Implantation optimisée sur circuit dédié d'algorithmes spécifiés sous la forme d'un Graphe Factorisé de Dépendances de Données : application aux traitements d'images

Linda KAOUANE[1], Mohamed AKIL[1], Thierry GRANDPIERRE[1], Yves SOREL[2]

[1]Groupe ESIEE–Laboratoire A2SI,
BP 99 - 93162 Noisy-le-Grand, France

[2]INRIA Rocquencourt–OSTRE,
BP 105 - 78153 Le Chesnay Cedex, France

{kaouanel, akilm, grandpit}@esiee.fr
yves.sorel@inria.fr

Résumé – On présente un flot de prototypage rapide pour l'implantation optimisée d'applications temps réel sur des circuits dédiés. Ce flot est basée sur un modèle unifié de graphes factorisés de dépendances de données autant pour spécifier l'algorithme que pour en déduire les implantaions possibles en termes de transformations de graphes.

Abstract – We present a fast prototyping flow for the optimized implementation of real-time applications onto specific circuits. This flow is based on an unified model of factorized data dependence graphs as well to specify the application algorithm, as to deduce the possible implementations in terms of graphs transformations.

1 Introduction

Les applications temps réel en traitement du signal et des images et de contrôle-commande mettent en oeuvres des algorithmes sophistiqués de plus en plus complexes. Cette complexité croissante augmente d'une part le coût et les délais du processus d'implantation et d'autre part la difficulté d'exploration de l'espace des implantations possibles en vue de trouver une implantation optimisée (celle qui respecte la contrainte temporelle et minimise les ressources matérielles utilisées). Il est ainsi nécessaire de définir des méthodes et outils logiciels d'aide à l'implantation intégrant l'ensemble des étapes, depuis la spécification haut niveau de l'algorithme jusqu'à son implantation optimisée sur une architecture cible. Pour répondre à ce besoin, l'équipe OSTRE de l'INRIA a mis au point une méthodologie formelle d'Adéquation Algorithme Architecture (AAA) basée sur des modèles de graphes, autant pour spécifier l'algorithme (mise en évidence du parallélisme potentiel) et l'architecture multi-composants (mise en évidence du parallélisme disponible), que pour déduire les implantations possibles en termes de transformations de graphes. Le résultat de ces transformations est un exécutif supportant l'exécution en temps réel de l'agorithme de l'application sur l'architecture cible [1]. Toutefois, ce passage systématique et automatisé d'un algorithme à son implantation optimisée est actuellement limité aux architectures cibles de type multiprocesseurs. Or, étant donné les besoins en puissance de calculs de ce type d'applications temps réel et la variété des algorithmes à implanter (algorithmes réguliers, irréguliers), l'architecture cible est souvent composée de processeurs connectés à des circuits intégrés spécifiques (ASIC et/ou FPGA). Dans cet article nous allons présenter notre extension de la méthodologie AAA aux circuits intégrés spécifiques et les bases d'intégation de cette extension au logiciel SynDEx (logiciel CAO niveau système suportant AAA).

2 Spécification algorithmique

La spécification algorithmique est le point de départ du processus d'implantation matérielle sur circuit de l'application. Cette spécification est modélisée par un hypergarphe orienté appelé "graphe factorisé de dépendances des données" (GFDD).

2.1 Modèle de Graphe Factorisé de Dépendances de Données

Dans ce modèle de graphe, chaque sommet est une opération (plus ou moins complexe, ex. une addition ou un filtre) et chaque hyperarc est une dépendance de données entre une opération productrice, et éventuellement plusieurs (cas de la diffusion) opérations consommatrices. Le modèle de GFDD permet non seulement de mettre en évidence le parallélisme potentiel de l'algorithme décrit par les dépendances de données mais aussi de réduire la taille de la spécification en factorisant ses parties répétitives ce qui très attractif pour la spécification des applications de traitement de signal et des images qui présentent souvent de telles caractéristiques. Cette spécification sous forme factorisée des répétitions de motifs d'opérations identiques opérant sur des données différentes fait apparaître des sommets opérations spéciaux (sommets frontière de factorisation) permettant de délimiter et de mettre en évidence le motif de la factorisation (partie régulière d'opérations opérant sur des données

différentes) : sommet Fork 'F' (partition d'un tableau en autant d'éléments que de répétitions du motif), sommet Join 'J' (composition d'un tableau à partir des résultats de chaque répétition du motif), sommet Diffuse 'D'(diffusion d'une même donnée à toutes les répétitions du motif) et sommet Iterate 'I' (dépendance de donnée inter-itération du motif). Chacun spécifie l'une des différentes manières de factoriser les données et les opérations en traversant une frontière de factorisation [2].

Du point de vue algorithmique ce processus de factorisation permet de réduire seulement la taille de la spécification en mettant en évidence ses parties régulières tout en maintenant sa sémantique. Á titre d'exemple, les graphes de la figure Fig.1 spécifient tous les deux le même produit vectoriel de deux vecteurs $VetV'$ de dimension 3, celui qui figure sur Fig.1.a est un graphe de dépendance de données non factorisé, alors que celui qui figure sur Fig.1.b est le graphe factorisé de dépendance de données équivalent. Bien que visiblement les deux graphes sont différents (noeuds et arcs différents), ils ont tout de même la même sémantique : appliquer l'opération de multiplication mul autant de fois qu'il y a d'éléments à multiplier (3 fois). Alors que, du point de vue comportemental ou opératoire ce processus de factorisation permet de décrire en intention plusieurs implantations plus ou moins séquentielles ou parallèles (transformation de la répétition en itération par implantation séquentielle du graphe factorisé de dépendance de données), chacune avec des caractéristiques (surface, latence) différentes.

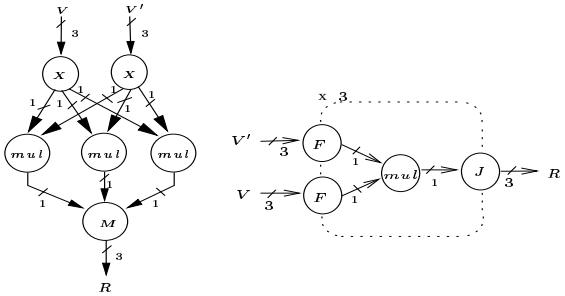

FIG. 1 – Exemple de graphe d'algorithme

2.2 Graphe de voisinage : relations entre frontières de factorisations

En fonction des dépendances de données entre frontières de factorisation, une frontière peut être consommatrice (située en aval) ou/et productrice (située en amont) par rapport à une autre frontière. Deux frontières sont donc voisines s'il existe entre elles au moins une relation de dépendance de données directe qui ne passe pas par l'intermédiaire d'une troisième frontière.

En se basant sur ces relations de voisinage entre les frontières de factorisation du graphe algorithmique, nous construisons un hypergraphe de voisinage où chaque sommet représente une frontière de factorisation et chaque hyperarc les dépendances de données entre frontières. L'orientation de l'arc indique les relations de production-consommation : l'arc part d'un producteur vers un consommateur. Comme la fréquence de consommations et productions de données diffère au niveau de chaque côté des zones délimitées par une frontière de factorisation FF, chaque frontière FF sépare donc deux zones l'une interne "rapide" et l'autre externe "lente". Ainsi, les sommets du graphe de voisinage repérensentant les frontières de factorisation au niveau du graphe de voisinage peuvent être sous-divisés schématiquement en quatre régions correpondants aux consommations/productions des deux côtés : lent et rapide (voir Fig.2).

- lent-amont : côté "lent" de FF, consommatrice ;
- rapide-aval : côté "rapide" de FF, productrice ;
- rapide-amont : côté "rapide" de FF, consommatrice ;
- lent-aval : côté "lent" de FF, productrice.

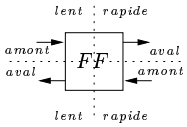

FIG. 2 – Sommet d'un graphe de voisinage représentant la frontière de factorisation FF

Ce graphe de relations de voisinage déduit automatiquement du graphe algorithmique GFDD, permet par la suite d'établir les relations de contrôle inter-frontières lors de l'implantation.

3 Synthèse automatique de circuits

Cette synthèse consiste à transformer automatiquement le graphe d'algorithme (GFDD) à implanter en un graphe matériel comprenant les chemins de données et de contrôle du circuit correspondant. L'automatisation de ce processus de synthèse réduit considérablement le cycle de développement du circuit et permet d'explorer différentes implantations matérielles pour obtenir un compromis surface/performance temporelle idéal.

Pour se faire, nous avons définis des régles simples permettant de synthétiser systématiquement les chemins de données et de contrôle du circuit correspondant.

3.1 Règles de synthèse du chemin de données

Ces règles se réduisent à une traduction directe du graphe d'algorithme, qui fait correspondre d'une part à chaque sommet opération le sommet opérateur matériel correspondant : cet opérateur est soit un composant d'une bibliothèque VHDL, combinatoire et/ou séquentiel pour un sommet opération de calcul. Soit un multiplexeur pour le sommet de factorisation Fork, démultiplexeur pour le sommet Join, registre pour le sommet Iterate I [3]. Et qui d'autre part, fait correspondre à chaque arc du graphe algorithmique, une connexion physique entre les opérateurs correspondants.

3.2 Règles de synthèse du chemin de contrôle

Le chemin de contrôle généré est obtenu en associant à chaque sommet frontière du graphe de voisinage sa propre unité de contrôle [4]. Chaque unité de contrôle se charge ainsi d'une part du bon déroulement de la factorisation interne à la frontière et d'autre part de la gestion des différents signaux de requête et d'acquittement associés aux relations de production/consommation de données avec les frontières voisines amont et aval.

L'unité de contrôle est donc composée (voir Fig.3) d'un compteur C à d états (d étant le facteur de factorisation) permettant de décompter les différents cycles de la factorisation et d'une

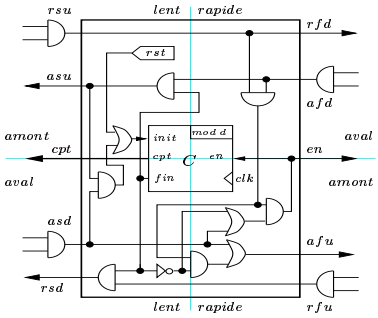

FIG. 3 – Unité de contrôle

logique supplémentaire afin de générer d'une part le protocole de communication entre frontières (signaux de requête et d'acquittement lent et rapide, en amont et en aval) et d'autre part la valeur du compteur (cnt) et le signal de validation en qui contrôlent les opérateurs de factorisation : $F,\ J\ et\ I$.

4 Optimisation de l'implantation

L'implantation séquentielle répétitive directe de la spécification initiale sous sa forme factorisée entraine souvent une utilisation minimale de surface en dépit d'un temps de calcul généralement élevé. Dès lors, si la latence de cette implantation directe ne respecte pas les contraintes temps réel de l'application, on procède à des défactorisations des frontières de factorisation du graphe initial (i.e déroulage de boucles) : défactoriser une frontière consiste donc à remplacer cette frontière par plusieurs frontières représentant le même motif de répétition, mais dont la somme des répétitions est égale à la répétition de la frontière initiale. Ces nouvelles frontières pouvant ainsi s'exécuter en parallèle, le temps de calcul en est diminué, mais en contrepartie la surface d'implantation est augmentée.

Parmi toutes les transformations possibles par défactorisation, on éliminera celles qui ne respectent pas les contraintes temps réel, et on choisira celle qui minimise les ressources nécessaires à l'implantation tout en respectant les contraintes temporelles. Pour que ce processus itératif ne soit pas trop fastidieux, ce choix doit être le plus automatisé possible et surtout la comparaison des caractéristiques (latence, quantités de ressources requises) des différentes implantations doit se faire sans que l'utilisateur n'ait à les réaliser. Pour se faire, nous avons développé un modèle prédictif de performance (temps, surface), appelé modèle de caractérisation, permettant de guider de manière efficace l'exploration de l'espace des solutions par l'heuristique d'optimisation de l'implantation. Comme nous nous intéressons au prototypage rapide, nous avons priviligié les "heuristiques gloutonnes" qui sont très rapides et donnent en moyenne des résultats de bonne qualité.

5 Exemple d'implantation de détecteurs de contours : filtre de Dériche

Nous allons illustrer notre extension AAA, sur un exemple concret d'algorithme récursif de détection de contour dû à Rachid Dériche, en particulier sa version optimisée par Garcia Lorca. Le choix de cet exemple très utilisé en traitement d'images pour sa qualité de détection, illustre parfaitement de nombreux problèmes liés aux implantations temps réel.

5.1 Présentation du filtre

L'implantation du filtre optimisé de Garcia-Lorca (Fig.4) comporte deux grandes étapes :

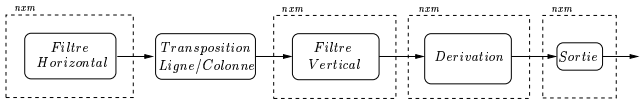

FIG. 4 – Algorithme du filtre de Dériche

– un Filtrage vertical et horizontal : pour un filtrage horizontal sur une ligne, on effectue tout d'abord deux passes d'un lisseur du premier ordre causal dont l'équation est : $y(n) = (1-\gamma)x(n) + \gamma y(n-1)$ ($x(n)$ étant le n^{ieme} pixel de la ligne en cours , $y(n)$ le n^{ieme} pixel filtré).
Puis deux passes d'un filtre anticausal dont l'équation est : $y(n) = (1-\gamma)x(n+1) + \gamma y(n+1)$. Le filtrage horizontal global (donc deux passes pour le filtre lisseur causal et deux pour le filtre lisseur anticausal) peut être vu comme 4 passes du même filtre causal à la condition d'intervertir les indices des éléments d'une ligne. C'est ce que réalise le bloc inversion sur la figure (Fig.5). On procède de même pour le filtre vertical sur les colonnes de l'image ;

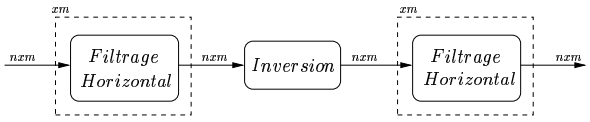

FIG. 5 – Filtre Horizontal

– une dérivation : procède par l'application des deux filtres de dérivation R_h et R_v dont les masques sont :

$$R_h \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} \quad et \quad R_v \begin{pmatrix} -1 & -1 \\ +1 & +1 \end{pmatrix}$$

5.2 Spécification algorithmique

Par la suite, nous présentons la spécification algorithmique sous forme de GFDD des principaux éléments constituants le filtre : le filtre lisseur causal d'ordre un (Fig.6) et le dérivateur (Fig.7).

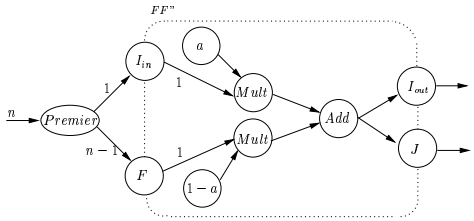

FIG. 6 – Filtre lisseur d'ordre un

Filtre lisseur d'ordre 1 Tout d'abord, un bloc **Premier** permet d'isoler le premier élement entrant $x(0)$ et de fournir les $n-1$ éléments restants dans un même vecteur. Au cycle i , l'élement $x(i)$ est dans l'*Iterate "I"* et l'élement $x(i+1)$ passe dans le fork "F". On applique alors les opérations du filtre lisseur sur ces deux élements *i.e* on fait : $(1-\gamma)x(i) + \gamma x(i+1)$

Le résultat de l'opération passe ensuite en même temps dans le *Join "J"* et est réutilisé dans l'*Iterate "I"* pour le cycle suivant (voir Fig.6).

Dérivation Le schéma général du filtre Fig.7 resssemble beaucoup dans sa conception au filtre lisseur d'ordre un : on se sert des *Iterate "I"* pour réutiliser l'élement que l'on vient de traiter.

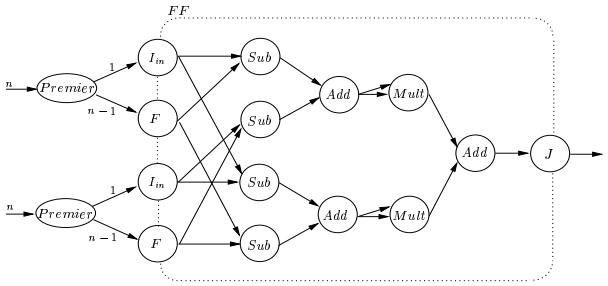

FIG. 7 – Le dérivateur

La prise de graphe sous SynDEx déduite du GFDD décrivant le filtre global est donnée dans la figure Fig.8. On présente en haut la spécification global du filtre suivie de la spécification hiérarchisée des différents élements constituants le filtre.

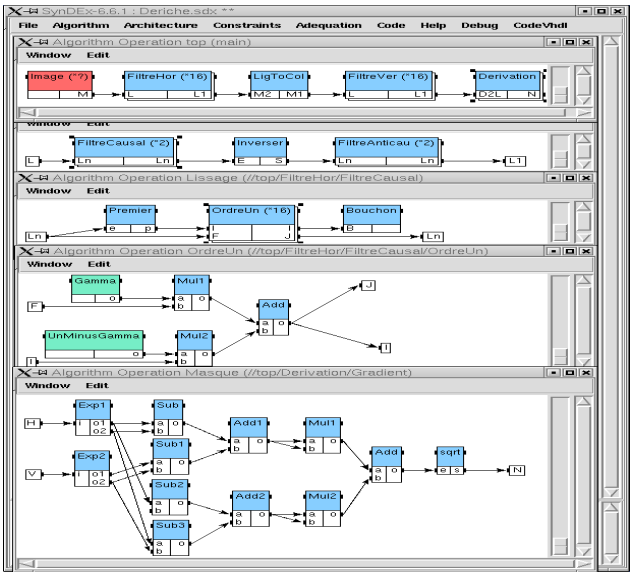

FIG. 8 – Ecran Syndex du filtre de déricheFIG. 9 – Résultats d'implantation sur FPGA

Contrainte (ns)	*Implementation*	*Surface* (CLB)	*Latence* (ns)
2000	Totalement Factorisée	1406	1961
1800	Défact.Part.FF par 2	1346	1453
1400	Défact.Part.FF par 3	1444	1235
1200	Défact.Part.FF par 4	1384	1162
1000	Défact.Tota.FF	2344	944

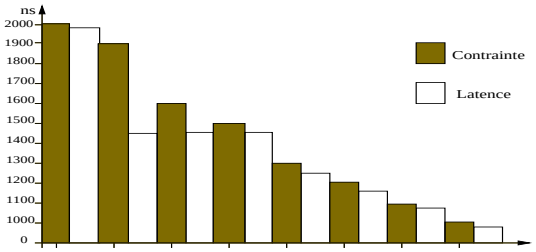

FIG. 10 – Optimisation sous contraintes de temps

5.3 Résultats d'implantation du filtre

Nous avons testé l'implantation du filtre sur des architectures FPGA *Virtex 300 BG 432* sous différentes contraintes de temps. Le Tab.9 présente les solutions obtenues par l'heuristique d'optimisation sous certaines contraintes. Par exemple pour une contrainte de latence de 1400 *ns* l'heuristique opte pour une implantation défactorisée de la frontière FF du dérivateur (fig.7) par un facteur de 3 (i.e remplace FF par trois frontières FF_1, FF_2, FF_3 pouvant s'exécuter en parallèle dont la somme des répétitions est égale au facteur de répétition de FF). Ces résultats présentés en termes, de surface pour les ressources matérielles (nombre de CLBs : Control Logic Blocs) et de latence (en *ns*) montre que les solutions les plus défactorisées permettent de réduire la latence mais augmentent en contrepartie la consommation en ressources. De ce fait, selon les contraintes imposées, les latences des solutions obtenues évoluent ainsi par palier Fig.10.

6 Conclusion

Nous avons présenté les principes permettant de déduire, par transformation de graphes, l'implantation optimisée sur circuit dédié d'algorithme spécifié sous forme de graphe factorisé de dépendances de donnése (GFDD). Ce flot de génération automatique d'implantation matérielle optimisée a été validé sur des exemples d'algorithmes de traitement d'images de bas niveau (filtre moyenneur, opérateurs de détection de contours : déricheFIG. sobel, ..) et a permis d'intégrer un générateur automatique de code VHDL structurel synthétisable dans SynDEx.

Références

[1] T. Grandpierre, C. Lavarenne, Y. Sorel. *Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors.* CODES'99 7th Intl. Workshop on Hardware/Software Co-Design, Rome, May 1999.

[2] C. Lavarenne, Y. Sorel. *Modèle unifié pour la conception conjointe logiciel-matériel. Traitement du Signal,* vol. 14, n. 6, 1997, p.569-577.

[3] A. F. Dias, M. Akil, C. Lavarenne, Y. Sorel. *Adéquation algorithme-architecture appliquée aux circuits reconfigurables.* Journées Adéquation Algorithme Architecture en traitement du signal et images, 4, Saclay, 1998.

[4] A. F. Dias, M. Akil, C. Lavarenne, Y. Sorel. *Vers la synthèse automatique de circuits à partir de graphes algorithmiques factorisés.* Journées Adéquation Algorithme Architecture en traitement du signal et images, 5, Rocquencourt, 2000.