ADÉQUATION ALGORITHME-ARCHITECTURE APPLIQUÉE AUX CIRCUITS RECONFIGURABLES

AILTON F. DIAS[1,2], MOHAMED AKIL[2], CHRISTOPHE LAVARENNE[3], YVES SOREL[3]

[1]CNEN/CDTN–Divisão de Computação e Informação,
CP 941 - 30123-970 Belo Horizonte, MG, Brésil
Tél : (55) 31 499 3100, E-mail : diasaf@urano.cdtn.br
[2]Groupe ESIEE–Laboratoire LPSI,
BP 99 - 93162 Noisy-le-Grand, France
Tél : (33) 1 45 92 65 00, E-mails : diasa@esiee.fr, akilm@esiee.fr
[3]INRIA Rocquencourt–Projet SOSSO,
BP 105 - 78153 Le Chesnay Cedex, France
Tél : (33) 1 39 63 55 11, E-mails : christophe.lavarenne@inria.fr, yves.sorel@inria.fr

RESUME. On présente la méthodologie "Adéquation Algorithme-Architecture" pour l'implantation optimisée d'applications temps réel sur des circuits reconfigurables. Cette méthodologie est basée sur un modèle de graphes factorisés qui utilise un ensemble d'opérations pour représenter un algorithme et un ensemble d'opérateurs pour représenter une architecture. Dans ce modèle, les optimisations sont exprimées en termes de transformations de factorisation. La méthodologie a été validée à travers quelques études de cas. On présente ici celle de l'algorithme du produit matrice-vecteur.

ABSTRACT. *We present the Algorithm Architecture "Adequation" methodology for the optimized implementation of real-time applications on reconfigurable circuits. This methodology is based on a factorized graphs model using a set of operations to compose an algorithm and a set of operators to compose an architecture. In this model, optimizations are expressed in terms of factorization transformations. The methodology was validated by some algorithms. In this paper, we study the matrix-vector product algorithm.*

1 Introduction

La complexité croissante des applications dans les domaines du contrôle-commande et du traitement du signal et des images et le coût élevé de leurs implantations, notamment dans un contexte temps réel, imposent le développement d'outils intégrant l'ensemble des étapes depuis la spécification haut niveau jusqu'à l'implantation. Ces applications nécessitent la mise en œuvre d'une implantation qui respecte les contraintes temps réel et minimise les ressources matérielles utilisées.

Le compromis entre les contraintes temps réel (latence ou temps de réponse, et cadence ou période d'échantillonnage) d'une application et la consommation des ressources matérielles nécessaires à son implantation peut être représenté par la Fig. 1. L'axe correspondant à la *cadence* (ou à la *latence*) a une borne supérieure représentant la contrainte temps réel qu'on ne peut dépasser. L'axe *ressources matérielles* peut représenter le nombre de composants, la surface en silicium, etc., en fonction du type d'architecture choisie pour l'implantation. Dans le cadre de cet article, on s'intéresse aux circuits reconfigurables tels les FPGA (*Field Programmable Gate Array*).

On propose une méthodologie qui part d'une description de l'algorithme de l'application (*spécification algorithmique*) sous la forme d'un graphe factorisé de dépendances entre opérations, pour arriver, par transformations de graphes, à plusieurs implantations matérielles.

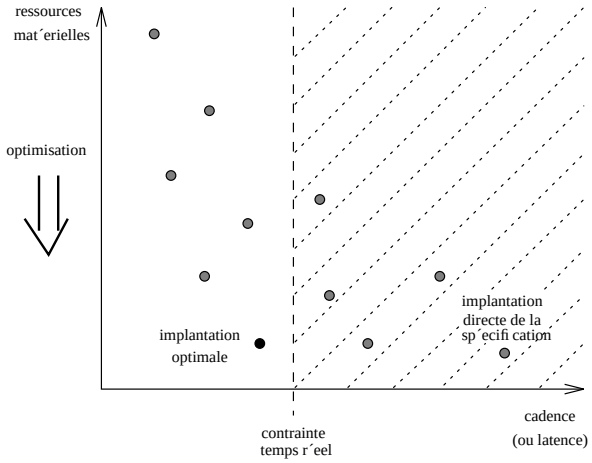

FIG. 1: Implantations d'une spécification algorithmique

Chacune de ces implantations est décrite sous la forme d'un graphe d'opérateurs interconnectés (*implantation matérielle*). La logique économique veut qu'on cherche à minimiser le coût d'implantation, donc à choisir l'implantation qui minimise les ressources matérielles. Mais les contraintes techniques, principalement temps réel, imposent une borne à cette minimisation. Ainsi, on est face à un problème d'*optimisation* : la recherche d'un minimum sous contraintes, comme le montre la Fig. 1. Comme le problème d'allocation de ressources est un problème NP-complet, cette optimisation n'est en pratique possible qu'à travers l'utilisation d'heuristiques. Ces heuristiques doivent fournir de bons résultats en un temps acceptable, éventuellement avec l'aide d'informations supplémentaires fournies par l'utilisateur.

Le langage VHDL (*Very high speed integrated circuits Hardware Description Language*), dans sa forme structurelle, synthétisable, a été choisi pour le codage de l'implantation matérielle.

La méthodologie a été validée à travers quelques études de cas. On a choisi de présenter comme cas d'étude l'algorithme du produit matrice-vecteur (PMV) qui met en évidence tous les cas intéressants de parallélisme de données et d'opérations.

À travers l'analyse du PMV, on a étudié un ensemble de transformations de graphes qui permettent de réduire soit les durées d'exécution, soit les ressources matérielles nécessaires pour sa réalisation, au détriment l'un de l'autre. On les appelle des *Transformations Spatio-Temporelles*.

2 Spécification algorithmique

La spécification algorithmique est le point de départ du processus d'implantation matérielle d'une application sur une architecture. Cette spécification consiste à modéliser l'application sous la forme d'un algorithme dont les opérations n'ont pour relation d'ordre d'exécution que leurs dépendances de données, établissant un ordre partiel qui met en évidence un parallélisme potentiel que ne mettrait pas en évidence un algorithme séquentiel. Pour représenter des algorithmes, on a choisi un modèle de graphe factorisé de dépendances de données (GFDD) entre opérations (*graphe algorithmique*) dont chaque sommet représente soit une opération de base (qui combine des données en entrée pour produire un(des) résultat(s) en sortie), soit une frontière de factorisation de motifs de sous-graphes répétitifs, et dont chaque arc représente une dépendance de données entre deux opérations. La modélisation basée sur la théorie des graphes permet de démontrer formellement des propriétés de la spécification, telles que par exemple l'absence de cycles de dépendance.

La factorisation consiste à remplacer, dans un graphe, un motif d'opérations répétif par un seul exemplaire du motif, délimité par des sommets spéciaux "frontières". La factorisation ne change pas les dépendances du graphe, elle permet simplement de réduire la taille de la spécification, tout en mettant en évidence ses parties répétitives.

Les applications qui nous intéressent interagissent avec leur environnement par une répétition infinie de la séquence acquisition-calculs-commande, qui correspond à un graphe de dépendances infiniment répétitif, dont la factorisation correspond à un graphe "flots de données".

L'ensemble des opérations d'un algorithme peut être spécifié sous différentes formes plus ou moins factorisées. La défactorisation, totale ou partielle, est une transformation triviale qui permet de produire automatiquement toutes les formes plus ou moins défactorisées d'une spécification.

2.1 Sommets de base

Les sommets de base décrivent les opérations combinatoires de l'algorithme. On en distingue cinq types :

- *DONNEE* : déclaration d'une donnée d'entrée, c'est une source du graphe,
- *RESULTAT* : représentation d'un résultat dans un graphe fini, c'est un puits du graphe,
- *CALCUL* : opération combinatoire arithmétique ou logique sans effet de bord,
- *IMPLODE* : regroupement ordonné de n données d'entrée de même type en un vecteur de sortie de dimension n,
- *EXPLODE* : décomposition d'un vecteur en ses éléments, c'est l'opération inverse de l'*IMPLODE*.

Les types représentent le codage binaire des données. Le type d'un vecteur s'exprime récursivement à partir du type de ses éléments qui peuvent être eux-même des vecteurs.

2.2 Sommets frontières de factorisation de graphes finis

Chacun de ces sommets spécifie une forme différente de factorisation des arcs coupés par la frontière du motif [LavSor 96] :

- *FORK* : factorisation des entrées d'un motif répétitif, énumération des éléments d'un vecteur, c'est le correspondant factorisé de l'*EXPLODE*,
- *JOIN* : factorisation des sorties d'un motif répétitif, c'est l'opération inverse du FORK et le correspondant factorisé de l'*IMPLODE*,
- *ITERATE* : factorisation des dépendances de données inter-motifs,
- *DIFFUSION* : factorisation des entrées d'un motif répétitif provenant toutes d'une même opération externe au motif.

2.3 Sommets frontières de factorisation de graphes infinis

On retrouve les mêmes genres de sommets que pour les graphes finis : *FORK*$^\infty$, *JOIN*$^\infty$, *ITERATE*$^\infty$ et *DIFFUSION*$^\infty$ sont équivalents respectivement aux entrées, sorties, retards et constantes des graphes flots de données. On verra que les versions finie et infinie ont des implantations matérielles différentes (sauf les sommets *DIFFUSION*).

3 Implantation matérielle

L'implantation consiste à faire correspondre à chaque sommet du GFDD d'une spécification un opérateur (composant d'une bibliothèque VHDL, combinatoire pour un sommet opération, multiplexeur et/ou registre pour un sommet frontière) et à chaque arc une connexion entre opérateurs.

3.1 Opérateurs frontières de factorisation de graphes infinis

Comme on s'intéresse aux applications interagissant de manière infiniment répétitive avec leur environnement, commençons par l'implantation des sommets de frontières de factorisation infinie. Ces sommets représentent les capteurs (*FORK*$^\infty$) et les actionneurs (*JOIN*$^\infty$) qui assurent l'interface avec l'environnement, ainsi que les retards (*ITERATE*$^\infty$) et/ou les constantes des graphes "flots de données" (*DIFFUSION*$^\infty$).

3.1.1 Opérateur *FORK*$^\infty$: capteur

L'opérateur *FORK*$^\infty$ représente un capteur qui convertit un signal physique en un signal numérique. On considère que ce capteur intègre tous les circuits nécessaires pour la conversion et l'enchantillonnage des signaux (le convertisseur analogique-numérique, CAN, et le registre, REG), comme le montre la Fig. 2.

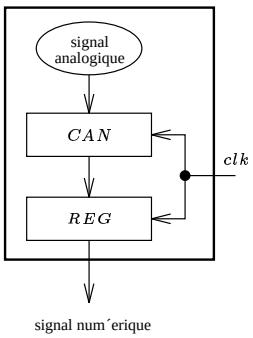

FIG. 2: Opérateur *FORK*$^\infty$

3.1.2 Opérateur *JOIN*$^\infty$: actionneur

L'opérateur *JOIN*$^\infty$ représente un actionneur qui convertit un signal numérique en un signal analogique. On considère que cet actionneur intègre tous les circuits nécessaires pour la mémorisation et la conversion des signaux (le registre, REG, et le convertisseur numérique-analogique, CNA), comme le montre la Fig. 3.

3.1.3 Opérateur *ITERATE*$^\infty$: retard

L'opérateur *ITERATE*$^\infty$ correspond à un registre REG, comme le montre la Fig. 4. REG maintient en sortie, pendant toute la durée du cycle d'horloge, la valeur qu'il avait en entrée à la fin du cycle d'horloge précédent. Pendant que le signal d'initialisation (rst) est présent, le registre est

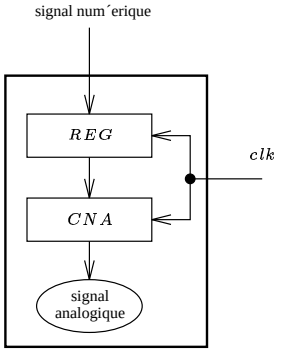

FIG. 3: Opérateur *JOIN*$^\infty$

forcé à la valeur $init$. On remarque que, par rapport à l'implantation de l'opérateur *ITERATE* fini (cf. 3.2.3), il n'y a pas la sortie fin, car dans le cas d'un graphe infini, il n'y a pas d'itération finale.

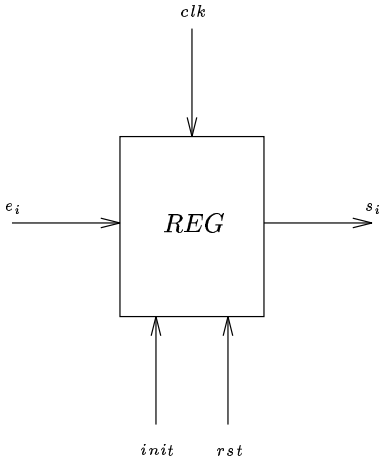

FIG. 4: Opérateur *ITERATE*$^\infty$

3.1.4 Opérateur *DIFFUSION*$^\infty$: constante

L'implantation de l'opérateur *DIFFUSION*$^\infty$ est identique à celle de l'opérateur *DIFFUSION* dans le cas fini (cf. 3.2.4). Il représente l'utilisation répétitive de constantes dans des graphes flots de données.

3.2 Opérateurs frontières de factorisation de graphes finis

Les graphes qui spécifient les applications qui nous intéressent peuvent, en plus de frontières de factorisation infinies, contenir d'autres frontières de factorisation finies.

3.2.1 Opérateur *FORK*

L'opérateur *FORK* est composé de (voir Fig. 5) :

– un compteur modulo d, identifié par C, reçoit en entrée les signaux d'horloge (clk) et de rémise à zéro

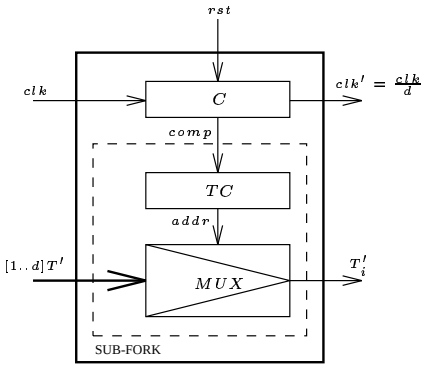

FIG. 5: Opérateur *FORK*

(rst). Il produit en sortie un signal d'horloge de sortie ($clk' = \frac{clk}{d}$) et l'état du compteur ($comp$) ;

- un transcodeur, identifié par TC, générant, à partir de la séquence de comptage ($comp$), une séquence d'adresses ($addr$) pour le multiplexeur (MUX) ;
- un multiplexeur, identifié par MUX, reçoit en entrée le signal T, vecteur composé des d signaux T'_i, et le signal $addr$. Il fournit en sortie le signal T'_i sélectionné.

3.2.2 Opérateur *JOIN*

L'opérateur *JOIN* est composé de (voir Fig. 6) :

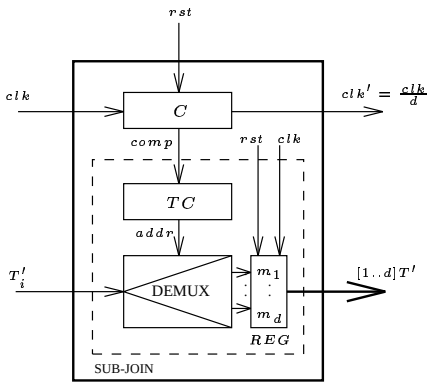

FIG. 6: Opérateur *JOIN*

- un compteur modulo d, identifié par C, cf. 3.2.1 ;
- un transcodeur, identifiée par TC, cf. 3.2.1 ;
- un banc de registres, identifié par REG, fournit en sortie le signal $[1..d]T'$ qui représente un vecteur à d éléments ;
- un démultiplexeur, identifié par $DEMUX$, à d variables d'adresse, réalise l'aiguillage d'un signal T'_i sur l'un des m registres, sélectionné par les signal $addr$.

3.2.3 Opérateur *ITERATE*

L'opérateur *ITERATE* est composé de (voir Fig. 7) :

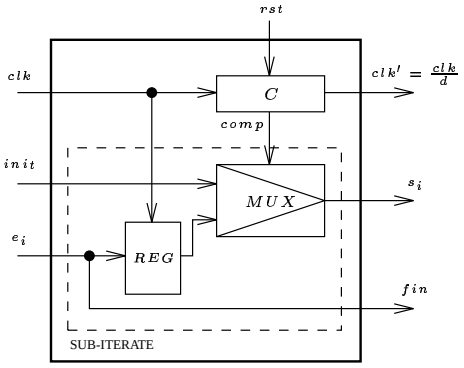

FIG. 7: Opérateur *ITERATE*

- un compteur modulo d, identifié par C. À l'état initial du compteur, il sélectionne la valeur initiale $init$. Dans l'état "final" ($d-1$), la valeur de sortie fin est celle désirée. Il reçoit en entrée les signaux d'horloge (clk) et de rémise à zéro (rst). Il produit en sortie le signal d'horloge de sortie ($clk' = \frac{clk}{d}$) et l'état du compteur ($comp$) ;
- un registre , identifié par REG, mémorise la valeur de l'itération précédente ;
- un multiplexeur, identifié par MUX, aiguille soit la valeur d'initialisation ($init$) si le compteur est dans son état "initial", sinon la sortie du registre REG.

3.2.4 DIFFUSION

Le sommet *DIFFUSION* ne sert qu'à marquer la frontière de factorisation. Il implante des connexions directes entre son entrée et sa sortie, comme le montre la Figure 8.

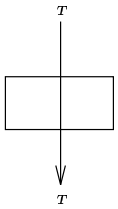

FIG. 8: Opérateur *DIFFUSION*

3.3 Opérateurs de base

Les opérateurs de base implantent les parties combinatoires de la spécification algorithmique. L'implantation d'un graphe algorithmique totalement défactorisé ne peut contenir que des opérateurs de base.

3.3.1 Opérateur *CONSTANTE*

L'opérateur *CONSTANTE*, identifié par K, correspond à l'implantation matérielle de l'opération *DONNEE*. Il est utilisé, par exemple, pour fournir la valeur initiale d'un opérateur *ITERATE*.

3.3.2 Opérateur *RESULTAT*

Les graphes infinis factorisés, spécifiant les applications qui nous intéressent, ne peuvent qu'avoir une infinité de résultats, factorisés par des opérateurs *JOIN*$^\infty$, donc ils ne peuvent avoir de sommets *RESULTAT*.

3.3.3 Opérateur *CALCUL*

L'ensemble d'opérateurs *CALCUL* peut, soit être disponible dans une bibliothèque d'opérateurs prédéfinis, soit être défini par l'utilisateur en fonction de ses besoins.

3.3.4 Opérateur *IMPLODE*

L'opérateur *IMPLODE* correspond à un regroupement ordonné de d bus unidirectionnels (le nombre de conducteurs reste le même), comme le montre la Fig. 9. Il implante des connexions directes entre des opérateurs en amont et l'opérateur en aval.

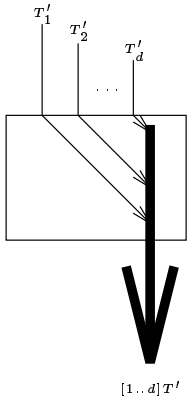

FIG. 9: Opérateur *IMPLODE*

3.3.5 Opérateur *EXPLODE*

L'opérateur *EXPLODE* correspond à une décomposition d'un bus unidirectionnel en d sous-bus (le nombre de conducteurs reste le même), comme le montre la Fig. 10. Il implante des connexions directes entre l'opérateur en amont et des opérateurs en aval.

4 Processus de défactorisation

Si l'implantation directe de la spécification sur l'architecture cible, en prenant en compte les contraintes technologiques, ne respecte pas les contraintes temps réel, il faut défactoriser le graphe correspondant à la spécification. La défactorisation est la transformation inverse à la factorisation et elle ne change pas la sémantique opératoire du graphe de dépendances de données.

La défactorisation du GFDD cherche à obtenir une implantation plus parallèle en améliorant les performances temporelles au prix de ressources matérielles suppémentaires. Une spécification factorisée peut avoir plusieurs défactorisations partielles : pour n frontières

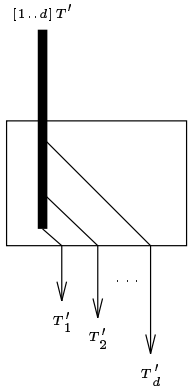

FIG. 10: Opérateur *EXPLODE*

de factorisations, il y a au minimum 2^n implantations défactorisées de la spécification. Chaque frontière peut n'être défactorisée que partiellement : une factorisation de r répétitions d'un motif peut se décomposer en f factorisations de r/f répétitions du motif [LavSor 96].

Le processus de défactorisation est illustré par le cas d'étude qui suit.

5 Cas d'étude : produit matrice-vecteur

Ci-dessous, on présente l'application des transformations spatio-temporelles à travers une étude de cas, à savoir l'algorithme du produit matrice-vecteur (PMV). On n'a présenté que les graphes algorithmiques correspondant aux différentes spécifications possibles du PMV.

5.1 Spécification

Le produit d'une matrice $A \in R^m \times R^n$ par un vecteur $B \in R^n$ donne un vecteur $C \in R^m$, et peut s'écrire, sous forme factorisée :

$$C = \left[c_i\right]_{i=1}^{m} = \left[\sum_{j=1}^{n} a_{ij} b_j\right]_{i=1}^{m} \tag{1}$$

que l'on peut défactoriser, ici pour $m = n = 3$,
soit partiellement par ligne :

$$C = \left[a_{i1} b_1 + a_{i2} b_2 + a_{i3} b_3\right]_{i=1}^{3} \tag{2}$$

soit partiellement par colonne :

$$C = \begin{bmatrix} \sum_{j=1}^{3} a_{1j} \cdot b_j \\ \sum_{j=1}^{3} a_{2j} \cdot b_j \\ \sum_{j=1}^{3} a_{3j} \cdot b_j \end{bmatrix} \tag{3}$$

soit totalement :

$$C = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + a_{13}b_3 \\ a_{21}b_1 + a_{22}b_2 + a_{23}b_3 \\ a_{31}b_1 + a_{32}b_2 + a_{33}b_3 \end{bmatrix} \quad (4)$$

La spécification factorisée présentée par la Fig. 11 a été conçue à partir de l'éq. 1, en considérant qu'on effectue d'abord les produits scalaires entre les lignes de la matrice A et le vecteur B et ensuite on collecte les résultats de ces produits scalaires sous la forme d'un vecteur qui correspond au PMV entre A et B. Ainsi, le graphe résultant possède deux frontières de factorisation (FF_1 et FF_2), correspondant aux deux étapes mentionnées ci-dessus.

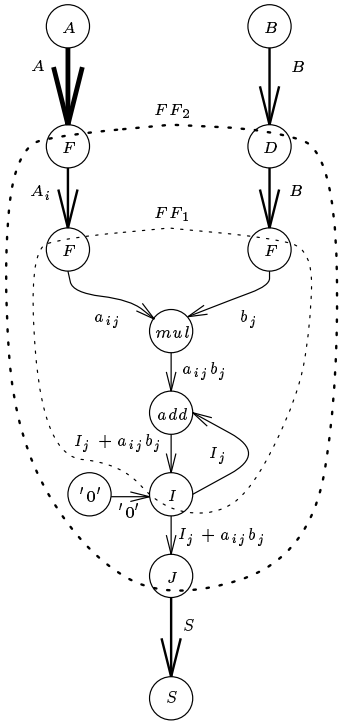

FIG. 11: Graphe factorisé (spécification)

5.2 Défactorisation

On présente ci-dessous les graphes algorithmiques correspondant aux différentes transformations du graphe de la spécification du PMV.

Dans le cas du graphe de la Fig. 11, on peut commencer à défactoriser soit par la frontière FF_1, soit par la frontière FF_2 avant d'arriver au graphe défactorisé par les frontières FF_1 et FF_2 (totalement défactorisé), montré par la Fig. 12. Ce graphe correspond à l'éq. 4. Les défactorisations intermédiaires de la spécification correspondent aux eq. 2 (défactorisation par FF_1, lignes) et 3 (défactorisation par FF_2, colonnes). Le graphe factorisé du départ possède deux frontières de factorisation. Ainsi, on obtient quatre (2^2) implantations différentes, sans compter les défactorisations partielles (qu'on n'a pas considérées à cause de la taille réduite du graphe défactorisé).

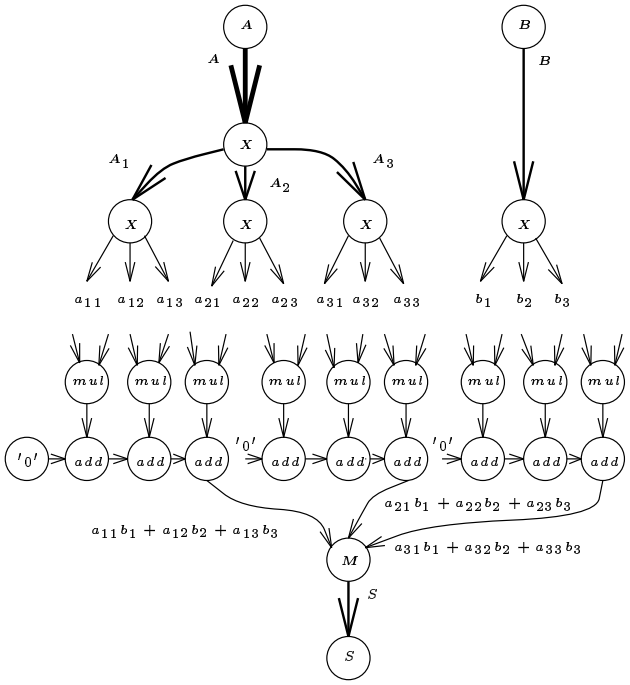

FIG. 12: Graphe défactorisé par FF_1 et FF_2

5.3 Implantation

L'implantation matérielle des graphes algorithmiques à consisté à écrire des programmes VHDL correspondant à ces graphes. Dans ces programmes, chaque sommet du graphe correspond à l'instanciation d'un composant VHDL appartenant à une bibliothèque d'opérateurs et chaque arc correspond à un signal VHDL.

Pour obtenir les résultats concernant l'implantation matérielle du PMV, on a utilisé les logiciels suivants :

- compilateur VHDL : *QuickVHDL qvcom v8.4-4.3e HP-UX A.09.05*, développé par *Mentor Graphics Co.* ;
- simulateur VHDL : *QuickVHDL qvsim v8.4-4.3e HP-UX A.09.05*, développé par *Mentor Graphics Co.* ;
- synthétiseur VHDL : *AutoLogic II UI v8.4-3.5*, développé par *Mentor Graphics Co.* ;
- générateur de netlist : *XMAKE Version 5.1.0*, développé par *Xilinx Inc.* ;
- estimateur de temps de propagation : *XDELAY Version 5.1.0*, développé par *Xilinx Inc.*

On a synthétisé et généré des netlists en utilisant des FPGAs *Xilinx*, modèle 4013PG223-4 dont les caractéristiques sont présentées ci-dessous :

Nb. CLB	*IO pins*	*CLB FFs*	*IOB FFs*	*Gén. func.*
576	192	1152	192	1728

Le Tab. 1 montre les résultats de l'implantation matérielle (synthèse) du PMV entre une matrice de 3×3

éléments codés sur 7 bits et un vecteur de 3 éléments codés sur 7 bits sur des circuits reconfigurables du type FPGA *Xilinx* 4013PG223-4. Ces résultats sont présentés en fonction de la surface occupée (nombre de CLB), du nombre de cycles nécessaires pour l'exécution de l'algorithme, de la fréquence maximale d'opération, de la cadence et de la latence des données. On remarque qu'il y a deux chiffres concernant le nombre de CLB : le premier représente les CLB occupés (avant l'optimisation réalisée par les outils de synthèse) et le deuxième représente les CLB compactés (après l'optimisation).

TAB. 1: Implantations matérielles du PMV

Implantation	*Surf. (CLB)*	*Nb. Cyc*	*Fréq. (MHz)*	*Cad. (ns)*	*Lat. (ns)*
Directe de la spécification	263/127	9	6,9	1297	1297
Défactorisée par FF_1	473/249	3	6,4	470	470
Défactorisée par FF_2	506/304	3	7,4	404	404
Défactorisée par FF_1 et FF_2	576/477	1	7,9	126	126

La Fig. 13 montre le rapport entre la surface et la cadence (ou la latence) à partir des données fournies par le Tab. 1. On considère que l'implantation directe de la spécification représente 100% de la surface et de la cadence. On représente les implantations défactorisées (défactorisation par FF_1, par FF_2 et par FF_1 et FF_2) en fonction du pourcentage de la surface occupée et de la cadence par rapport à l'implantation directe de la spécification. Supposons qu'il y a une contrainte temps réel correspondant à 50% de la cadence de l'implantation directe de la spécification. L'implantation optimale (Fig. 14), qui utilise le minimum de CLB tout en respectant la contrainte temps réel, est celle défactorisée par FF_1.

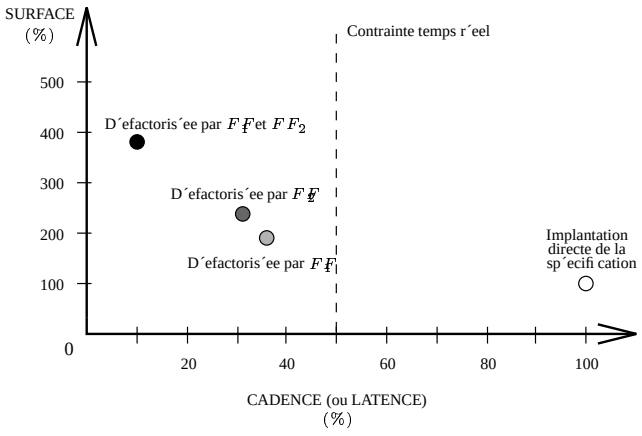

FIG. 13: Surface × cadence (ou latence)

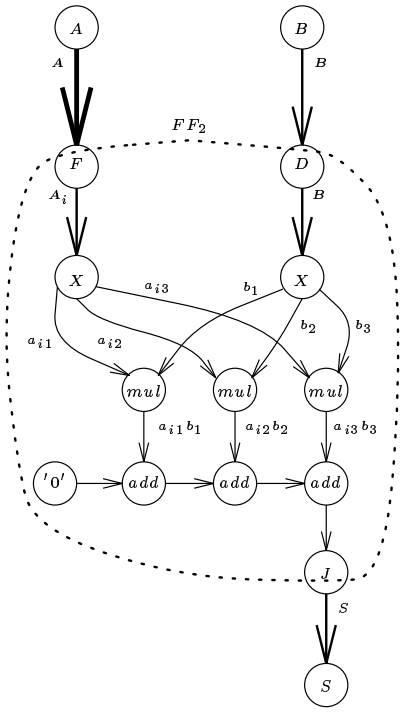

FIG. 14: Graphe défactorisé par FF_1

6 Conclusions

À chaque défactorisation d'une spécification correspond une implantation. Le plus souvent, une implantation plus défactorisée nécessite plus de ressources matérielles mais moins de latence et/ou cadence qu'une implantation moins défactorisée, car les données traitées en parallèle par des opérateurs différents dans le cas plus défactorisé, sont multiplexées pour être traitées par le même opérateur dans le cas moins défactorisé, mais cette règle n'est pas systématique.

Il peut arriver qu'une défactorisation ne diminue pas la cadence et/ou la latence des données, soit à cause d'une dépendance inter-motif (dont le chemin critique ne varie pas avec la factorisation), soit à cause d'une augmentation de "fan-out", surtout dans le cas où la répétition du motif doit être distribué sur plusieurs FPGA. Il peut arriver par contre qu'une défactorisation totale ne nécessite pas plus de ressources matérielles. Cela peut se produire lorsque la suppression des opérateurs frontières de factorisation représente un gain supérieur au coût de duplication de la combinatoire du motif défactorisé.

Ces cas particuliers sont à considérer dans la conception d'une heuristique d'optimisation qui, en gros, consiste à défactoriser jusqu'à ce que la contrainte temps réel soit respectée.

7 Perspectives

On a réalisé les transformations de graphes à la main, ainsi que l'estimation de leurs performances globales (surface et latence de chaque opérateur) à l'aide d'outils four-

nis par le constructeur des FPGA. On étudie actuellement un modèle ouvert d'estimation de performances globales, indépendant du constructeur, basée sur une composition des caractéristiques des opérateurs (surface, temps de réponse), obtenue par les outils de simulation appliquées à chaque opérateur isolé.

On étudie également une heuristique utilisant cette estimation de performances pour choisir et réaliser automatiquement les défactorisations qui respectent les contraintes temps réel en minimisant l'augmentation de surface de l'implantation.

Enfin, on envisage une génération automatique du code VHDL correspondant au graphe défactorisé choisi par l'heuristique.

8 Remerciements

Cette étude a été soutenue par la *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*–CAPES (dossier 0100-95/13) et de la *Comissão Nacional de Energia Nuclear*–CNEN.

Références

[1] Y. Sorel. *Massively Parallel Computing Systems with Real-Time Constraints : the "Algorithm Architecture Adequation" methodology.* Proc. of Massively Parallel Computing Systems, Ischia Italy, May 1994.

[2] C. Lavarenne, Y. Sorel. *Mod`ele unifi´e pour la conception conjointe logiciel-mat´eriel.* Journées Adéquation Algorithme Architecture en Traitement du Signal et Images, CNES, Toulouse France, January 1996.

[3] R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard. *VHDL du langage `a la mod´elisation.* Lausanne : Presses polytechniques et universitaires romandes, 1990.

[4] Xilinx. *The programmable logic data book.* San Jose : Xilinx, Inc., 1994.

[5] Mentor Graphics. *QuickVHDL user's and reference manual - software version 8.4_4.* Mentor Graphics Co., 1995.