Loop Transformations for Data Transfer and Storage Reduction on Multiprocessor Systems*. – Thèse de Doctorat, Katholieke Universiteit Leuven and IMEC, mai 2001. 236 p.

[Dar99] Darte (Alain). – *De l'organisation des calculs dans les codes répétitifs*. – Habilitation à diriger des recherche, École Normale Supérieure de Lyon, décembre 1999. 179 p.

[Dar00] Darte (Alain). – On the Complexity of Loop Fusion. *Parallel Computing*, juillet 2000, vol. 26, n9, p. 1175–1193.

[DCM00] Danckaert (Koen), Catthoor (Francky) et Man (Hugo De). – A preprocessing step for global loop transformations for data transfer and storage optimization. *In : Compilers, Architectures and Synthesis for Embedded Systems*. – San Jose, CA, novembre 2000.

[De 98] De Greef (Eddy). – *Storage Size Reduction for Multimedia Application*. – Thèse de Doctorat, Katholieke Universiteit Leuven and IMEC, janvier 1998. 219 p.

[DH98] Darte (Alain) et Huard (Guillaume). – *Retiming et parallélisation automatique*. – École Normale Supérieure de Lyon, 1998. 35 p., Rapport de recherche nRR1998-33.

[dW90] de Werra (Dominique). – *Éléments de programmation linéaire avec applications aux graphes*. – Lausanne : Presses polytechniques romandes, 1990. 306 p. ISBN 2–88074–176–9.

[EIA98] EIAJ EDA Technology Roadmap Group. – Cyber-Giga-Chip in 2002. *In : EDA Technofair Handout*. – février 1998.

[FaHSCM94] Franssen (F.), ans H. Samson (L. Nachtergaele), Catthoor (F.) et Man (H. De). – Control flow optimization for fast system simulation and storage minimization. *In : Design, Automation and Test in Europe*. – Paris, France, février 1994. p. 20–24.

[Fea91] Feautrier (Paul). – Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 1991, vol. 20, n1, p. 23–53.

[Fea92a] Feautrier (Paul). – Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time. *International Journal of Parallel Programming*, octobre 1992, vol. 21, n5, p. 313–348.

[Fea92b] Feautrier (Paul). – Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multidimensional Time. *International Journal of Parallel Programming*, 1992, vol. 21, n6, p. 389–420.

[Fea96] Feautrier (Paul). – Automatic Parallelization in the polytope model. *In : The Data Parallel Programming Model*, éd. par Perrin (G. R.) et Darte (A.). – Berlin : Springer Verlag, 1996. p. 79–100.

[FEG00] Furber (S. B.), Edwards (D. A.) et Garside (J. D.). – AMULET3 : a 100 MIPS Asynchronous Embedded Processor. *In : International Conference on Computer Design : VLSI in Computers and Processors (ICCD2000)*. – 2000. p. 329–334.

[GM85] Gondran (Michel) et Minoux (Michel). – *Graphes et algorithmes*. – Paris : Eyrolles, 1985. 545 p. (Collection de la direction des études et recherches d'Électricité de France, volume 37).

[GOST92] Gao (Guang R.), Olsen (Russ), Sarkar (Vivek) et Thekkath (Radhika). – Collective Loop Fusion for Array Contraction. *In : Workshop on Languages and Compilers for Parallel Computing*. – New Haven, Conn., 1992. p. 281–295.

[Ham99] Hamilton (Scott). – Taking Moore's Law Into the Next Century. *Computer*, janvier 1999, vol. 32, n1, p. 43–48.

[Hau95] Hauck (Scott). – Asynchronous Design Methodologies : An Overview. *Proceedings of the IEEE*, janvier 1995, vol. 83, n1, p. 69–93.

[HKQ+98] Hong (Inki), Kirovski (Darko), Qu (Gang), Potkonjak (Miodrag) et Srivastava (Mani). – Power optimization of variable voltage core-based systems. *In : Design Automation Conference*. – 1998. p. 176–181.

[HMK+99] Hemani (A.), Meincke (T.), Kumar (S.), Olsson (T.), Nilsson (P.), Öberg (J.), Ellervee (P.) et Lundqvist (D.). – Lowering power consumption in clock by using globally asynchronous locally synchronous design style. *In : Design Automation Conference*. – juin 1999. p. 873–878.

[HPK97] Hong, Potkonjak et Karri. – Power optimizations using divide-and-conquer techniques for minimization of the number of operations. *In : IEEE/ACM International Conference on Computer Aided Design*. – 1997. p. 176–181.

[Hua01] Huard (Guillaume). – *Algorithmique du décalage d'instructions.* – Thèse de Doctorat, École Normale Supérieure de Lyon, novembre 2001. 160 p.

[IME] IMEC. – Acropolis. – On-line, disponible sur `http://www.imec.be/acropolis/`. consulté le 23 septembre 2001.

[JR97] Johnson (Mark C.) et Roy (Kaushik). – Datapath Scheduling with Multiple Supply Voltages and Level Converters. *ACM Transactions on Design Automation of Electronic Systems*, 1997, vol. 2, n3, p. 227–248.

[KBN98] Ko (U.), Balsara (P. T.) et Nanda (A. K.). – Energy Optimization of multilevel cache architectures for RISC and CISC processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1998, vol. 6, n2, p. 299–308.

[KCA01] Kjeldsberg (P. G.), Catthoor (F.) et Aas (E. J.). – Detection of partially simultaneously alive signals in storage requirement estimation for date-intensive applications. *In : 38th ACM/IEEE Design Automation Conference.* – Las Vegas, NV, juin 2001. p. 365–370.

[KLPMS99] Kirovski (Darko), Lee (Chunho), Potkonjak (Miodrag) et Mangione-Smith (William H.). – Application-Driven Synthesis of Memory-Intensive Systems-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, septembre 1999, vol. 18, n9, p. 1316–1326.

[KM94] Kennedy (Ken) et McKinley (Kathryn S.). – *Typed Loop Fusion with Applications to Parallel and Sequential Code Generation.* – Rice University, Center for Research on Parallel Computation, 1994. Technical Report nCRPC-TR94646.

[KNDK96] Kolson (David), Nicolau (Alexandru), Dutt (Nikil) et Kennedy (Ken). – Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 1996, vol. 1, n2, p. 251–279.

[KP93] Kelly (W.) et Pugh (W.). – *A framework for unifying reordering transformations.* – College Park MD, USA, Dept. of CS, Univ. of Maryland, avril 1993. Rapport technique nCS-TR-3193.

[KS98] Kalavade (Asaweree) et Subrahmanyam (P. A.). – Hardware/Software Partitioning for Multifunction Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, septembre 1998, vol. 17, n9, p. 819–837.

[Kul01] Kulkarni (Chidamber). – *Cache Optimization for Multimedia Applications.* – Thèse de Doctorat, Katholieke Universiteit Leuven and IMEC, février 2001. 215 p.

[Len93] Lengauer (C.). – Loop parallelization in the polytope model. *In : Proc. 4th Intl. Conf. on Concurrency Theory (CONCUR'93).* – Hildesheim, Germany, août 1993. p. 398–416.

[LMW99] Li (Yau-Tsun Steven), Malik (Sharad) et Wolfe (Andrew). – Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems*, juillet 1999, vol. 4, n2, p. 257–279.

[LRJD99] Lakshminarayana (Ganesh), Raghunathan (Anand), Jha (Hiraj K.) et Dey (Sujit). – Power Management in High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, mars 1999, vol. 7, n1, p. 7–15.

[LS91] Leiserson (Charles E.) et Saxe (James B.). – Retiming Synchronous Circuitry. *Algorithmica*, 1991, vol. 6, n1. – p. 5-35.

[LWH97] Lin (Yann Rue), Wu (Allen C.-H.) et Hwang (Cheng Tsung). – Scheduling Techniques for Variable Voltage Low Power Designs. *ACM Transactions on Design Automation of Electronic Systems*, avril 1997, vol. 2, n2, p. 81–97.

[MA97] Manjikian (Naraig) et Abdelrahman (Tarek S.). – Fusion of Loops for Parallelism and Locality. *IEEE Transactions of Parallel and Distributed Systems*, février 1997, vol. 8, n2, p. 193–209.

[MCJM98] Miranda (Miguel), Catthoor (Francky), Janssen (Martin) et Man (Hugo De). – High-level address optimization and synthesis techniques for date-transfer intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, décembre 1998, vol. 6, n4, p. 677–686.

[MCT96] McKinley (Kathryn S.), Carr (Steve) et Tseng (Chau-Wen). – Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, juillet 1996, vol. 18, n4, p. 424–453.

[MDV92] Moonen (M.), Dooren (P. Van) et Vandewalle (J.). – An SVD updating algorithm for subspace tracking. *SIAM J. of Matrix Anal. Appl.*, 1992, vol. 13, n4, p. 1015–1038.

[MHK+98] Meincke (Thomas), Hemani (Ahmed), Kumar (Shashi), Ellervee (Peeter), Öberg (Jhonny), Liqvist (Dan), Tenhunen (Hannu) et Postula (Adam). – Evaluating benefits of Globally Asynchronous Locally Synchronous VLSI architecture. *In : Proceedings of the 16th Norchip.* – novembre 1998. p. 50–57.

[MK93] McKinley (Kathryn S.) et Kennedy (Ken). – Maximizing Loop Parallelism and Improving Data Locality via loop fusion and distribution. *In : The Sixth Annual Languages and Compiler for Parallelism Workshop.* – 1993. p. 301–320, (Lecture Notes in Computer Science 768).

[MPS98] Macii (Enrico), Pedram (Massoud) et Somenzi (Fabio). – High-Level Power Modeling, Estimation, and Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, novembre 1998, vol. 17, n11, p. 1061–1079.

[MS97] Megiddo (Nimrod) et Sarkar (Vivek). – Optimal weighted loop fusion for parallel programs. *In : 9th annual ACM symposium on parallel algorithms and architectures (SPAA'97).* – Newport, Rhode Island, 1997. p. 282–291.

[NCK+96] Nachtergaele (L.), Catthoor (F.), Kapoor (B.), Moolenaar (D.) et Janssens (S.). – Low power storage exploration for H.263 video decoder. *In : VLSI Signal Processing IX*, éd. par Burleson (W.), Konstantinides (K.) et Meng (T.). – New York : IEEE, 1996. p. 116–125.

[Pan98] Panda (Preeti Ranjan). – *Memory optimization and exploration for embedded systems.* – Irvine, Thèse de Doctorat, University of California, janvier 1998.

[Pat98] Patterson (David). – Vulnerable Intel. – The New-York Times, 9 juin 1998.

[PD99] Panda (Preeti Ranjan) et Dutt (Nikil). – Low-Power Memory Mapping Through Reducing Address Bus Activity. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, septembre 1999, vol. 7, n3, p. 309–320.

[PDN99a] Panda (Preeti Ranjan), Dutt (Nikil) et Nicolau (Alexandru). – Local Memory Exploration and Optimization in Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, janvier 1999, vol. 18, n1, p. 3–13.

[PDN99b] Panda (Preeti Ranjan), Dutt (Nikil) et Nicolau (Alexandru). – *Memory Issues in Embedded Systems-On-Chip.* – Boston : Kluwer Academic Publisher, 1999. 188 p. ISBN 0–7923–8362–1.

[Ped96] Pedram (Massoud). – Power Minimization in IC Design : Principles and Applications. *ACM Transactions on Design Automation of Electronic Systems*, 1996, vol. 1, n1, p. 3–56.

[PH94] Patterson (David) et Hennessy (John). – *Organisation et conception des ordinateurs : l'interface matériel/logiciel.* – Paris : Dunod, 1994. pagination multiple. ISBN 2–10–002150–8.

[PNDN97] Panda (Preeti Ranjan), Nakamura (Hiroshi), Dutt (Nikil D.) et Nicolau (Alexandru). – Improving cache performance through tiling and data alignment. *In : Workshop on Parallel Algorithms for Irregularly Structured Problems.* – 1997. p. 167–185.

[PNDN99] Panda (Preeti Ranjan), Nakamura (Hiroshi), Dutt (Nikil D.) et Nicolau (Alexandru). – Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 1999, vol. 48, n2, p. 142–149.

[RB95] Rozenblit (Jerzy) et Buchenrieder (Klaus) (édité par). – *Codesign : computer-aided software/hardware engineering.* – Piscataway : IEEE, 1995. 452 p.

[RK98] Roth (Gerald) et Kennedy (Ken). – Loop fusion in high performance fortran. *In : International Conference on Supercomputing (ICS'98).* – Melbourne, Australia, 1998. p. 125–132.

[SC99] Schiue (W.) et Chakrabarti (C.). – Memory exploration for low power, embedded systems. *In : Proceedings of the Conference on Design Automation.* – 1999. p. 140–145.

[Sch86] Schrijver (Alexander). – *Theory of Linear and Integer Programming.* – New York : John Wiley and Sons, 1986. 471 p.

[SD98] Su (C.-L.), et Despain (A. M.). – Cache design trade-offs for power and performance optimization : a case study. *In : International Symposium on Low Power Design.* – avril 1998. p. 63–68.

[SG91] Sarkar (Vivek) et Gao (Guang R.). – Optimization of Array Accesses by Collective Loop Transformations. *In : International Conference on Superconputing.* – Cologne, Allemagne, 1991. p. 194–205.

[TMW94] Tiwari (V.), Malik (S.) et Wolfe (A.). – Power analysis of embedded software : a first step towards software power minimization. *In : Design Automation Conference.* – novembre 1994. p. 384–390.

[TSR+98] Tiwari (Vivek), Singh (Deo), Rajgopal (Suresh), Mehta (Gaurav), Patel (Rakesh) et Baez (Franklin). – Reducing power in high-performance microprocessors. *In : Design Automation Conference.* – 1998. p. 732–737.

[WCR+99] Wei (Liqiong), Chen (Zhanping), Roy (Kaushik), Johnson (Mark C.), Ye (Yibin) et De (Vivek K.). – Design and Optimization of Dual-Threshold Circuits for Low-Voltage Low-Power Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, mars 1999, vol. 7, n1, p. 16–24.

[Weh95] Wehn (Norbert). – High level synthesis : a critical assessment. *In : Logic and Architecture Synthesis – State-of-the-art and novel approaches*, éd. par Saucier (Gabrièle) et Mignotte (Anne). – London : Chapman & Hall, 1995. p. 289–299. ISBN 0–412–72690–4.

[Wol96] Wolfe (Michael). – *High Performance Compilers for Parallel Computing.* – Redwood-city : Addison-Wesley Publishing Company, 1996. 570 p.

[YCLV+99] Ykman-Couvreur (Ch.), Lambrecht (J.), Verkest (D.), Catthoor (F.) et Man (H. De). – Exploration and Synthesis of Dynamic Data Sets in Telecom Network Applications. *In : IEEE International Symposium on System Synthesis*. – 1999. p. 85–91.

# Publications personnelles

**Rapports de recherche :**

[1] Fraboulet (Antoine), Godary (Karen) et Mignotte (Anne). – *Loop Fusion for Memory Optimization.* – Rapport technique L3i-04-01, INSA-Lyon, France, L3i, 2001. 9 p.

[2] Bonnevay (Stéphane), Feschet (Fabien) et Fraboulet (Antoine). – *Regularity of digital lines.* – Rapport technique, Université Lyon 1, France, LASS, 2001. 8 p.

[3] Fraboulet (Antoine), Huard (Guillaume) et Mignotte (Anne). – *Loop alignment for memory access optimization.* – Rapport technique RR1999-26, ENS-Lyon, France, LIP, 1999. 13 p.

[4] Cifuentes (Cristina) et Fraboulet (Antoine). – *Interprocedural Data Flow Recovery of High-level Language Code from Assembly.* – Rapport technique, Department of Computer Science and Electrical Engineering, Université du Queensland, Australie, décembre 1997. 16 p.

**Conférence nationale :**

[1] Fraboulet (Antoine), Huard (Guillaume) et Mignotte (Anne). – Optimisation de la consommation et de la place mémoire par transformations de boucles. *In : Colloque CAO de circuits intégrés et systèmes.*– Aix-en-Provence, mai 1999. p. 181–184.

**Conférences internationales :**

[1] Fraboulet (Antoine), Godary (Karen) et Mignotte (Anne). – Loop Fusion for Memory Optimization. *Fourteenth International Symposium on System Synthesis Proceedings (ISSS'01)*, IEEE Computer Society Press. – Montréal, Canada, octobre 2001. p. 95–100.

[2] Fraboulet (Antoine) et Mignotte (Anne) – Source Code Loop Transformations for Memory Hierarchy Optimizations. *International Conference on Parallel Architectures and Compilation Techniques (PACT'01) – Workshop on Memory Access Decoupled Architectures (MEDEA)* – Barcelone, Espagne, septembre 2001. 6 p.

[3] Fraboulet (Antoine), Meunier (Laurence) et Mignotte (Anne). – Memory optimization of data flow applications at the codesign level. *In : Cadence Technical Conferences (CTC'01).* – mars 2001. 6 p.

[4] Fraboulet (Antoine), Meunier (Laurence) et Mignotte (Anne). – Memory optimization of data flow applications at the codesign level. *In : Sophia Antipolis Forum on MicroElectronics (SAME 2000).* – Sophia Antipolis, France, octobre 2000. p. 16–21.

[5] Fraboulet (Antoine), Huard (Guillaume) et Mignotte (Anne). – Loop alignment for memory access optimization. *Twelfth International Symposium on System Synthesis Proceedings (ISSS'99)*, IEEE Computer Society Press. – San Jose, USA, novembre 1999. p. 71–77.

[6] Cifuentes (Cristina), Fraboulet (Antoine) et Simon (Doug). – Assembly to High-Level Language Translation. *In : Proceedings of the International Conference on Software Maintenance (ICSM'98)*, IEEE Computer Society Press. – Washington DC, USA, novembre 1998. p. 228–237.

[7] Cifuentes (Cristina) et Fraboulet (Antoine). – Intraprocedural Slincing of Binary Executables. *In : Proceedings of the International Conference on Software Maintenance (ICSM'97)*, IEEE Computer Society Press. – Bari, Italie, octobre 1997. p. 188–195.

**Articles soumis :**

[1] Feschet (Fabien) et Fraboulet (Antoine). – Regularity of digital lines. – *Computer and Graphics*. 2001. Soumis à publication.

# FOLIO ADMINISTRATIF

## THÈSE SOUTENUE DEVANT L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON


Nom : Fraboulet
Prénom : Antoine



date de soutenance : 23 novembre 2001


Titre : Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués


Nature : Doctorat
Numéro d'ordre : 01 ISAL 0054
Formation doctorale : Informatique et information pour la société (IIS)



Résumé : L'évolution des techniques et des outils de compilation logicielle et de synthèse automatique de matériels permet maintenant de concevoir de manière conjointe (*Codesign*) des systèmes électroniques intégrés sur une seule puce de silicium, appelés « *System on Chip* ». Ces systèmes dans leurs versions embarquées doivent répondre à des contraintes spécifiques de place, de vitesse et de consommation. De plus, les capacités sans cesse croissantes de ces systèmes permettent aujourd'hui de développer des applications complexes comme les applications multimédia. Les applications multimédia travaillent, entre autres, sur des images et des signaux de grande taille ; elles génèrent de gros besoins en place mémoire et des transferts de données volumineux, traités par des boucles imbriquées. Il faut donc se concentrer sur l'optimisation de la mémoire lors de la conception de telles applications dans le monde de l'embarqué. Deux moyens d'action sont généralement mis en œuvre : le choix des architectures (hiérarchies mémoire et mémoires caches) et l'adéquation du code décrivant l'application avec l'architecture générée. Nous développerons ce second axe d'optimisation de la mémoire et comment transformer automatiquement le code de l'application, en particulier les boucles, pour minimiser les transferts de données (grands consommateurs d'énergie) et la place mémoire (grande utilisatrice de surface et d'énergie).



Mots clés : synthèse système, conception conjointe, codesign, système embarqué, consommation, multimédia, mémoire, transformation boucle



Laboratoires de recherche : Laboratoire de l'Informatique du Parallélisme (LIP) de l'ENS de Lyon et Laboratoire d'Ingénierie de l'Informatique Industrielle (L3I) de l'INSA de Lyon


Directeur de thèse : Pr Anne Mignotte

Président du jury :

Pr Yves Robert, ENS de Lyon

Composition du jury :

- Pr Michel Auguin, université de Nice, rapporteur
- Pr Francky Catthoor, université catholique de Louvain (Belgique), examinateur
- Pr Anne Mignotte, INSA de Lyon, directeur
- Dr Jacques-Olivier Piednoir, société Cadence, examinateur
- CR-INRIA HDR Tanguy Risset, IRISA de Rennes, rapporteur
- Pr Yves Robert, ENS de Lyon, président