Architecture matérielle et flot de programmation associé pour la conception de systèmes numériques tolérants aux fautes

Thomas Peyret

THÈSE / UNIVERSITÉ DE BRETAGNE SUD
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : Sciences et Technologies de l'Information et de la Communication
École Doctorale SICMA

présentée par

Thomas PEYRET

Préparée au CEA Saclay, DRT/LIST/DM2I/LCAE,
Laboratoire Capteurs et Architectures Électronique

Architecture matérielle et flot de programmation associé pour la conception de systèmes numériques tolérants aux fautes

Thèse soutenue le 2 décembre 2014
devant le jury composé de :

Lilian BOSSUET
HDR, Université Jean Monnet / *Rapporteur*

Gwenolé CORRE
Docteur, CEA Saclay / *Encadrant*

Philippe COUSSY
Professeur, Université de Bretagne Sud / *Directeur de thèse*

Youri HELEN
Docteur, DGA / *Invité*

Kevin MARTIN
Docteur, Université de Bretagne Sud / *Encadrant*

Michel PAINDAVOINE
Professeur, Université de Bourgogne / *Examinateur*

François PÉCHEUX
Professeur, Université Pierre et Marie Curie / *Rapporteur*

Olivier SENTIEYS
Professeur, IRISA/ENSSAT / *Président du jury*

Mathieu THEVENIN
Docteur, CEA Saclay / *Encadrant*

Laboratoire Capteurs et Architectures
Électroniques (LCAE)
DRT/LIST/DM2I
CEA Centre de Saclay
91191 Gif-sur-Yvette

École Doctorale Santé, Information,
Communication, Mathématiques,
Matière (ED SICMA 373)
UFR Sciences et Techniques
6 avenue Le Gorgeu, BP 809
29285 Brest Cedex

À Fernande

« *The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter - for the future. His duty is to lay the foundation for those who are to come, and point the way.* »

- Nikola Tesla

Remerciements

Je souhaite tout d'abord remercier les membres de mon jury de thèse, qui ont tous pris de leur temps pour lire et analyser mes travaux et en particulier Lilian Bossuet et François Pécheux qui m'ont fait l'honneur de rapporter mon manuscrit. Je remercie Michel Paindavoine et Youri Helen pour l'intérêt qu'ils ont porté à mes travaux et enfin Olivier Sentieys pour avoir accepté de présider mon jury de thèse et pour cette question qui est maintenant devenu une boutade : « Mais pourquoi le CGRA ? ».

Je tiens ensuite à remercier mon directeur de thèse, Philippe Coussy, pour m'avoir guidé tout au long de ces travaux. Un vrai esprit scientifique, rigoureux, précis, qui sait poser les bonnes questions au bon moment, qui peut, et m'a intimidé au début de ma thèse mais qui est aussi un homme exceptionnel sur le plan humain. Il m'a permis d'avancer lorsque je n'arrivais pas à prendre de décision, rassuré quant au déroulement de la thèse que cela soit sur le plan scientifique ou humain (« tu es fin de deuxième année, c'est absolument normal que tu doutes de tout ce que tu as fait... On est tous passé par là ! »), et qui malgré la distance, par sa disponibilité et son intérêt, a fait que ces travaux sont déroulés dans de bonnes conditions.

Bien entendu, ma thèse n'aurait pas été celle qu'elle est aujourd'hui sans Gwenolé Corre que je tiens premièrement à remercier pour avoir soumis ce sujet de thèse et m'avoir retenu comme candidat. Nos discussions ont toujours été productives car posées et argumentées. Gwenolé m'a toujours épaulé sans jamais rien m'imposer et je lui en suis extrêmement reconnaissant.

Enfin, je tiens à remercier mes deux autres encadrants, Kevin Martin et Mathieu Thevenin, pour leurs apports respectifs durant ces années de thèse. Ils m'ont tous deux rassuré quand j'en avais besoin, ont apporté leur expertise technique en réunion et permis d'avancer au travers de *brainstormings* toujours productifs (dont certains ont engendré des dépôts de brevet).

Tous les quatre, vous m'avez permis de m'épanouir durant cette thèse et j'espère sincèrement qu'il sera possible de continuer à travailler avec vous sur ce sujet qui a pratiquement ouvert plus de perspectives que véritablement conclu.

Je souhaite maintenant remercier les personnes qui ont dû me supporter au quotidien. Je commence tout d'abord par Stéphane Normand qui a accepté ce sujet de thèse et m'a accueilli dans son laboratoire (le LCAE) dans les meilleures conditions qui soient. Bien que nos domaines d'expertises n'étaient pas vraiment les mêmes, il m'a suivi, encouragé et permis de partir plusieurs semaines à Lorient pour mieux avancer dans mes travaux et je lui en suis très reconnaissant.

Yoann Moline... Que dire ? J'ai découvert, en mon « co-bureau », quelqu'un d'incroyable. Il a supporté mes innombrables questions. Nous avons eu des grands débats sur un très grand nombre de sujets qu'ils soient scientifiques, musicaux ou humains. Bien que pas toujours en accord, ces débats et ces discussions m'ont beaucoup apporté. Nous nous sommes même découvert des passions communes et avons redécouvert le jeu d'échecs et Magic... Il est devenu bien plus qu'un collègue – un ami – avec qui je souhaite continuer à échanger. Merci !

Bien qu'arrivé au sein du laboratoire alors que je commençais ma troisième année de thèse, je tiens à remercier Jonathan Dumazert pour ces discussions toujours passionnantes et profondes que l'on a eues. Comme le disait Pauline dans ses remerciements, il est quelqu'un d'exceptionnel

et je ne peux que la rejoindre sur cet aspect. Un esprit toujours vif, rigoureux ; c'est quelqu'un que j'écoute et que j'apprécie au plus haut point. Bien plus qu'un collègue lui aussi, un ami, presque un frère spirituel, j'espère sincèrement que nous continuerons à échanger et partager.

La quatrième personne que je tiens à remercier nominativement est Licinio Rocha. Il est la figure même de la bonne humeur : toujours souriant, intéressant et intéressé par tout ce que je pouvais lui dire, que cela soit scientifique ou non. Ces interludes m'ont permis de retourner travailler de bonne humeur et « *à fond !* ». Je n'oublierai pas non plus qu'il m'a gentiment charrié pendant une grande partie de la fin de ma thèse sur mes horaires de travail en soulignant que « Ça c'est du thésard ! »... Une autre citation mémorable : « Le thésard est un surhomme : il est capable de rédiger sa thèse toute la nuit et d'être en forme après ! ».

Je tiens ensuite tout particulièrement à remercier les personnes qui m'ont permis de me changer les idées et de relâcher la pression au travers du sport. En particulier Gwenolé, Yoann, Jonathan, Matthieu, Fabien, Guillaume, Marion, Karim, Hassen, Mathieu Tr., Cheik, Jordan et Emmanuel pour m'avoir fait rejouer au foot ; mais aussi à Thierry, Juan-Carlos, Mathieu A., David et Héléna, collègues du LM2S, pour les footings.

Je tiens à remercier les autres doctorants que j'ai eu la chance de côtoyer. Mes ainés au LCAE, Adrien, Maugan et Pauline, qui m'ont montré la voie et qui ont permis par leurs conseils de mieux appréhender cette aventure. Mais aussi ceux qui soutiendront après moi : Isabelle, Yoann, Hermine, Jonathan, Emmanuel, Héléna, Camille et Éva, qui m'ont soutenu en particulier pendant la fin de ma rédaction. À ces derniers, je ne pourrais que leur recommander d'écouter le *Requiem* de Mozart dirigé par Karajan : c'est véritablement cette musique qui m'a permis de gérer le stress dans les périodes de rush et de terminer mon manuscrit dans les temps... Profitez de ce chef-d'œuvre sans modération !

Je tiens aussi à remercier tous les autres membres du 516, qui ont eux aussi, à leur manière, participé à la réussite de ma thèse, que cela soit en répondant à des questions logistiques comme Nathalie ou Caroline ; sur LaTeX(Maugan, Juan, Mathieu) ou encore en organisant les « restos labo » qui ont permis une certaine unité et convivialité entre nous...

Enfin je tiens à remercier ma famille, en particulier Amélie, Florence, Jack, Jocelyne et Maud, ainsi que mes amis pour le soutien et l'aide qu'ils m'ont apporter ; que cela soit en me changeant les idées ou par des corrections orthographiques (bien que mon manuscrit ne soit probablement pas encore tout à fait « tolérant aux fautes [d'orthographe] » (*François Pécheux*))... Du fond du cœur : merci !

Table des matières

Table des figures

Liste des tableaux

Introduction

De plus en plus de systèmes électroniques autrefois analogiques sont maintenant réalisés avec de l'électronique numérique. Celle-ci présente de nombreux avantages en termes de consommation, de puissance de calcul, de précision, de taille, de flexibilité, mais surtout de reproductibilité. Une grande partie de l'électronique numérique est reprogrammable ce qui permet de réutiliser la même architecture dans différents domaines applicatifs et présente un intérêt économique certain. Parmi les applications les plus contraignantes se placent les traitements du signal en flux tels que les filtrages. Cette famille d'applications impose que le système développé puisse fournir des résultats sans interruption ni retard, c'est-à-dire une contrainte de débit de données et de latence.

Cependant, l'environnement dans lequel est utilisé un appareil électronique n'est pas sans conséquence sur son fonctionnement. La température, le vieillissement et les rayonnements peuvent provoquer des fautes qui engendrent généralement l'apparition d'erreurs dans le fonctionnement du système électronique et impactent les résultats qu'il fournit. Si dans une grande majorité des applications grand public, ces erreurs ont des conséquences limitées, il n'en est pas de même pour les applications critiques qui vont avoir besoin de mécanismes permettant de s'en prémunir. En effet, dans beaucoup d'applications des domaines de l'aéronautique, du nucléaire, du spatial, du militaire et même dans les domaines des applications grand public comme l'automobile, la présence d'erreurs n'est pas acceptable.

C'est pourquoi des technologies spéciales dites « rad-hard » permettant de tolérer une grande partie des fautes ont été proposées [Redmond, 2001, Roosta, 2004, Vázquez-Luque et al., 2013]. Cependant, comme exprimé dans le titre de l'article [Alexander et al., 2008] : « Affordable Rad-Hard – An Impossible Dream ? », ces technologies sont extrêmement coûteuses. De plus, elles sont dépendantes du fondeur et parfois soumises à des règles particulières d'importation (en particulier dans le cadre de l'*International Traffic in Arms Regulation* (ITAR)).

Notre objectif est de réaliser un système numérique complet, programmable, flexible, tolérant aux fautes et pouvant respecter des contraintes de débit et de latence. De plus ce système doit être indépendant de la technologie d'intégration. Cela implique de réaliser la tolérance aux fautes au niveau architectural [D'Angelo et al., 1998, Carmichael, 2006, Bolchini et al., 2011] et nous conduit à définir une nouvelle architecture électronique. Nous avons développé les outils qui permettent de transformer des applications complètes écrites en langage de haut niveau en une représentation exécutable sur cette architecture qui inclut des mécanismes de tolérance aux fautes. Le système complet proposé fonctionne de manière autonome afin de ne pas dépendre de la fiabilité d'un autre composant tel qu'un processeur hôte.

Ce document est organisé comme suit :

Le chapitre 1 présente la problématique de l'utilisation de l'électronique numérique dans un environnement susceptible de provoquer des fautes et les différentes techniques et architectures permettant de mettre en œuvre la tolérance aux fautes. Comme il n'existe pas dans l'état de l'art une architecture capable de conjointement respecter les critères de programmabilité, de contrainte de flux et de tolérance aux fautes, la suite du chapitre se focalise sur le type

d'architecture le plus à même de respecter nos contraintes ainsi que les flots de conception associés.

Le chapitre 2 détaille plus particulièrement la problématique de cette thèse au vu des travaux de l'état de l'art, justifie nos choix et présente la méthode globale, à la fois matérielle et logicielle, que nous avons retenue pour la résoudre. En particulier, les implications des choix des méthodes de tolérance aux fautes sur le flot de conception et l'architecture seront détaillées.

Le chapitre 3 présente une première proposition de flot de conception orienté « tolérance aux fautes » pour le modèle d'architecture retenu et pour projeter des applications ne possédant pas de flots de contrôle.

Le chapitre 4 présente une extension du flot précédemment introduit permettant de projeter des applications plus complexes, *i.e.* possédant un flot de contrôle. Il détaille ensuite comment gérer l'ensemble des configurations qui sont générées par le flot pour reconfigurer efficacement l'architecture.

Le chapitre 5 détaille l'architecture électronique que nous proposons qui permet d'exécuter des applications complexes. Puis il analyse comment la tolérance aux fautes peut être réalisée dans cette architecture. Enfin, il détaille notre proposition permettant de rendre le système global fiable tout en répondant aux contraintes de débit de données et de latence.

Enfin, le dernier chapitre conclut cette thèse et présente ses différentes perspectives.

Chapitre 1

État de l'art

1.1 Introduction

Les composants utilisés en électronique numérique se sont largement démocratisés ces dernières années. Seulement, ils sont sensibles aux rayonnements qui peuvent induire l'apparition d'erreurs. Ce problème a été identifié pour la première fois en 1975 dans les satellites, où il n'y a pas d'atmosphère pour protéger l'électronique, mais depuis 1996, avec la diminution des tailles de transistors, il a été montré que des fautes pouvaient être produites même au niveau de la mer, malgré la protection qu'apporte l'atmosphère [Normand, 1996]. En effet, la probabilité d'apparition est inversement proportionnelle à l'altitude, mais n'est pas nulle au niveau du sol.

L'impact de ces erreurs n'est pas le même en fonction du domaine d'utilisation : les systèmes de navigation d'un avion doivent être protégés alors que les systèmes de confort des passagers sont moins critiques, simplement gênants pour l'utilisateur. L'exemple de la machine de vote électronique de Schaerbeek (Belgique) en 2003, qui a ajouté 4096 voix pour un candidat et dont l'origine, après enquête, a été attribuée à un rayon cosmique [Chambre des Représentants Belgique, 2003], a remis en cause l'utilisation de ce genre d'électronique et provoqué l'abandon du projet de machines de vote électronique en Belgique.

Concevoir des systèmes tolérants aux fautes relativement bas coût se révèle donc indispensable même pour des systèmes non critiques. Pour cela, il faut comprendre les fautes auxquelles cette électronique est confrontée, connaître les méthodes qui existent pour les tolérer et enfin choisir un type d'architecture qui permettra de mettre en œuvre cette tolérance. Dans un premier temps, ce chapitre présente l'origine et les conséquences des fautes sur l'électronique, puis décrit les méthodes de tolérance qui existent pour les tolérer. Enfin, à la lumière de cette analyse, l'état de l'art du type d'architecture électronique qui semble le plus à même d'intégrer ces mécanismes est présenté.

1.2 Les fautes en électronique numérique

En électronique numérique, il est possible d'être confronté à deux grands types de fautes : les fautes « transitoires » et les fautes « permanentes ». On assimile parfois l'erreur à la faute bien qu'elle ne soit que l'observation/la manifestation d'une faute.

1.2.1 Fautes transitoires

Une faute transitoire est une faute qui n'est pas due à un problème matériel du composant, mais induite par son environnement. Ce phénomène apparaît quand une particule (un neutron ou un rayon cosmique, comme illustré dans la figure 1.1) entre en collision avec un atome du circuit électronique. Cette collision entraîne l'apparition de charges électriques de façon proportionnelle à l'énergie de la particule. Ces charges sont à l'origine des erreurs dites transitoires.

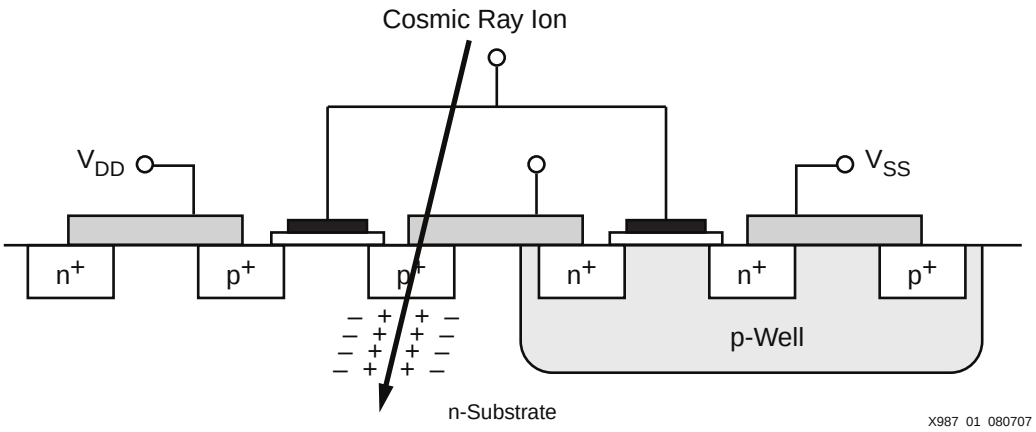


FIGURE 1.1 – Illustration d'une interaction entre une particule et un composant électronique CMOS [Bridgford et al., 2008].

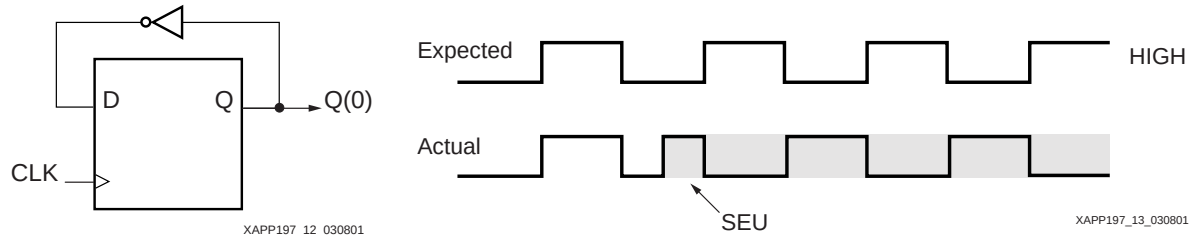
Avec la diminution de la taille des transistors, l'énergie devant être apportée par une particule pour causer une erreur transitoire a diminué, ce qui en augmente la probabilité d'apparition. Ceci est d'autant plus vrai pour les transistors de la technologie *Complementary Metal Oxide Semiconductor* (CMOS) majoritairement utilisés actuellement [Nicolaidis, 2000].

Si une collision a lieu, on distingue deux cas. Dans le premier, la particule est entrée en collision au niveau d'un transistor. Si l'énergie déposée par la particule est suffisamment grande, alors le transistor va changer d'état (de bloqué à passant ou l'inverse). Si ce transistor fait partie d'une cellule mémoire (groupement d'environ 6 transistors) ou d'un registre, alors la valeur mémorisée peut changer (passant d'un '0' logique à un '1' logique). C'est ce qu'on appelle un *Single Event Upset* (SEU). Dans le second, la particule est entrée en collision au niveau d'une interconnexion du circuit, c'est-à-dire un « fil » conducteur raccordant deux ou plus éléments logiques entre eux. Dans ce cas, on observe une variation de tension (due à l'apport de charges) qui peut être vue comme une inversion logique momentanée ('0' vers '1' par exemple). C'est ce qu'on appelle un *Single Event Transient* (SET). Un SET n'est pas une erreur en tant que telle, mais il peut en engendrer une. En effet, s'il se produit à l'entrée d'un registre juste avant qu'il ne mémorise la valeur, le contenu du registre sera erroné, ce qui aura les mêmes conséquences que si un SEU s'était produit dans le registre. Un SET est caractérisé par sa durée, qui est proportionnelle à l'énergie de la particule, car elle permet de connaître la probabilité qu'il se « transforme en SEU ».

Pour illustrer les problèmes que peut entraîner une faute transitoire, considérons le compteur 1-bit auto-incrémentant de la figure 1.2a. Son comportement normal consiste à changer de valeur à chaque coup d'horloge. Dans la figure 1.2b, un SEU fait passer la valeur du compteur de '0' à '1' entre deux coups d'horloge. Ceci a pour effet d'inverser la sortie du compteur. Cette inversion sera propagée tout au long du fonctionnement. Il peut y avoir trois origines différentes à ce SEU, mais qui ont exactement la même conséquence :

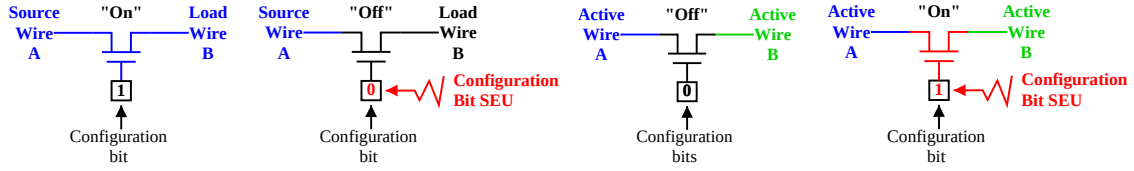
- un SEU dans le registre ;
- un SET sur l'inverseur ou le « fil » reliant la sortie « Q » à l'entrée « D » ;
- un SET sur le « fil » de l'horloge.

Remarque : Dans le cas particulier d'une Architecture Reconfigurable à Grain Fin (FPGA), un SEU survenant dans la mémoire de configuration (*bitstream*) peut être vu comme une erreur permanente car il affecte le fonctionnement du composant sur le long terme en changeant la fonctionnalité, voire en le rendant inopérationnel [Quinn et al., 2008]. Ce type d'erreur est appelé *Single Event Functionnal Interrupts* (SEFI). La figure 1.3 illustre quatre exemples des fautes qui peuvent survenir dans un FPGA :



(a) Schéma d'un compteur-1 bit. (b) Chronogramme de fonctionnement avec apparition d'un SEU.

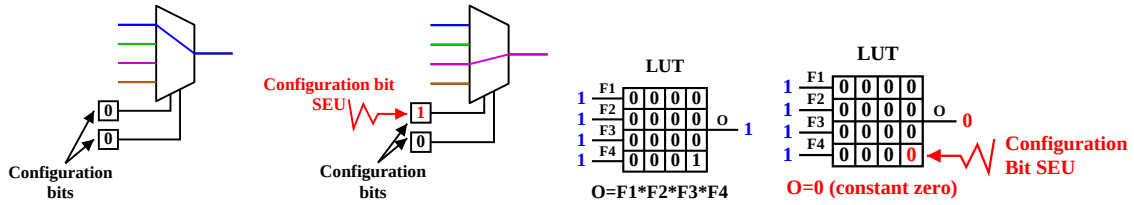
Figure 1.2 – Exemple de l'impact d'un SEU sur le fonctionnement d'un compteur 1-bit [Carmichael, 2006].



1) État initial. 2) État final. 1) État initial. 2) État final.

(a) Exemple de SEFI sur fil d'interconnexion en état passant.

(b) Exemple de SEFI sur fil d'interconnexion en état bloqué.



1) État initial. 2) État final. 1) État initial. 2) État final.

(c) Exemple de SEFI sur multiplexeur. (d) Exemple de SEFI sur une LUT.

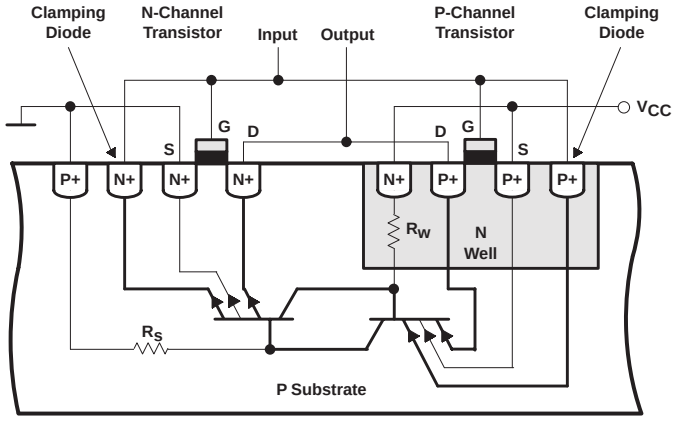
Figure 1.3 – Exemple de différents SEFIs sur FPGA [Quinn et al., 2008].

— dans les parties (a) et (b), les deux changements d'états d'un interrupteur de l'interconnexion (élément majoritaire d'un FPGA) ;
— dans la partie (c), l'impact sur un multiplexeur, changeant la donnée transmise ;
— et dans la partie (d), sur une table de scrutation ou *Look-Up Table* (LUT) pour laquelle la fonction qui était un ET logique sur les quatre entrées devient la constante '0'.

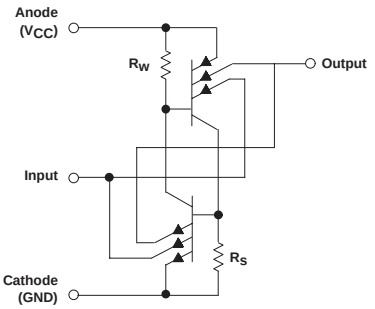
Lorsque ce type d'erreur se produit, seul un rechargement du *bitstream* permettra de corriger cette erreur (voir section 1.3.2.2).

1.2.2 Fautes permanentes

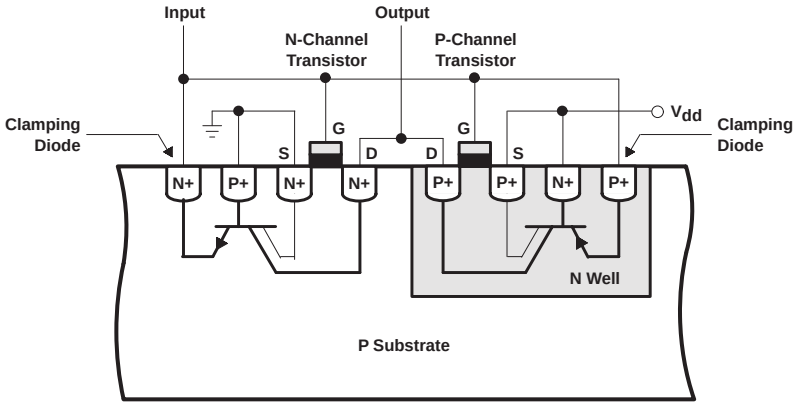
Une erreur permanente est une manifestation d'une défaillance, appelée faute permanente, d'une partie d'un composant électronique. Des erreurs permanentes typiques sont celles du « *stuck-at-'0'/'1'* », c'est-à-dire qu'en sortie d'un registre ou d'une partie combinatoire, la valeur d'un bit est toujours la même (collage à '0' ou '1'). Les erreurs permanentes sont généralement dues à une destruction locale du circuit électronique qui peut être provoquée par le vieillissement (par exemple : l'électromigration) ou par un rayonnement extérieur. Par exemple la collision d'un neutron avec un atome de silicium libère des charges électroniques, mais entraîne aussi un changement infime du dopage du silicium qui, à long terme (après un grand nombre de collisions), engendrera la défaillance de l'électronique. Dans une technologie CMOS « classique », le rayonnement neutronique peut avoir une action directement destructive pour le composant : le *Single Event Latch-up* (SEL). Il s'agit d'un mécanisme qui détruit localement une



(a) Thyristors parasites.



(b) Schéma équivalent des thyristors parasites.



(c) Transistors bipolaires parasites.

FIGURE 1.4 – Transistors parasites intrinsèques à la technologie CMOS [Haseloff, 2000].

paire de transistors et qui est inhérent à la façon de réaliser les transistors *Metal Oxide Semiconductor* (MOS). Comme l'illustre la figure 1.4, l'assemblage des deux transistors dopés n et p, réalisant le transistor CMOS, crée des transistors bipolaires et thyristors parasites. Un apport d'énergie suffisant, provenant d'un neutron par exemple, peut engendrer un court-circuit dans les thyristors parasites qui ont une tension de claquage bien inférieure aux transistors bipolaires parasites [Haseloff, 2000]. Si rien n'est fait pour limiter le courant, le silicium fondra localement, détruisant définitivement le transistor et éventuellement détériorant d'autres parties du circuit.

Un autre type de rayonnement (le rayonnement gamma) interagit lui aussi avec l'électronique et augmente l'ionisation globale du composant, réduisant ses performances jusqu'à ce qu'il ne fonctionne plus par impossibilité de différencier l'état électrique « haut » de l'état électrique « bas », c'est-à-dire de différencier un '1' d'un '0'. La dose totale minimale de rayonnement qu'est capable de supporter un composant est appelée *Total Ionizing Dose* (TID). Cette information est généralement disponible uniquement pour les composants dits « durcis », c'est-à-dire testés et qualifiés pour résister à des rayonnements. C'est le cas par exemple pour des utilisations militaires, spatiales ou nucléaires. Cependant, ces composants sont extrêmement couteux (environ 100 fois le tarif d'un composant non durci [Roosta, 2004, Grassi, 2008]) et ont plusieurs générations de retard (*e.g.* seulement le Virtex-5QV alors que le Virtex-7 est disponible), ce qui réduit fortement les possibilités de les utiliser pour réaliser des systèmes relativement bas coût [1].

1. Par exemple, pour un Virtex-4QV, il faudra dépenser plus de 24 000$ alors qu'un Virtex-4 peut être acheté pour moins de 200€ et un Virtex-5 pour moins de 250€ (prix Radiospare au 03/10/2014 pour les composants grand public).

1.2.3 Fautes intermittentes

Il est possible de trouver dans la littérature une troisième catégorie de fautes dites « intermittentes » qui révèlent un fonctionnement aléatoire du composant. Dans [Guilhemsang, 2011], il est donné un modèle probabiliste pour ces fautes ainsi qu'une caractérisation de leur apparition. Ces fautes surviennent généralement par « rafale », c'est-à-dire avec plusieurs cycles successifs de fonctionnement erroné. Du fait de ce comportement, si la « rafale » est longue, on considérera généralement que le composant a une erreur permanente et si la « rafale » est courte elle sera considérée comme étant un petit nombre d'erreurs transitoires successives. Nous proposons de classer les composants présentant des erreurs intermittentes comme ayant des erreurs permanentes. Ceci permet d'éviter de les utiliser et ainsi de limiter le risque d'apparition de fautes.

1.3 La tolérance aux fautes

Cette section présente les différentes méthodes permettant de tolérer les fautes sur des composants électroniques numériques. Il existe deux grandes catégories de tolérance aux fautes qui dépendent du niveau d'abstraction [Abd-El-Barr, 2007] : la tolérance aux fautes au niveau matériel d'une part et la tolérance aux fautes au niveau architectural d'autre part. La littérature est très riche dans chacun de ces deux domaines qui ne sont pas à opposer et peuvent être complémentaires. L'objectif de cette thèse étant de permettre la mise en œuvre de la tolérance aux fautes sans être dépendant de la technologie de fabrication, la tolérance au niveau matériel n'est que très rapidement abordée via ses grands principes.

1.3.1 Tolérance au niveau matériel

Il est possible d'empêcher l'apparition de certaines fautes de manière technologique. Ces composants sont dits « *rad-hard* ». Ils sont qualifiés et certifiés (qualité militaire « Q » ou spatiale « V »). Ces qualifications permettent de connaître la valeur de leur TID, l'énergie minimale déclenchant une SEL et le nombre de SEFIs qui peuvent survenir à une certaine orbite. Le tableau 1.1 donne certaines valeurs pour des FPGAs Xilinx Virtex 4QV et 5QV. Pour comparaison, un composant électronique « classique » ne fonctionnera plus après avoir subi une dose de quelques dizaines de milliers de *rad*.

Certaines technologies de fabrication, comme par exemple la technologie *Silicon On Insulator* (SOI), permettent d'éliminer complètement le problème des SELs en empêchant la création des transistors parasites comme illustré dans la figure 1.5 [Redmond, 2001]. En effet, les tranchées isolantes ajoutées entre les compartiments n et p ainsi que la couche d'isolant sur laquelle sont formés les transistors empêchent ces parasites d'apparaître. Dans la technologie SOI, l'isolant utilisé est de l'oxyde de silicium SiO_2. Dans une autre technologie appelée *Silicon On Sapphire* (SOS), l'isolant utilisé est de l'oxyde d'aluminium (Al_2O_3).

Lorsque ce type de procédé de fabrication n'est pas choisi, il est possible d'utiliser pour le substrat des matériaux présentant une large bande interdite. Ceci a pour effet de limiter la propagation des électrons et des trous à proximité du lieu de l'interaction entre le rayonnement et le substrat. C'est en particulier le cas des substrats à base de carbure de silicium et

Tableau 1.1 – Caractéristiques des FPGAs Xilinx 4QV et 5QV [Xilinx, 2010, Xilinx, 2012].

Modèle	TID (min) $krad(Si)$	SEL immunity (min) $Mev.cm^2/mg$	SEFI (typical)	
			$Upset/device/day$	$Upset/bit/day$
Virtex-4QV	300	100	1.5×10^{-6}	∅
Virtex-5QV	1000	100	2.76×10^{-7}	3.80×10^{-10}

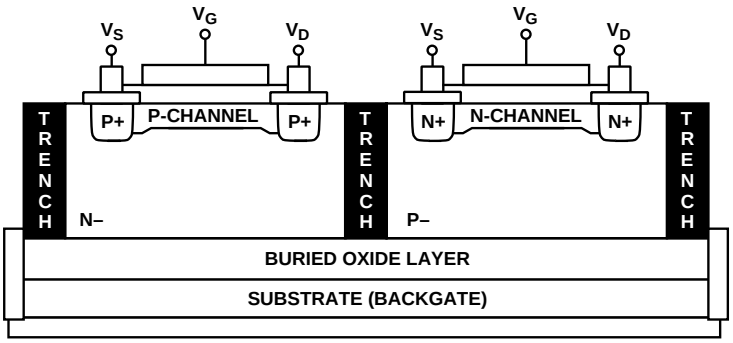


FIGURE 1.5 – Exemple de réalisation d'un transistor CMOS sans transistor parasite avec une technologie *Silicon On Insulator* [Redmond, 2001].

du nitrate de gallium. Pour limiter l'apparition de fautes, il est aussi possible de blinder de façon extérieure le composant. Par exemple, l'ajout de plomb permet de stopper les gamma, mais cette méthode n'est plus utilisée entre autres à cause de la directive européenne environnementale sur la « restriction de l'utilisation de certaines substances dangereuses dans les équipements électriques et électroniques » ou *Restriction of the use of certain Hazardous Substances in electrical and electronic equipment* (RoHS) sur les composants électroniques. Il est possible d'utiliser pour le circuit un verre de borophosphosilicate appauvri en bore 10 (^{10}B) comme dans [Vázquez-Luque et al., 2013], qui possède une section efficace de capture des neutrons plus faible et ainsi réduit l'apparition d'erreur.

À un niveau d'abstraction un peu plus élevé, on trouve les méthodes permettant d'obtenir de la tolérance aux fautes en changeant la manière dont sont assemblés les transistors et leurs dimensions [She et al., 2010, Krishnamohan and Mahapatra, 2005, Blaauw et al., 2008, Fazeli and Miremadi, 2008]. Par exemple, dans [Fazeli and Miremadi, 2008], une architecture de *latch* tolérant aux SEUs est proposée. Pour cela, l'architecture du *latch* SETUR est composée, comme le montre la figure 1.6, de deux *latchs* classiques avec en plus un troisième élément appelé « *C-element* ». Un élément de délai permet au *latch* de ne changer de valeur que si la donnée est stable. Ainsi, le « *C-element* », pour lequel les rapports de largeur sur longueur (W/L) des transistors ne sont pas les mêmes que pour les autres transistors du circuit, permet de conserver la bonne valeur dans le *latch* tant que l'énergie apportée reste inférieure à un certain seuil. Ce seuil est directement relié au rapport W/L. Toutefois, ce type d'approche n'est accessible qu'aux fondeurs et aux conceptions « *full-custom* ».

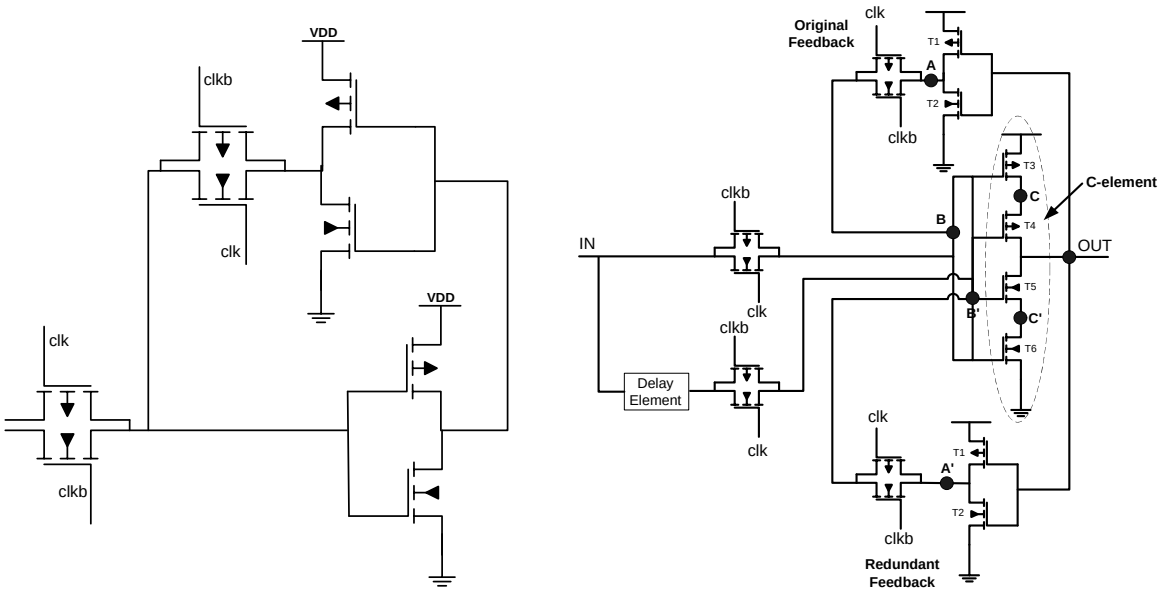
1.3.2 Tolérance au niveau architectural

La tolérance au niveau architectural est un domaine de recherche très vaste. De très nombreuses études ont été menées sur ce sujet. Dans cette section, nous commencerons par présenter les fautes transitoires, puis passerons aux fautes permanentes. Un bilan sera ensuite établi.

1.3.2.1 Tolérance aux fautes transitoires

La question de la tolérance aux fautes transitoires n'est pas récente. Elle a été étudiée et il existe de nombreuses méthodes classiques que l'on peut classer en trois grandes catégories :

- la détection de fautes, qui consiste à simplement savoir qu'une erreur s'est produite et que le résultat ou la donnée reçue est potentiellement erroné ;
- la correction de fautes, qui consiste d'une part à savoir qu'une erreur s'est produite, mais aussi à savoir dans quelle partie du résultat afin de pouvoir le corriger ;
- le masquage de fautes, qui est un mécanisme quasiment automatique qui permet d'obtenir le résultat correct sans nécessairement informer l'utilisateur qu'une erreur a pu se



(a) *Latch* standard. (b) *Latch* SETUR.

Figure 1.6 – *Latch* standard et tolérant aux SEUs [Fazeli and Miremadi, 2008].

produire. Il se différencie de la correction de fautes par le fait qu'il n'est pas possible de savoir si une faute est apparue ou non d'un point de vue extérieur (*e.g.* pas de latence supplémentaire).

La plupart de ces méthodes sont caractérisées par le nombre maximal d'erreurs qu'elles supportent.

Détection de fautes

Pour savoir qu'une erreur s'est produite, deux grandes approches existent. La première consiste à ajouter à la donnée de l'information permettant de vérifier si elle est correcte. Une technique parmi les plus simples est l'ajout d'un bit de parité (par exemple, le protocole RS232 utilise huit bits utiles dont un bit de parité). Il est possible de détecter avec cette méthode n'importe quel nombre impair de modifications sur le mot de donnée reçu. Par exemple, pour transmettre « 0100110 » avec une parité impaire (c'est-à-dire que le nombre de 1 du message doit être impair), un 0 est ajouté au message. C'est cette approche qui débouchera sur les codes correcteurs d'erreurs (voir section 1.3.2.1). En revanche, avec un seul bit de parité, il n'est pas possible de détecter un nombre pair d'erreurs. En effet, les mots « 0100110 » et « 0100000 » ont la même parité.

La seconde approche est la *Duplication With Comparison* (DWC). Elle consiste à doubler les opérations et à comparer si les résultats obtenus sont les mêmes. Cette duplication peut avoir lieu à différents niveaux d'abstraction et s'effectuer de différentes manières (*e.g.* séquentiellement ou concurremment). Par exemple, envoyer deux fois le même message permet de détecter si une erreur est survenue puisque la probabilité que l'erreur se produise deux fois au même endroit est faible. Il est aussi possible de dupliquer le chemin de données au sein des unités de traitement. Cela nécessite deux fois plus de ressources matérielles, mais il n'y a pas besoin d'attendre une seconde itération.

Corrections de fautes

Détecter qu'une faute a eu lieu est une chose, mais corriger cette erreur est bien plus intéressant du point de vue de l'utilisateur. En prenant comme point de départ la DWC, il est possible de créer une méthode qui corrige les erreurs. Lorsqu'une erreur a été détectée, il suffit de relancer une troisième fois l'opération (envoi de message, calcul, ...). À ce moment, il sera possible d'effectuer un vote majoritaire sur les trois résultats obtenus pour sélectionner le bon. Cette méthode est utilisée dans [Scholzel, 2010] pour rendre un processeur tolérant aux fautes.

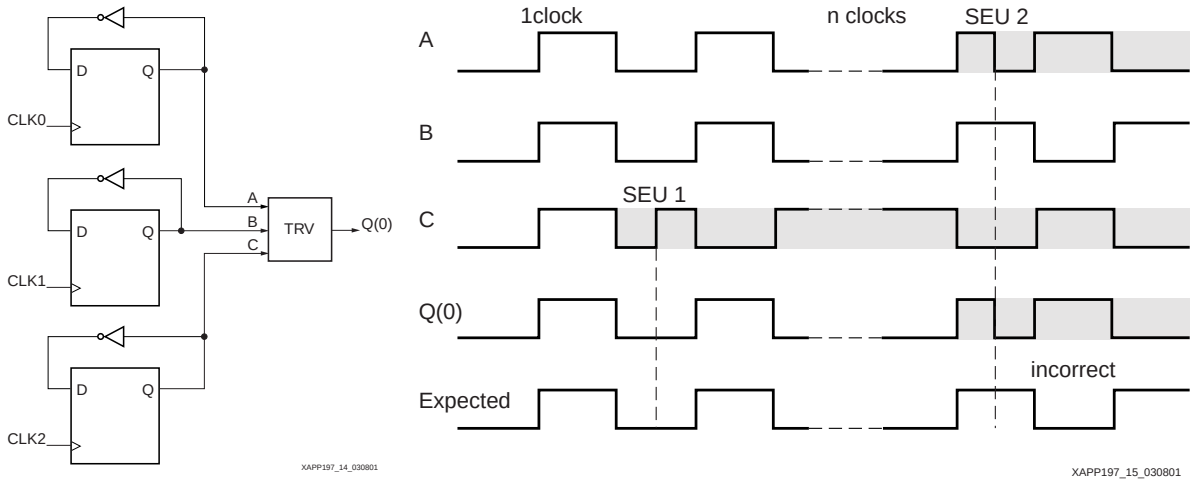
L'idée d'ajouter de l'information aux données a été poursuivie et a abouti aux Codes Correcteurs d'Erreurs (ECCs) [Peterson and Weldon, 1972, MacWilliams and Sloane, N. J., 1977]. Le but est d'introduire une « distance » entre les mots valides. Plus cette distance est grande, plus il est possible de corriger d'erreurs dans un mot. Par exemple si « 0 » est codé par « 0000 » et « 1 » par « 1111 », alors, si on reçoit « 0100 », il est fortement probable que ce soit un « 0 » qui ait été envoyé. La classe de code correcteur la plus employée est celle de Hamming qui a aussi défini la distance la plus utilisée (distance de Hamming) qui correspond au nombre de bits séparant deux mots valides. Il existe un lien entre la distance de Hamming et le nombre d'erreurs qu'il est possible de détecter et de corriger. Si D est la distance minimale entre deux mots valides, il est possible de détecter $D-1$ erreurs et d'en corriger au maximum $\lceil D/2-1 \rceil$.

Masquage de fautes

Le masquage de fautes consiste à intégrer des mécanismes qui sont « transparents » pour avoir le bon résultat, c'est-à-dire qu'il n'y a pas de temps ou de calculs supplémentaires pour connaître le bon résultat. La *Triple Modular Redondancy* (TMR), dont les concepts de base ont été introduits initialement par von Neumann [Neumann, 1956], est la méthode de masquage la plus simple et aussi une des méthodes les plus employées. Elle consiste à multiplier par trois (tripliquer) tous les éléments de calcul ou de mémorisation et d'ajouter des votes majoritaires pour déterminer la bonne valeur [D'Angelo et al., 1998, Carmichael, 2006]. La mise en œuvre de la TMR n'est pas forcément une chose simple comme l'illustre la figure 1.7 pour l'exemple du compteur 1-bit de la figure 1.2. En effet, avec cette implémentation, l'architecture n'est capable de corriger qu'une seule erreur pour un coût triple en ressource. Une amélioration serait de reboucler la sortie du voteur de la figure 1.7a vers les entrées des compteurs tripliqués, mais cette version ne résout pas tous les problèmes car si un SEU apparaît en sortie du voteur, les trois compteurs seront affectés et prendront la mauvaise valeur. La proposition de la figure 1.8, préconisée par von Neumann dans [Neumann, 1956], permet non seulement un masquage, mais une réelle correction de l'erreur grâce à la triplication de la logique et des voteurs. Il faut donc prêter le plus grand soin lorsque l'on veut utiliser ce genre de méthode de tolérance.

La littérature fournit beaucoup d'améliorations ou d'évolutions de la triplication : un placement amélioré des « voteurs » [Johnson and Wirthlin, 2010], des versions relativement « bas-coûts » qui ne protègent pas nécessairement l'intégralité des résultats [Samudrala et al., 2004, Schweizer et al., 2011, Bolchini et al., 2011], etc.

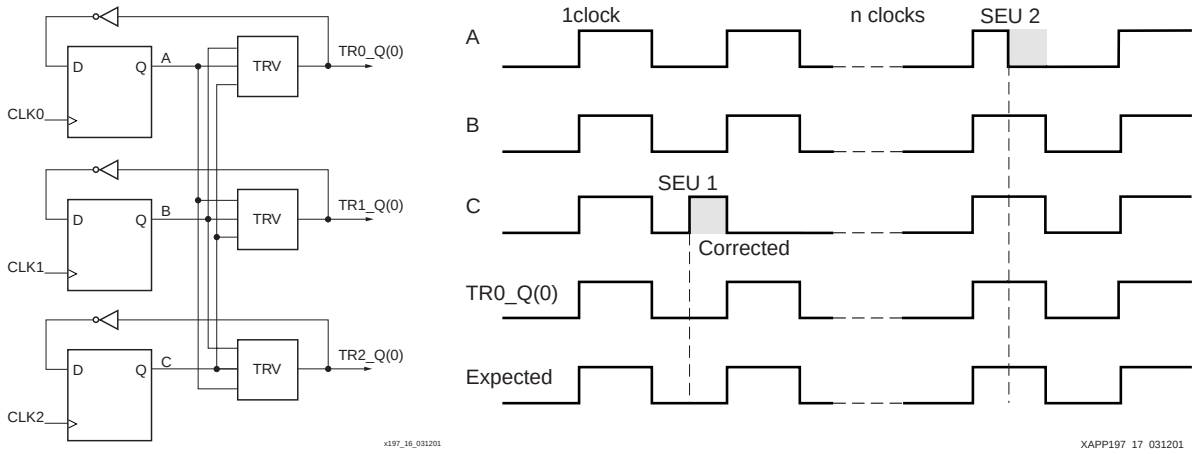
Dans [Lima et al., 2003a, Lima et al., 2003b], une approche hybride entre la DWC et la TMR a été proposée. Les parties combinatoires sont dupliquées. Elles sont ensuite échantillonnées par des registres décalés temporellement d'une demi-période. Quatre valeurs sont donc utilisées pour déterminer laquelle est correcte. Le décalage temporel permet de ne pas prendre en compte un éventuel SET, pourvu que la durée du SET soit inférieure à une demi-période d'horloge. La valeur correcte est ensuite mémorisée dans des registres qui sont tripliqués et dont l'horloge est encore une fois décalée par rapport à celles qui échantillonnent les données. L'architecture résultante est donnée dans la figure 1.9 avec à gauche la duplication des parties combinatoires, à droite la triplication des registres et au centre la mécanique permettant de choisir la bonne valeur.



(a) Schéma d'implémentation.

(b) Chronogramme de fonctionnement avec deux SEUs.

Figure 1.7 – Implémentation non-satisfaisante de la TMR pour un compteur 1-bit ne permettant le masquage que d'un SEU [Carmichael, 2006].



(a) Schéma d'implémentation.

(b) Chronogramme de fonctionnement avec deux SEUs.

Figure 1.8 – Implémentation correcte de la TMR pour un compteur 1-bit [Carmichael, 2006].

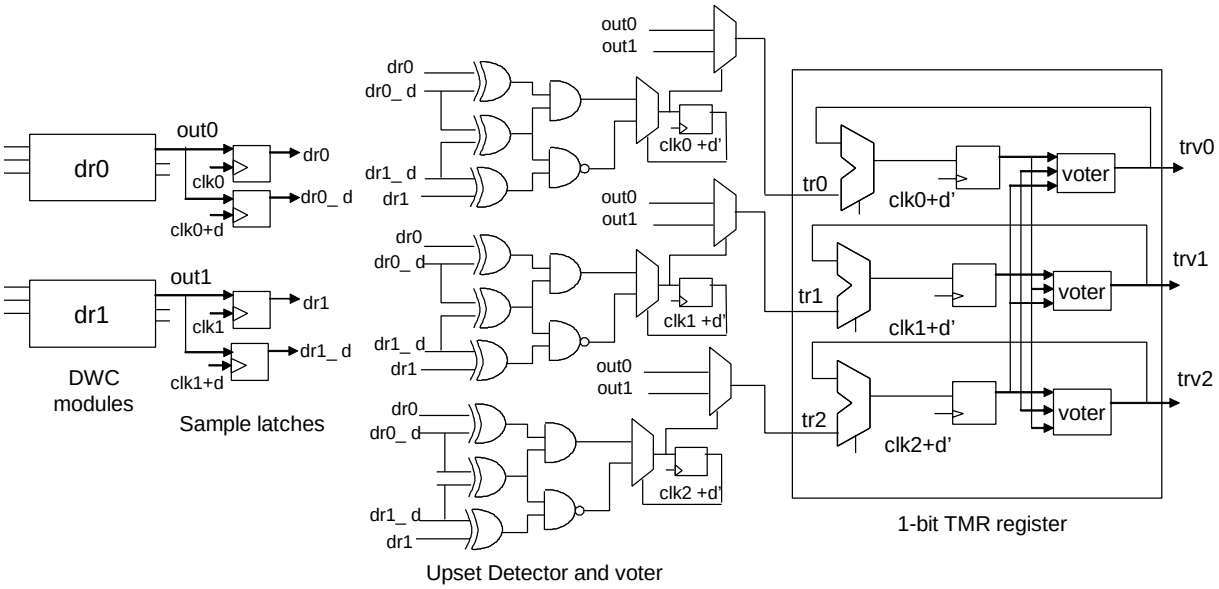


FIGURE 1.9 – Schéma pour 1-bit de la tolérance par duplication avec comparaison et redondance temporelle [Lima et al., 2003a, Lima et al., 2003b].

Avec la multiplication spatiale des calculs pour produire une donnée, un nouveau type d'erreur est possible. Il s'agit des modifications multiples sur un même bit, appelées *Multiple Bit Upset*s (MBUs). La TMR ne permet pas de se protéger contre ce type d'erreurs (et même pire, si deux erreurs surviennent sur un bit, alors la sortie « corrigée » sera en fait la version erronée). Dans les cas particuliers de deux ou trois SEUs simultanés on utilise respectivement les termes *Double Event Upset* (DEU) et *Triple Event Upset* (TEU). De nouveaux mécanismes ont été trouvés pour prendre la bonne décision comme dans [Baloch et al., 2006] où, en plus de la triplication avec trois horloges décalées d'un quart de cycle, un deuxième échantillon de chaque voie tripliquée, décalé d'une demi-période d'horloge, est mémorisé. De cette manière, il y a six échantillons en entrée du voteur ce qui permet de masquer tous les DEUs et une partie des TEUs comme présenté en figure 1.10a. Mais, comme l'illustre la figure 1.10b, cette technique présente l'inconvénient de faire fonctionner l'architecture deux fois moins vite.

En ce qui concerne la tolérance aux fautes transitoires dans les Files de Registres (RFs), la triplication et les méthodes basées sur des ECCs ont été analysées et comparées dans la littérature. Les graphiques de la figure 1.11, tirés de [Naseer et al., 2006], illustrent les résultats de l'analyse sur le taux d'erreur, les impacts sur la surface et les temps d'accès qu'ont quatre méthodes différentes de tolérance aux fautes transitoires pour une RF possédant des mots de 32 bits. Les quatre méthodes comparées sont la triplication et trois mises en œuvre de code de Hamming : (7, 4), (12, 8), (21, 16) et (38, 32) [2]. La figure 1.11a présente la probabilité d'avoir une erreur sur un mot stocké dans la RF par rapport à la probabilité d'apparition d'une erreur pour un bit. On remarque que les quatre méthodes ont des comportements assez similaires par rapport aux taux d'erreur et que c'est la triplication qui offre les meilleurs résultats. La figure 1.11b donne pour les quatre méthodes le surcoût en surface et en temps d'accès. Dans le cas de la triplication, le surcoût en surface est de 204%, dont 200% pour la triplication en tant que telle et 4% pour le voteur. Pour les méthodes utilisant des codes de Hamming, le surcoût en surface diminue avec la taille du code (de 67% à 27%) grâce à la mutualisation des ressources de correction. La tendance pour la latence d'accès est opposée : plus le code de Hamming est grand et plus cette latence est importante. La triplication n'augmente que très peu la latence, et en lecture seulement, par le besoin de traverser l'étage de vote. Il existe donc un compromis

2. (n, m) signifie que les mots codés possèdent n bits dont m significatifs, ceux restants servant à la redondance.

à faire entre surcoût en surface et temps d'accès. La conclusion de cette étude est qu'il faut privilégier la triplication si la latence est critique et que le code (7, 4) est un bon compromis dans la majeure partie des autres cas.

C'est ce compromis qui a été utilisé et amélioré dans [Esmaeeli et al., 2011] pour des mots de tailles plus importantes. La figure 1.12 présente le taux d'erreur et le surcoût en surface pour cette amélioration. On remarque sur la figure 1.12a qu'elle obtient un taux d'erreur plus faible que la triplication pour des mots de 64 bits. En revanche, la figure 1.12b illustre le fait qu'elle n'est rentable en surface que si les mots stockés sont de grande largeur (au minimum 64 bits) du fait de la complexification de la logique d'encodage et de décodage qu'elle implique. Cette complexification ajoute aussi de la latence à la lecture par rapport aux autres codes correcteurs.

1.3.2.2 Tolérance aux fautes permanentes

Les méthodes présentées jusqu'ici adressent plus particulièrement le problème des fautes transitoires. Les méthodes utilisant de la redondance matérielle se basent sur le fait que toutes les voies de calcul donneront majoritairement le bon résultat. Or, pour la TMR par exemple, si une des trois voies est défectueuse et si une des deux voies restantes a une erreur transitoire, alors le résultat du vote sera faux.

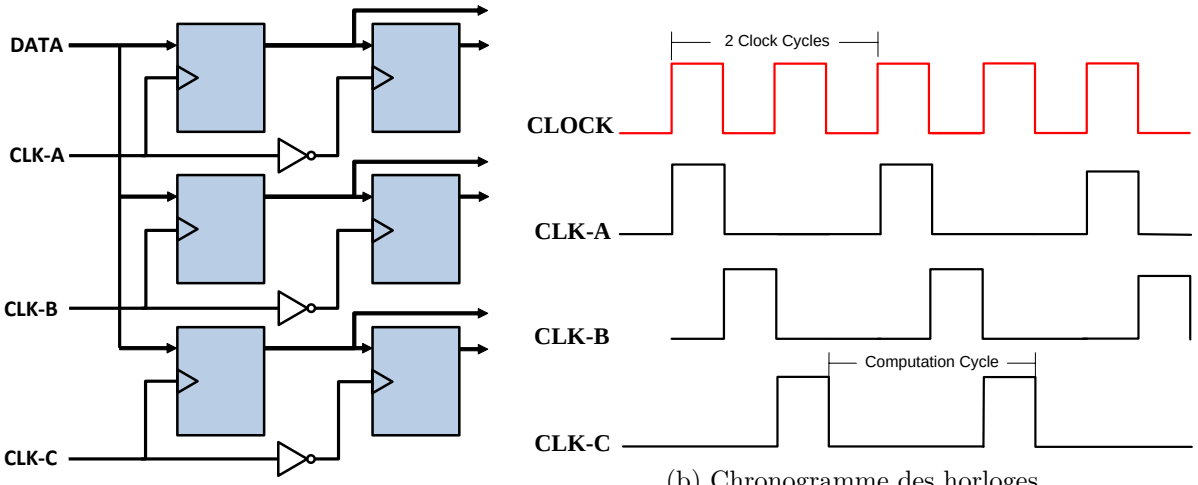
Détection de fautes

Il faut être capable de savoir si les unités de calcul matérielles fonctionnent correctement. Pour cela, deux méthodes sont envisageables :

- la première est de tester le matériel en mettant en entrée des valeurs connues et en vérifiant que les sorties sont correctes. Cette méthode est classiquement utilisée pour vérifier le fonctionnement d'un composant après sa fabrication, en utilisant par exemple du *Built-In Self-Test* (BIST) [Mccluskey, 1985, Voyiatzis and Halatsis, 2005], mais elle nécessite de pouvoir effectuer des tests sur le matériel et donc de la disponibilité. Ce type de test est traditionnellement lancé au démarrage du composant et/ou effectué périodiquement. Un compromis doit alors être trouvé entre le temps d'exécution pour l'application et le temps de test. Cependant, ce type de test ne permet pas une validation en continu du composant et donc a une réactivité limitée qui dépend du compromis utilisé. Aussi, faut-il être sûr que le résultat du test n'est pas altéré par un SEU par exemple.
- Une seconde approche, plus empirique, est de disposer de compteurs permettant de mémoriser le nombre d'erreurs de l'unité de calcul/processeur. Dans un contexte de triplication par exemple, si une erreur permanente apparaît dans une unité, cette dernière donnera souvent un résultat faux. La décision de ne plus utiliser cette unité dépend du seuil fixé sur le nombre d'erreurs [Aliee and Zarandi, 2011]. Un raffinement de cette technique est de pouvoir décrémenter ce compteur quand un certain nombre de résultats sont corrects pour autoriser quelques erreurs transitoires aux unités de calcul.

Correction de fautes

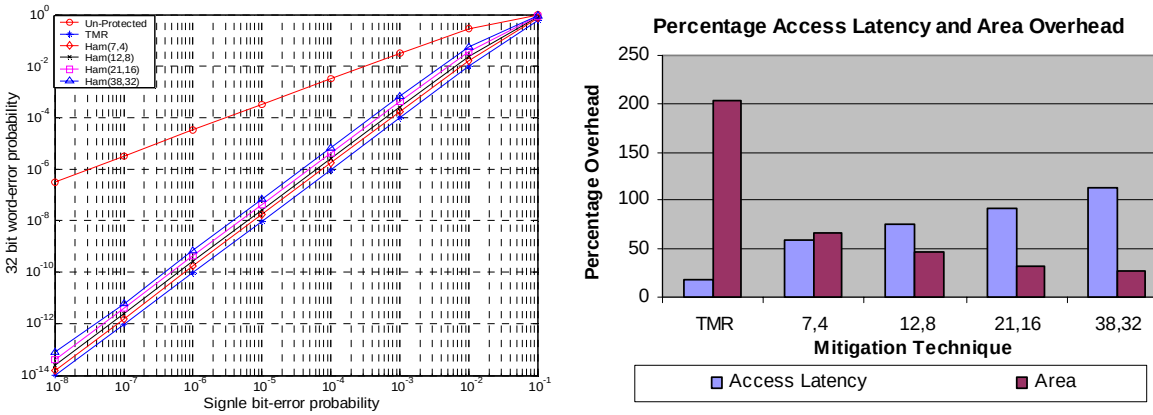
Après avoir détecté la présence d'une erreur permanente, il faut être capable d'en tenir compte. Ceci implique généralement de la reconfiguration/reprogrammation pour « éviter » d'utiliser l'unité de calcul endommagée [Cheatham et al., 2006]. Sur FPGA, la reconfiguration dynamique et partielle permet de changer à la volée la configuration. Mais elle implique d'avoir en mémoire la nouvelle configuration car les processus de synthèse et d'optimisation ne sont pour l'instant pas réalisables en ligne. Dans [Bolchini et al., 2011], un exemple de flot de conception orienté fiabilité est présenté. Il permet d'intégrer de la détection ou du masquage de fautes sur les parties définies par le concepteur comme étant à protéger, mais surtout permet de reconfigurer dynamiquement des sous-parties fonctionnelles en cas de détection d'une faute dans le *bitstream*. La reconfiguration dynamique nécessite l'ajout d'un composant externe au FPGA pour fonctionner correctement en présence de rayonnement (le gestionnaire de reconfiguration).



(a) Schéma d'implémentation.

(b) Chronogramme des horloges.

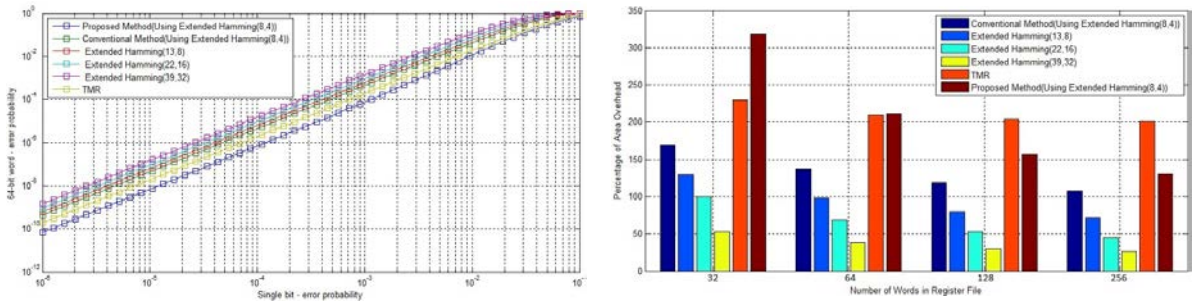
FIGURE 1.10 – Implémentation de la TMR avec double échantillonage des données proposée dans [Baloch et al., 2006].



(a) Probabilité d'apparition d'une erreur.

(b) Compromis surface - latence induite.

FIGURE 1.11 – Comparaison de la triplication et de trois variantes de code de Hamming [Naseer et al., 2006].



(a) Probabilité d'apparition d'une erreur.

(b) surcoût en surface.

FIGURE 1.12 – Comparaison des méthodes de protection pour des RFs de grandes largeurs proposée dans [Esmaeeli et al., 2011].

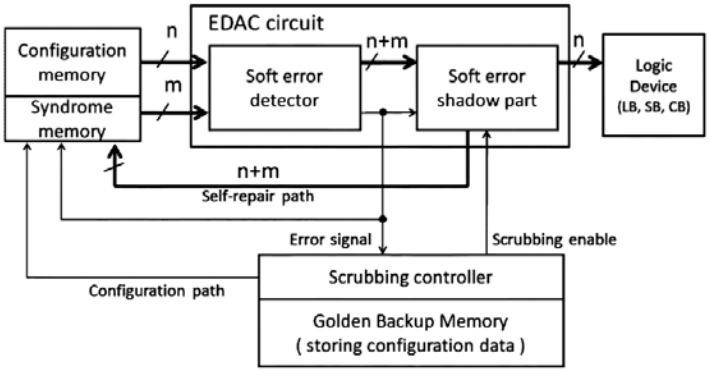


Figure 1.13 – Illustration de l'intégration d'un EDAC entre la mémoire de configuration d'un FPGA et la partie logique [Zhao et al., 2011].

Dans [Huang and Mccluskey, 2001], la tolérance est apportée dans le FPGA par le changement de configuration en déplaçant le placement des opérations de colonne en colonne au fur et à mesure de l'apparition des fautes. Ainsi, si une faute survient dans la colonne i, alors les opérations qui étaient sur la colonne i se déplacent sur la colonne $i+1$, celles de la $i+1$, sur la $i+2$, etc. Cette méthode impose qu'il soit possible de « shunter » les colonnes possédant une faute, ce qui ne pose pas de problème particulier sur ce type d'architecture extrêmement connecté. Elle est néanmoins limitée par le nombre de colonnes libres disponibles dans l'architecture. Dans [Wang et al., 2011] une méthode similaire pour les architectures Architectures Reconfigurables à Gros Grains (CGRAs) est proposée. Toujours dans le cas particulier des FPGAs, où un SEU dans la mémoire peut être vu comme une erreur permanente, une autre technique appelée le « *scrubbing* » de la mémoire, peut être utilisée pour fiabiliser le *bitstream* du FPGA [Berg et al., 2008]. Le principe est de ré-écrire régulièrement la configuration du composant pour supprimer les éventuels SEU qu'il y aurait en mémoire. Cette idée peut être étendue à d'autres mémoires de programmation. Elle suppose néanmoins d'avoir une mémoire de référence tolérante aux radiations, garantissant son contenu.

Dans [Zhao et al., 2011], un module de détection et correction d'erreur (ou *Error Detection And Correction* (EDAC)) est intégré entre la mémoire de configuration et la partie logique du FPGA permettant de corriger une partie des erreurs comme illustré dans la figure 1.13. Lorsqu'une ou plusieurs erreurs dans la mémoire de configuration sont détectées par l'EDAC, elles sont corrigées dans la mesure du possible, sinon, un contrôleur extérieur réalise la réécriture de la mémoire de configuration à partir d'une mémoire de référence (*golden memory* dans la figure) et fournit simultanément au plan logique la véritable valeur de la configuration. Ceci permet d'éviter d'avoir des valeurs erronées ou des instabilités pendant la réécriture de la mémoire. L'avantage principal d'une telle technique est sa réactivité quasi immédiate. Son inconvénient principal est son coût en surface.

L'approche détaillée dans [Rakossy et al., 2013], qui sera plus amplement décrite dans la section 1.4.4, ajoute des ressources supplémentaires à un CGRA pour permettre d'exécuter de l'autotest en parallèle du code applicatif. Les ressources en autotest changent au fur et à mesure de l'exécution de manière à toutes les tester régulièrement. La réactivité de cette approche est cependant limitée par cette périodicité et par le nombre de ressources supplémentaires ajoutées.

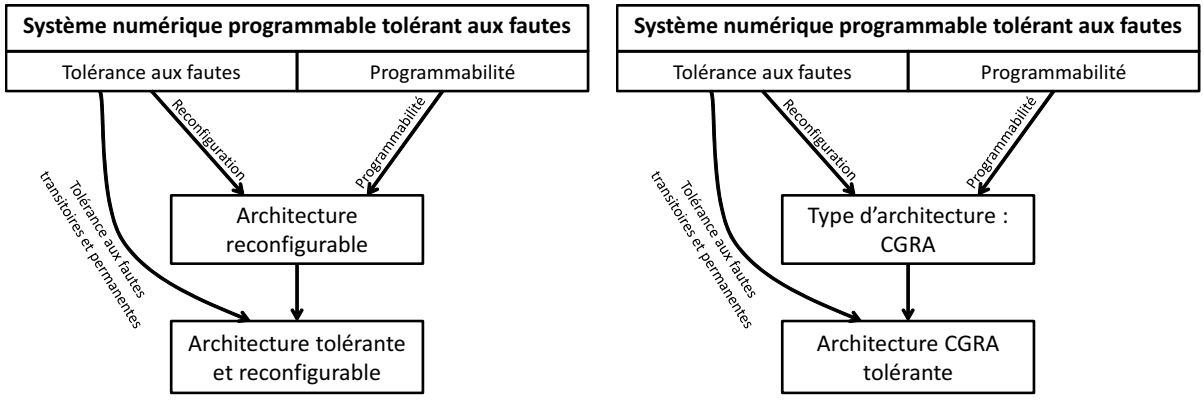
Dans [Abella et al., 2010], une méthode pour tolérer les fautes permanentes dans une RF est présentée. Il s'agit de scinder les registres en plusieurs morceaux et de combiner les morceaux valides pour augmenter la durée de vie de la RF. Cette méthode nécessite d'avoir une table de correspondance entre les indices des registres des instructions et la localisation réelle de la donnée. La limitation de cette approche est que la RF dispose d'un nombre de registres supplémentaires égal au nombre minimal de fautes que l'on veut pouvoir tolérer.

Récemment, de nouvelles approches bio-inspirées ont vu le jour. Ces méthodes s'inspirent des mécanismes vivants pour tolérer les fautes. Les différentes unités de calcul se comportent comme des êtres vivants qui communiquent et qui sont capables d'appréhender leur environnement proche. Ces systèmes multi-agents sont souvent décrits comme étant naturellement tolérants aux fautes car en cas de défaillance d'une unité, une autre prendra sa place. Dans [Pani and Raffo, 2006], un système est réalisé en utilisant ce type d'approche d'intelligence collective dit « *Swarm intelligence* ». Lorsqu'un traitement doit être fait, une unité libre l'effectue. Il n'y a pas comme dans les systèmes classiques d'étape d'ordonnancement, placement et routage ; et donc pas de reconfiguration/replacement dynamique à intégrer. Le problème majeur de ce genre d'approche est qu'elle est non déterministe et donc qu'il n'est pas possible de garantir un débit de données.

1.3.3 Bilan

Il existe de nombreuses méthodes pour tolérer les fautes transitoires et permanentes dans un système électronique. Les domaines d'applications ciblés par ces travaux sont des domaines où il faut être capable de garantir qu'un résultat fourni est juste et où il faut le fournir dans un temps imparti fixé, c'est-à-dire que le système est sous contrainte de temps. Cette condition exclut, d'emblée, les méthodes de tolérance qui ne sont pas capables de fournir un résultat juste, mais simplement de dire que le résultat est erroné comme la DWC. De même, il n'est pas acceptable qu'un résultat puisse être considéré comme juste alors qu'il est en fait erroné : il est donc nécessaire de mettre en œuvre des méthodes qui permettent de corriger les fautes transitoires et permanentes. La seconde condition impose que la correction n'entraîne pas de retard dans l'exécution du code applicatif. En particulier, des solutions comme celle proposée dans [Scholzel, 2010], où en cas de détection d'erreur par duplication des calculs, l'exécution normale s'arrête pour pouvoir réaliser une troisième fois l'opération et ainsi pouvoir effectuer un vote majoritaire, ne répondent pas aux contraintes fixées. La correction de fautes transitoires doit se faire sans intervention, c'est-à-dire être une méthode de masquage de fautes. La triplication est souvent considérée comme étant la référence en matière de masquage de fautes de par sa simplicité et son efficacité, c'est donc ce type de méthode que nous avons choisi de mettre en œuvre pour les opérations de calcul. Son inconvénient majeur est son coût en ressources, mais il reste inférieur au surcoût d'un composant rad-hard. Un autre avantage de la triplication est qu'elle permet aussi, toujours grâce au vote majoritaire, de détecter qu'une faute s'est produite. En ajoutant un simple suivi des fautes (*e.g.* un compteur de nombre d'erreurs successives), il est possible de détecter qu'une faute permanente s'est produite de façon quasi-immédiate. Cet aspect permet donc une plus grande réactivité qu'un système devant tester ses éléments, soit de manière hors ligne, soit de façon périodique, durant l'exécution. La figure 1.14a illustre l'état à ce niveau d'analyse de la proposition.

Après avoir détecté une faute permanente, il faut être capable de la corriger. « Corriger » une faute permanente revient à ne plus utiliser la ressource fautive. Pour cela, il faut d'une part être capable de changer l'utilisation des ressources et d'autre part posséder dans l'architecture un « nombre important » de ressources. Cette seconde condition vient du fait que si l'architecture ne possède qu'un petit nombre de ressources, comme par exemple un processeur multi-cœurs, si une ressource est victime d'une faute permanente, alors un très grand pourcentage de l'architecture ne sera plus utilisable. Pour limiter cet impact, avoir plus de ressources de plus petites tailles semble pertinent, mais il ne faut pas qu'elles soient trop petites. En effet, dans un FPGA par exemple, qui possède une granularité très fine, perdre une ressource n'aura pas forcément de grave conséquence, mais il n'est pas vraisemblable de suivre l'état de l'intégralité de ces ressources (*e.g.* un Virtex 7 de Xilinx possède jusqu'à deux millions de cellules logiques [Xilinx, 2014]), c'est-à-dire d'être capable pour chaque élément de savoir si il est défectueux ou non. Une architecture possédant une granularité de ressources de taille intermédiaire, qui est communément appelée CGRA, semble donc être le bon compromis. Ceci conduit à la figure 1.14b.



(a) Prise en compte de la tolérance aux fautes. (b) Prise en compte de la grannularité.

Figure 1.14 – Illustration du cheminement logique menant à l'utilisation de CGRAs.

1.4 Les Architectures Reconfigurables à Gros Grains (CGRAs)

Comme nous l'avons vu précédemment, le type d'architecture qui semble le plus à même d'être capable de respecter nos contraintes est un CGRA. De nombreuses définitions existent pour définir ce qu'est un CGRA. La définition que nous avons retenue est : « *un CGRA est une architecture capable de modifier son comportement par la modification du schéma de connexion entre ses éléments de base, et ce au niveau des mots de données* ». Cette définition permet de les distinguer des architectures reconfigurables à grains fins (FPGAs) pour lesquelles la modification de comportement peut se faire au niveau des bits.

Durant les vingt dernières années, de très nombreuses architectures CGRA ont été créées. Dans [Hartenstein, 2001], un état de l'art des architectures créées entre 1990 et 2000 a été réalisé. Il montre que ce type d'architecture est très prometteur mais pointe du doigt la nécessité de les associer à des chaînes de programmation automatiques. En effet, durant ces premières années, l'essentiel des architectures ne possédaient pas de flot permettant de les programmer. Cette programmation devait se faire en partie voire entièrement à la main. Les travaux de ces dernières années se sont surtout penchés soit sur la réalisation d'architectures plus économes en énergie, soit sur des flots automatiques permettant d'y porter les codes à accélérer.

Dans cette section, les caractéristiques principales des CGRAs sont présentées ainsi que les différentes techniques existantes permettant d'y projeter des applications. Puis certaines architectures seront plus particulièrement détaillées. Enfin, les méthodes spécifiques de tolérance existantes dans l'état de l'art pour les CGRAs seront décrites.

1.4.1 Caractéristiques principales

Un CGRA est généralement formé d'un ensemble d'unités de calcul gros grains reliées entre elles par un ou plusieurs réseaux d'interconnexion. Ces unités de calcul gros grains, appelées *tuiles*, peuvent posséder ou avoir accès à de la mémoire et/ou des RFs partagées. Les CGRAs se différencient généralement sur les éléments suivants : la tuile, le réseau d'interconnexion et la mémoire.

1.4.1.1 Les tuiles

Une tuile est une unité qui contient la partie opérative du CGRA. Il existe deux critères différenciants concernant les tuiles : la granularité et l'homogénéité.

La granularité de la tuile

Une des architectures possédant la granularité de tuile la plus faible est RICA pour *Reconfi-*

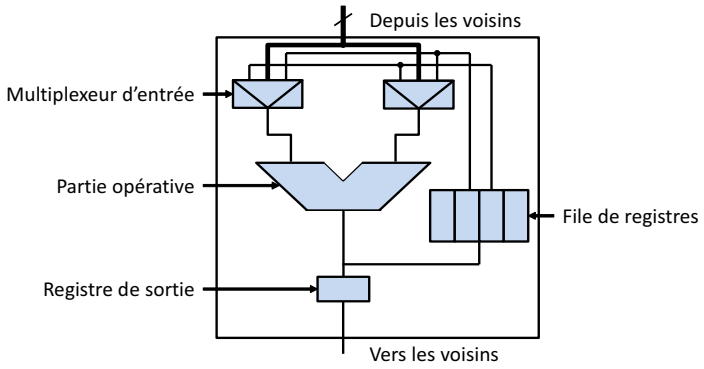


FIGURE 1.15 – Exemple d'une tuile à granularité intermédiaire.

gurable Instruction Cell Array [Khawam et al., 2008]. Chaque tuile ne possède qu'une seule fonction ou opérateur. Par exemple, certaines tuiles possèdent des additionneurs, d'autres des soustracteurs ou de la logique combinatoire, d'autres sont formées uniquement de registres, d'autres permettent d'accéder à la mémoire centrale, etc. L'architecture RaPiD [Ebeling et al., 1996] peut aussi être classée dans cette catégorie.

À une granularité un peu plus importante, les tuiles peuvent effectuer plusieurs opérations avec par exemple avec une Unité Arithmétique et Logique (ALU). Les architectures telles que KressArray [Hartenstein and Kress, 1995], Chess [Marshall et al., 1999], PipeRench [Goldstein et al., 2000], Morphosys [Singh et al., 2000], Montium [Heysters and Smit, 2003], ADRES [Mei et al., 2003a] ou FDR-CGRA [Wan et al., 2009] se classent à ce niveau de granularité. La figure 1.15 donne un exemple schématique d'une tuile de cette catégorie.

À une granularité encore plus importante, la tuile peut pratiquement être vue comme un processeur élémentaire. Les architectures telles que REMARC [Miyamori and Olukotun, 1998], RAW [Taylor et al., 2002], PiCoGa [Campi et al., 2007], RPA [Zhou and Shen, 2007], MORA [Lanuzza et al., 2007] ou PPA [Park et al., 2009] se classent à ce niveau.

L'architecture CRISP [Chen and Chien, 2008] marque une limite avec le niveau inférieur du fait de la taille de ses tuiles. Elles ne sont pas l'équivalent de processeurs élémentaires, mais sont très spécialisées dans le type de calcul qui leur seront demandées (*e.g.* de l'interpolation sur plusieurs entrées). L'architecture DART [David et al., 2002a], qui sera plus amplement décrite dans la section 1.4.3 peut difficilement se classer ici du fait de ses trois niveaux hiérarchiques possédant chacun des caractéristiques propres.

Remarque La notion d'*Expression-Grained Reconfigurable Architecture* (EGRA), introduite dans [Ansaloni et al., 2008], est une extension de celle de CGRA où l'unité de base considérée n'est plus une tuile, mais un ensemble de tuiles regroupées dans un *cluster*.

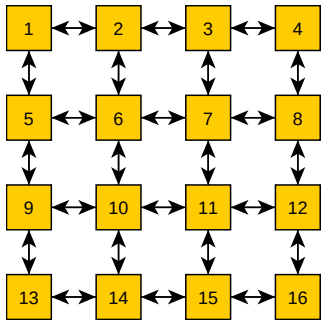
L'homogénéité/hétérogénéité des tuiles

L'homogénéité/l'hétérogénéité des tuiles est une autre caractéristique essentielle d'un CGRA. On peut distinguer trois variantes : homogène, régulière et hétérogène. Une architecture possédant des tuiles homogènes est une architecture pour laquelle toutes les tuiles sont identiques. C'est par exemple le cas d'ADRES, de MORA ou de RPA. Des architectures non homogènes mais régulières comme FDR-CGRA, ou certains modèles d'architecture CGRA comme celui utilisé dans [Park et al., 2008], prennent en compte le fait que toutes les tuiles ne peuvent pas réaliser d'opérations d'accès à la mémoire centrale. Mais celles qui le peuvent sont généralement réparties de manière régulière dans l'architecture. Enfin certaines architectures comme CRISP, RICA ou RaPiD, possèdent des tuiles toutes différentes les unes des autres.

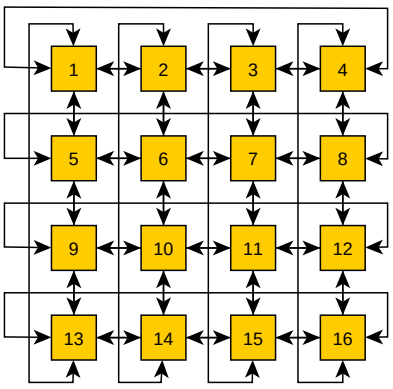
1.4.1.2 Les réseaux d'interconnexion

Dans un CGRA, les tuiles échangent des données au travers d'un réseau d'interconnexion. Ce réseau est généralement point-à-point. La figure 1.16 illustre quatre réseaux parmi les plus utilisés. Dans cette figure, comme dans le reste du manuscrit pour les représentations simplifiées, les tuiles sont représentées par des carrés et l'interconnexion est matérialisée par les arcs. Le réseau de la figure 1.16a est un mesh-2D simple, c'est-à-dire que chaque tuile ne se trouvant pas au bord de l'architecture peut communiquer avec ses quatre voisins les plus proches. De manière à réduire l'irrégularité de cet effet bord, des réseaux toriques ont été proposés (figure 1.16b). Avoir seulement quatre voisins étant parfois contraignant par rapport aux besoins applicatifs, des réseaux deux dimensions à huit voisins ont été proposés. C'est le cas de MORA qui possède un réseau de type mesh-plus (figure 1.16c) ou de Quku [Shukla et al., 2006] qui possède un réseau de type mesh-X (figure 1.16d). Certains CGRAs peuvent avoir différents types de réseaux en fonction des besoins utilisateur. C'est par exemple le cas d'ADRES [Mei et al., 2005].

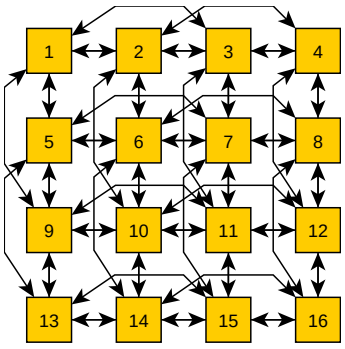
D'autres réseaux d'interconnexion existent, comme le réseau Morphosys qui correspond à un mélange entre un mesh-2D torique et un mesh-plus. Avec un CGRA 4×4 cela signifie que les tuiles peuvent communiquer avec toutes les tuiles de leur ligne et toutes les tuiles de leur colonne. L'utilisation de réseaux d'interconnexion de type *crossbar* est assez rare du fait de son coût de réalisation. Si il est utilisé, c'est pour réaliser des connexions entre un nombre restreint de tuiles, comme dans [Rakossy et al., 2013] où il n'y a que quatre tuiles à faire communiquer avec quatre autres via un *crossbar*. L'utilisation de bus, de bus segmentés ou d'une interconnexion hiérarchisée existe dans la littérature (*e.g.* RaPiD, DART, etc.), mais n'est pas majoritaire.



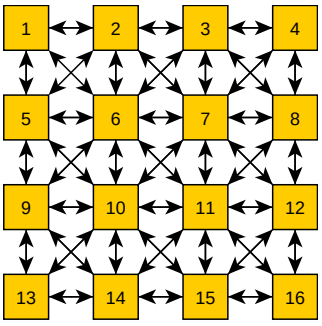
(a) Exemple de mesh-2D simple.



(b) Exemple de mesh-2D torique.



(c) Exemple de mesh-plus.



(d) Exemple de mesh-X.

Figure 1.16 – Exemples d'interconnexions pour un CGRA 4×4.

1.4.1.3 La mémoire

Le positionnement et la quantité de mémoire de l'architecture sont aussi des éléments différenciants. Selon la granularité de la tuile, il est possible qu'il n'y ait pas de registre dans les tuiles (comme dans l'architecture RICA, à l'exception des tuiles de registres), qu'il y ait beaucoup de mémoire (comme dans CRISP pour mémoriser un voisinage de pixel ou dans MORA qui possède dans chaque tuile une mémoire double port de 256×8bits) et/ou qu'il y ait des files de registres partagées. Mais dans un modèle de CGRA classiquement utilisé dans la littérature (celui représenté dans la figure 1.15), chaque tuile possède une RF interne de quelques registres de large et un registre de sortie permettant de communiquer une valeur aux voisins.

1.4.2 Flots de conception

Nombre d'architectures CGRAs ne possèdent pas de flot entièrement automatisé pour projeter des codes applicatifs, ce qui diminue grandement leur utilisabilité. En effet, projeter une application sur CGRA revient à résoudre un problème NP-Complet [Lee et al., 2011]. Cette sous-section présente les approches de la littérature pour projeter des applications sur CGRA.

Modèles utilisés

Dans la littérature, l'application est généralement représentée sous forme de graphe. Ce graphe est généralement un *Data Flow Graph* (DFG), c'est-à-dire un graphe orienté acyclique et bipartite $G(V, E)$ composé de trois éléments :

- les nœuds V_i représentant des opérations (par exemple : $+, -, \times$ etc.) ;
- les nœuds V_j représentant des variables, qui sont des opérandes d'opération et/ou des résultats de calcul ;
- et des arc orientés $E_{i,j}$, représentant les dépendances entre les différents nœuds.

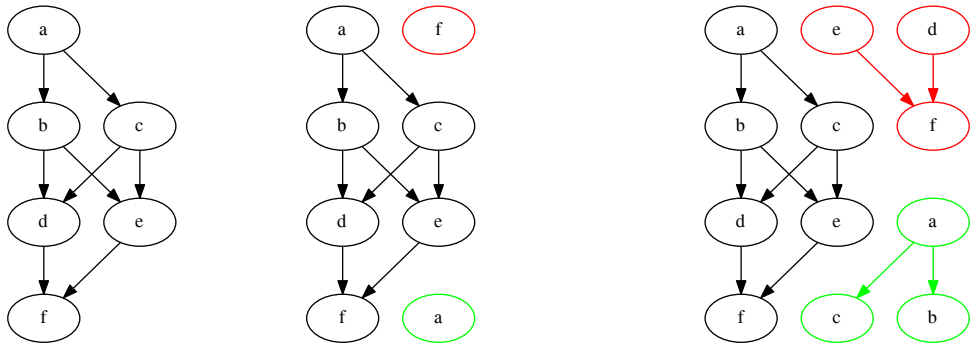
Étant donné que les opérations représentées dans un DFG ne fournissent qu'un seul résultat, les nœuds de variables ne sont parfois pas représentés, comme dans la figure 1.17a. C'est, dans ce cas, directement les nœuds d'opérations qui sont reliés par des arcs qui représentent à la fois la dépendance et les données. Le résultat d'une opération peut être utilisé par plusieurs autres opérations. Les nœuds représentant des opérations non commutatives (*e.g.* la soustraction) doivent avoir un moyen de différencier l'opérande de gauche et de droite. Cela est généralement fait en définissant dans le nœud d'opération des ports d'entrée numérotés. Ce type de représentation est par exemple utilisé dans [Chen and Mitra, 2014, De Sutter et al., 2008, Friedman et al., 2009, Hamzeh et al., 2012, Hamzeh et al., 2013, Lee et al., 2011, Mei et al., 2002, Park et al., 2008].

Les CGRAs ont été initialement créés pour agir comme un co-processeur capable d'accélérer le traitement des parties régulières et pipelinables des codes, c'est-à-dire les cœurs de boucle. La majeure partie des flots existants se concentre donc sur la résolution du problème d'ordonnancement et d'assignation uniquement pour les cœurs de boucle. Pour cela, un intervalle de récurrence ou Intervalle d'initiation (II) est généralement défini, signifiant que le cœur de boucle doit re-débuter tous les II cycles. Cet intervalle permet de calculer les débits de données entrant (équation 1.4.1) et sortant (équation 1.4.2) de l'application :

$$DebitEntrant = \frac{nbDataLues}{Latence \times II} \tag{1.4.1}$$

$$DebitSortant = \frac{nbDataEcrites}{Latence \times II} \tag{1.4.2}$$

avec $nbDataLues$ égal au nombre total de données lues dans le cœur de boucle ; $Latence$ égal au nombre de cycles d'exécution du cœur de boucle et $nbDataEcrites$ égal au nombre de données produites dans le cœur de boucle. Le débit et la latence ne sont liées que si l'exécution du code n'est pas pipelinée. La résolution de ce problème se fait en changeant la représentation



(a) DFG initial. (b) Modulo-DFG avec un II de 3 cycles. (c) Modulo-DFG avec un II de 2 cycles.

Figure 1.17 – Illustration de Modulo-DFGs représenté sur un cœur de boucle complet.

du graphe d'opérations pour généralement utiliser un *Modulo Data Flow Graph* (MDFG), introduit par [Rau, 1994]. La figure 1.17 présente à gauche un DFG simple, au centre le MDFG correspondant avec un II de 3 cycles et à droite le MDFG correspondant avec un II de 2 cycles. Seule une exécution complète du DFG en noir est représentée. La partie en rouge correspond au recouvrement avec l'itération précédente et celle en vert avec l'itération suivante. On remarque que plus l'II diminue et plus le nombre de ressources nécessaires augmente, tout comme le débit, alors que la latence reste inchangée (4 cycles ici).

Concernant la représentation du CGRA, il existe de très nombreuses variantes. Les tuiles sont parfois considérées dans des tableaux de ressources plus ou moins évolués (par exemple la table de réservation de ressources de [Park et al., 2008]). D'autres représentations sont sous forme de graphes comme dans [De Sutter et al., 2008, Friedman et al., 2009, Mei et al., 2002]. La notion de temps/cycles est souvent ajoutée à cette représentation comme dans le *Time-Extended CGRA* (TEC) de [Hamzeh et al., 2012, Hamzeh et al., 2013] (voir figure 1.18c). Projeter l'application sur l'architecture est un problème mathématiquement équivalent à « trouver » dans le graphe du CGRA le graphe de l'application (DFG) comme dans la figure 1.18(d) [Hamzeh et al., 2012]. Cela est possible car les graphes utilisés ont été construits de manière à pouvoir définir des équivalences entre eux. Pour les flots qui ne sont pas rattachés à une architecture particulière, la cible du flot est un CGRA « générique » qui possède un mesh-2D, des tuiles homogènes avec parfois seulement certaines capables d'effectuer des opérations d'accès à la mémoire et une RF dans chaque tuile comme illustré dans la figure 1.19.

Approches

Les flots permettant de projeter des applications sur CGRA peuvent être classés en fonction de leur façon de résoudre d'une part le problème de l'ordonnancement et d'autre part le problème du placement. Le routage, qui correspond à l'étape de vérification de l'existence et de la disponibilité d'un chemin permettant de relier les différents placements d'opérations, étant intimement lié au placement des opérations et des variables, nous ne le considèrerons pas indépendamment du placement dans ce classement. Il existe quatre grandes approches :

— ordonnancement et assignation simultanés par une méthode exacte ;
— ordonnancement et assignation simultanés par une heuristique ;
— ordonnancement et assignation séparés par des heuristiques ;
— ordonnancement et assignation séparés par des méthodes partiellement exactes.

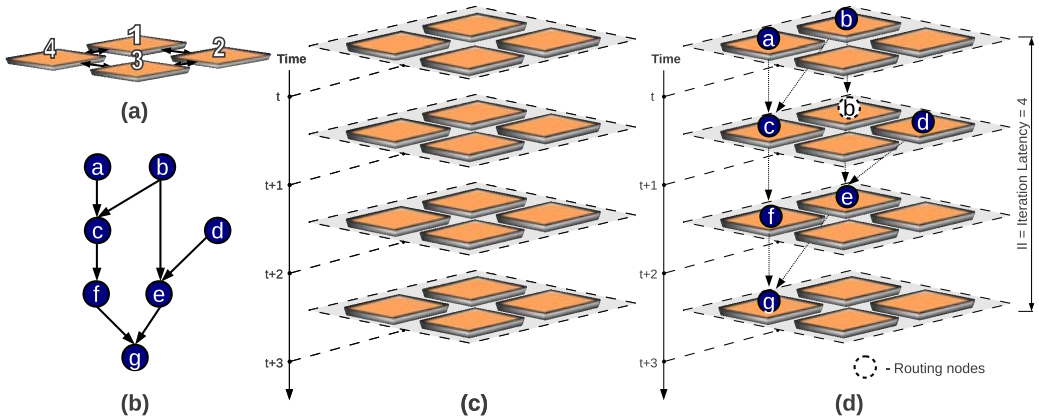


FIGURE 1.18 – Représentation de l'application et de l'architecture proposées dans EPIMap : (a) Un CGRA 2×2, (b) un DFG de six opérations, (c) la représentation TEC, (d) exemple de placement du DFG sur le TEC [Hamzeh et al., 2012].

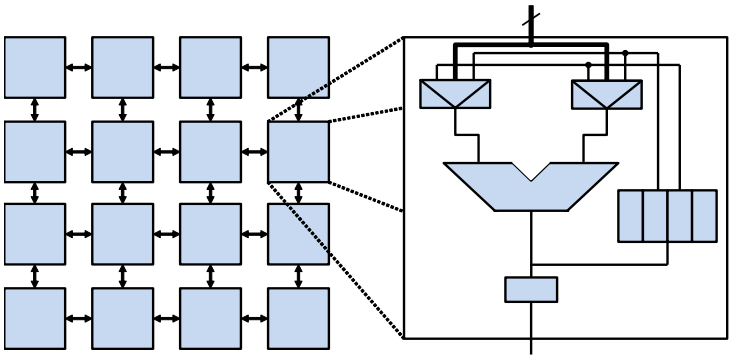


FIGURE 1.19 – Architecture « générique » de CGRA.

La première catégorie correspond à la solution intuitive qui consiste à résoudre le problème dans sa globalité grâce à des méthodes exactes comme de la programmation linéaire en nombre entier ou *Integer Linear Programming* (ILP), mais ces méthodes ne passent pas à l'échelle [Raffin et al., 2010, Brenner et al., 2006].

Constatant cet échec, d'autres flots ont été proposés comme le flot DRESC initialement proposé dans [Mei et al., 2002] et amélioré dans [De Sutter et al., 2008] qui utilise une méthode de recuit simulé (méta-heuristique) pour l'ordonnancement et le placement des opérations. Dans cette méthode, l'algorithme déplace aléatoirement des opérations dans le temps et l'espace et évalue la solution grâce à une fonction de coût. En fonction de l'évaluation et de la température (paramètre interne de l'algorithme décroissant avec le nombre d'itérations), la solution sera conservée ou non pour l'itération suivante. Ce genre de méthodes, bien qu'efficaces, souffre d'un temps de convergence vers une solution généralement long.

La troisième catégorie correspond aux flots qui résolvent les problèmes d'ordonnancement et de placement de façon séparée, c'est-à-dire d'abord l'un, puis l'autre et en utilisant des heuristiques ou des méta-heuristiques comme un algorithme génétique [Lee et al., 2011], des heuristiques adaptées de la recherche sur la synthèse pour FPGA [Friedman et al., 2009] ou d'autres heuristiques comme celle de [Park et al., 2008] qui n'est pas centrée sur les nœuds comme les autres, mais sur les arcs. Cependant, l'ordonnancement et l'assignation sont des problèmes interdépendants et les résoudre de manière séparée n'est pas optimum comme cela sera illustré au chapitre 2.

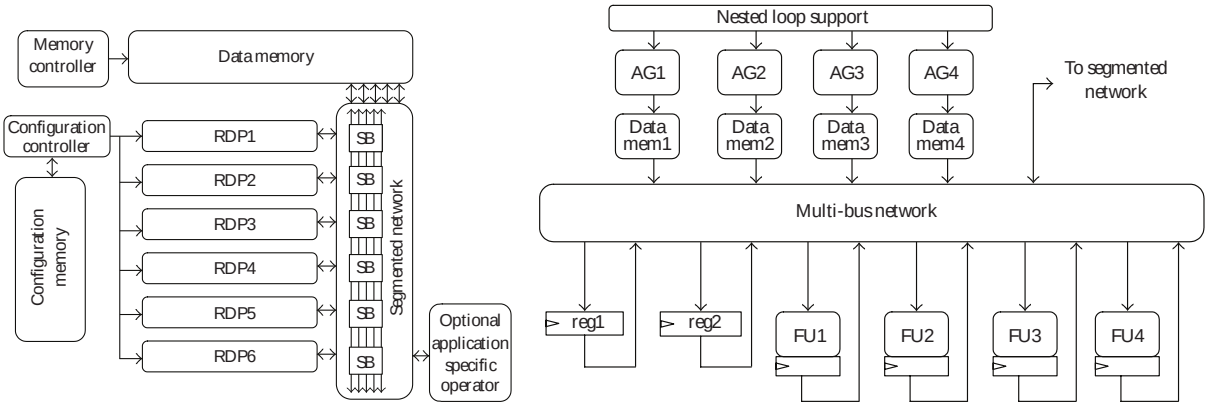
La quatrième famille d'approches résout de façon séparée les deux problèmes, mais en utilisant à la fois une heuristique d'ordonnancement et une méthode exacte de placement/routage. Contrairement à ce qui est classiquement fait, dans [Hamzeh et al., 2012, Hamzeh et al., 2013], l'ordonnancement n'est pas réalisé par un list-scheduling [Rau, 1994], mais par une heuristique essayant d'adapter le graphe d'application au graphe de l'architecture grâce à des transformations formelles. Ensuite, le flot tente de résoudre le problème qui consiste à trouver le graphe de l'application dans celui de l'architecture par résolution exacte du problème du sous-graphe commun maximal en utilisant l'algorithme de Levi [Levi, 1973]. Cet algorithme étant exhaustif, si après un temps de recherche défini par l'utilisateur, aucune solution n'est trouvée, il est interrompu. La flot augmente alors l'II, relance l'heuristique d'ordonnancement, recrée le modulo-DFG et tente à nouveau de trouver le graphe de l'application dans le graphe de l'architecture. Les transformations proposées sont le déplacement dans le temps grâce à l'introduction de nœuds de routage et la duplication de calculs qui permet de réduire le nombre de successeurs d'un nœud particulier pour faciliter son placement et/ou répondre à une contrainte architecturale du nombre maximal de voisins auxquels une tuile peut communiquer un résultat.

Remarque

Les flots de conception/compilation sur CGRA se distinguent de ceux ciblant des architectures à encore plus gros grains, telles que les *many-cores*, par le fait que ce sont les opérations qui sont ordonnancées sur les tuiles et non pas des tâches ou des programmes complets. C'est essentiellement ce point qui permet de classer une architecture à très gros grains comme étant un CGRA ou une architecture *many-core*.

1.4.3 Description de certaines architectures CGRA

Dans cette sous-section, trois architectures CGRA sont présentées plus en détails. La première est l'architecture DART qui a fait l'objet de différentes implémentations de méthodes de tolérance aux fautes. Elle est dans un premier temps décrite et dans la section suivante les méthodes mises en œuvres seront présentées. Les deux autres architectures sont en partie ou totalement capables d'exécuter un code complet (et pas simplement des cœurs de boucles).



(a) Schéma d'un cluster. (b) Schéma d'un RDP.

FIGURE 1.20 – Schéma d'un cluster et d'un RDP de DART [Pillement et al., 2008].

DART

DART [David et al., 2002a, David et al., 2002c, David et al., 2002b, Pillement et al., 2008] est une architecture CGRA particulière car elle dispose de trois niveaux hiérarchiques. Le premier niveau est composé de quatre clusters, d'un contrôleur de tâches, de la mémoire de données et de configuration et des interfaces entrées/sorties. Chaque cluster, comme illustré en figure 1.20a possède son contrôleur, une mémoire de données, un composant d'Accès Direct à la Mémoire (DMA) et sept chemins de données dont un reconfigurable à grains fins pour réaliser des opérateurs spécifiques (*Optionnal application specific operator* sur le schéma). Ceux reconfigurables à gros grains sont appelés RDPs pour Reconfigurable DataPaths. Ils sont connectés à un bus segmenté ce qui leur permet de communiquer entre eux et d'avoir accès à la mémoire centrale. Chaque RDP contient quatre unités fonctionnelles (FUi sur le schéma) de deux types : le premier type est capable d'effectuer des multiplications ou des additions ainsi que du décalage de bits ; le second type est composé d'une ALU, capable donc d'effectuer des additions, des soustractions, les opérations logiques ET, OU et NON ainsi que donner la valeur absolue. Chaque FU est couplée à un registre permettant de mémoriser le résultat. Les RDPs contiennent aussi deux registres connectés directement sur le réseau multibus qui relie les différents éléments. Quatre unités de gestion d'adresses prennent en charge la génération d'adresses pour les données. Elles se composent d'un petit processeur type *Reduced Instruction-Set Computer* (RISC) permettant de calculer les motifs classiques d'accès à la mémoire de données présente dans le RDP comme le modulo ou le pré-/post-incrément, de sorte que la partie calculatoire soit toujours alimentée et n'ait pas besoin d'effectuer ces calculs d'adresses. L'objectif étant d'avoir une architecture efficace en termes de consommation d'énergie, les RDP non-utilisés peuvent être éteints par le gestionnaire de configuration.

Le flot d'implémentation de DART est donné dans la figure 1.21 et se compose de cinq parties : SUIF, cDART, gDART, ACG et scDART. Ces entrées sont une description de l'architecture et un code C. Le code C va être compilé et éventuellement transformé pour augmenter son parallélisme. L'outil SUIF va alors séparer le code entre les parties régulières et irrégulières. Pour les parties régulières, gDART génère les chemins de données dans les RDPs de façon à correspondre au mieux aux cœurs de boucle. Pour les parties irrégulières, c'est un flot de compilation classique, cDART, qui est utilisé en considérant que les chemins de données sont fixes dans les RDPs et que seule l'opération réalisée peut changer (*e.g.* addition vers soustraction). Ce changement d'opération peut se faire à chaque cycle, contrairement au changement de chemin de données qui nécessite plusieurs cycles avant de pouvoir continuer l'exécution. En parallèle, l'ACG génère le code des unités de gestion d'adresses pour les RDPs utilisés. scDART permet de simuler le système complet et d'analyser les performances pour éventuellement modifier certains choix d'allocation.

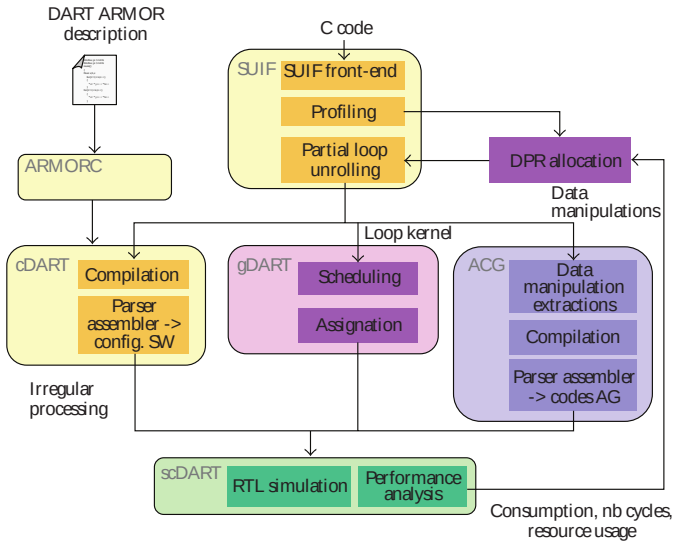


Figure 1.21 – Flot d'implémentation de DART [Pillement et al., 2008].

ADRES

Contrairement aux autres architectures CGRAs classiques, dans ADRES [Mei et al., 2003a, Mei et al., 2005], le processeur hôte possède un couplage très fort avec la grille de tuiles en partageant une file de registres et une partie des tuiles comme illustré dans la figure 1.22. ADRES peut donc être vu comme un processeur classique à « mot d'instruction très long » ou *Very Long Instruction Word* (VLIW) possédant six unités fonctionnelles en parallèle ou bien comme une grille de tuiles avec un réseau d'interconnexion de type mesh-plus. L'intérieur d'une tuile est très classique : des multiplexeurs d'entrée, un opérateur, un registre de sortie et une RF interne. De manière à répondre au mieux aux besoins, les concepteurs ont créé un outil permettant d'analyser un ensemble de codes cibles pour optimiser l'architecture pour cet ensemble d'applications (*e.g.* la taille de RF, le réseau d'interconnexion, etc.) [Mei et al., 2005]. La figure 1.23 illustre les liens au niveau du flot qui existent entre la partie processeur et la partie matrice reconfigurable. Le code d'entrée, du C, est séparé entre les cœurs de boucles et le reste du code. Pour les cœurs de boucle, c'est l'intégralité du CGRA qui est considéré. L'outil utilisé est DRESC [Mei et al., 2002, Mei et al., 2003b, De Sutter et al., 2008] développé pour ADRES, mais qui reste une des références de l'état de l'art sur des architectures « génériques ». Il utilise un recuit simulé global pour ordonnancer et placer les opérations sur l'architecture dont l'inconvénient est le temps important de convergence. Pour le reste du code, c'est un flot exact utilisant de la programmation linéaire en nombre entier (ILP) qui est utilisé. Dans ce cas particulier, son temps ne diverge pas car il y a un nombre très limité de ressources dans la « vue VLIW » qui ne communiquent entre elles qu'au travers d'une RF partagée.

Pendant le fonctionnement, la partie CGRA à proprement parler est configurée pour un cœur de boucle particulier. En cas de changement de cœur de boucle ou de passage dans une partie irrégulière, le CGRA doit être reconfiguré. Pour le reste du code, c'est seulement une sous-partie qui exécute les instructions à la manière d'un processeur généraliste remplaçant le processeur classiquement utilisé à côté du CGRA. Le lien entre les deux se fait par les ressources appartenant à la fois à la vue VLIW et à la vue CGRA. La figure 1.23b illustre un lien possible entre ces deux parties : des variables initialisées par la vue VLIW et stockées dans la RF sont utilisées pour lire des données en mémoire, effectuer des calculs, écrire les résultat dans la mémoire et mettre à jour une des variables de la RF.

ADRES possède une architecture intéressante dans le cadre de la tolérance aux fautes de par la granularité des tuiles, son flot unifié et donne une première idée d'approche pour porter des codes complets sur CGRA.

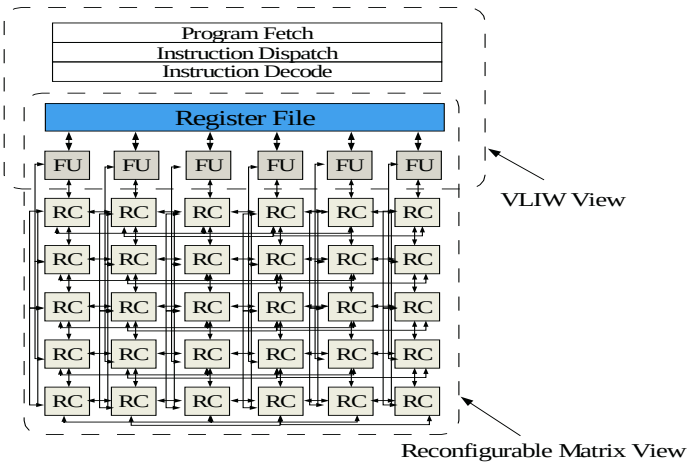
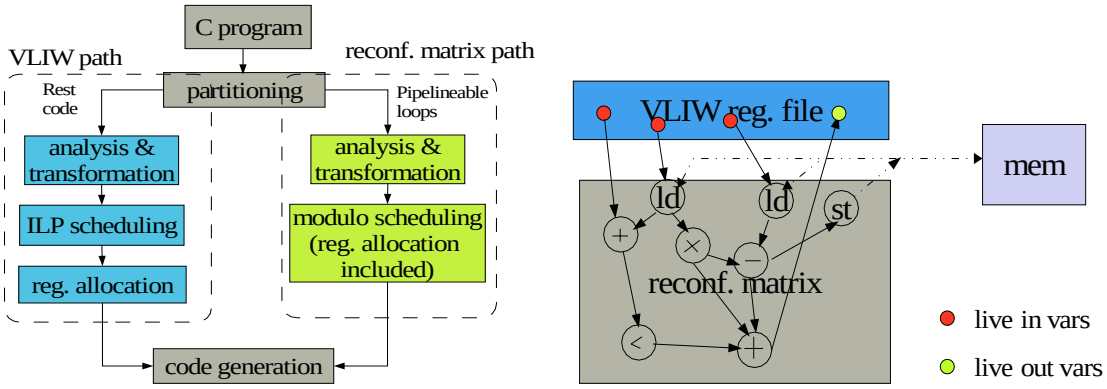


FIGURE 1.22 – Architecture d'ADRES [Mei et al., 2003a] avec les deux vues (VLIW et matrice reconfigurable).



(a) Modèle de programmation. (b) Modèle d'exécution.

FIGURE 1.23 – Flot de compilation d'ADRES et interface entre la partie VLIW et la matrice reconfigurable [Mei et al., 2003a].

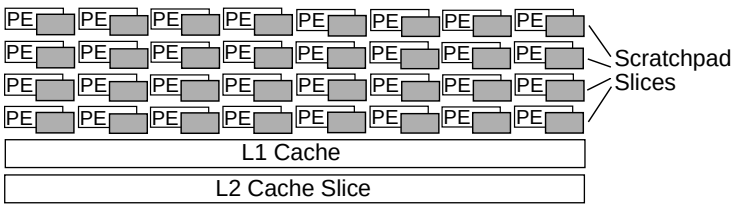


FIGURE 1.24 – Schéma de l'architecture à « instructions déclenchées » [Parashar et al., 2013].

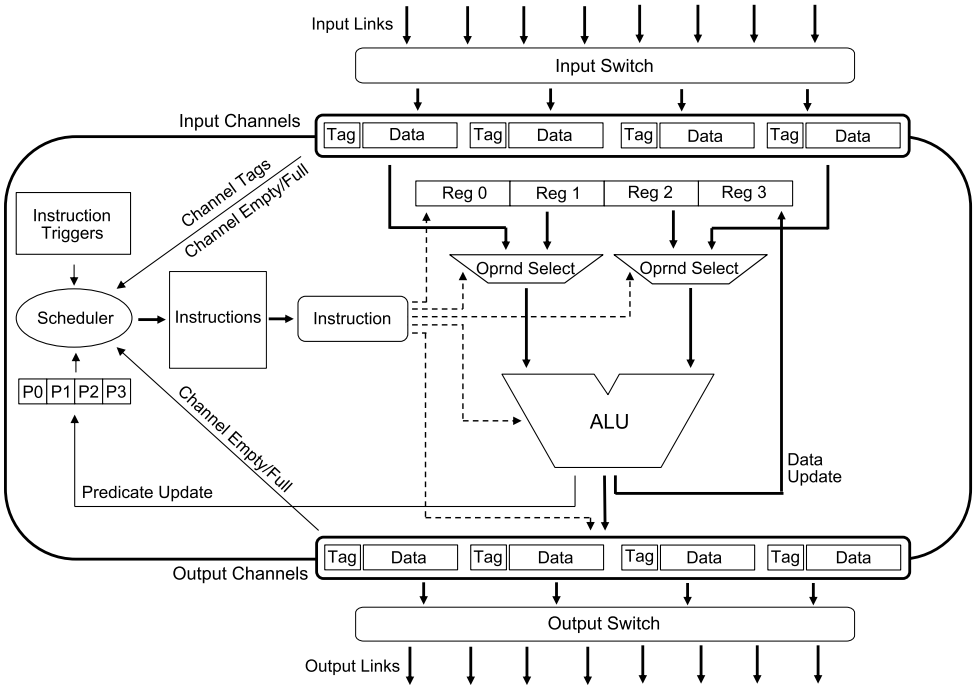


Figure 1.25 – Intérieur d'une tuile de *Triggered Instruction* [Parashar et al., 2013].

Triggered Instruction

Récemment, une nouvelle approche a été proposée [Parashar et al., 2013]. Elle propose un nouveau paradigme de programmation et d'exécution pour des architectures CGRAs. Dans cette architecture, illustrée en figure 1.24, les tuiles sont indépendantes dans l'exécution des instructions et il n'y a plus de flot de contrôle à proprement parler. Ce dernier est remplacé par des prédicats pour chacune des instructions. L'architecture est dite à « instructions déclenchées ». La structure générale de la tuile est donnée dans la figure 1.25. Elle comporte une partie exécution traditionnelle avec des registres en local ainsi qu'une partie opérative (ALU). Chaque tuile peut recevoir des données de la part de ses voisins en fonction du réseau d'interconnexion. La particularité de cette architecture vient de la partie contrôle et de ses implications sur le reste de la tuile. Dans cette architecture, chaque tuile possède une mémoire d'instructions. Un prédicat est associé à chaque instruction. C'est ce prédicat qui permet de savoir si l'instruction peut être exécutée à l'instant courant. Les instructions sont donc exécutées sans ordre prédéterminé (« *out of order* ») et les prédicats permettent de s'assurer des dépendances de données. Dans le cas où plusieurs instructions peuvent être exécutées, un ordonnanceur élémentaire (composé de portes logiques « ET ») décide quelle est l'instruction la plus prioritaire (voir figure 1.26). Les résultats/données possèdent tous un tag qui permet à l'unité de contrôle de la tuile de savoir si le prédicat correspondant est vérifié. L'exécution aléatoire des instructions impose à cette architecture de posséder une RF en entrée et en sortie de chaque tuile afin de mémoriser les données que la tuile va potentiellement utiliser.

Cette approche est orthogonale à toutes les autres de l'état de l'art pour projeter des applications sur CGRA. Elle pose néanmoins quelques interrogations sur la méthode de détermination/création des prédicats ainsi que du chargement correct des instructions dans la mémoire d'instructions des tuiles. En effet, si il n'y a pas d'instruction capable de s'exécuter dans la tuile, elle ne fera rien en attendant que des données utilisables soient produites par d'autres tuiles. De même, il faut que les données produites par une tuile puissent être utilisées par la tuile qui possède l'instruction adéquate, ceci pouvant éventuellement conduire à un inter-blocage complet.

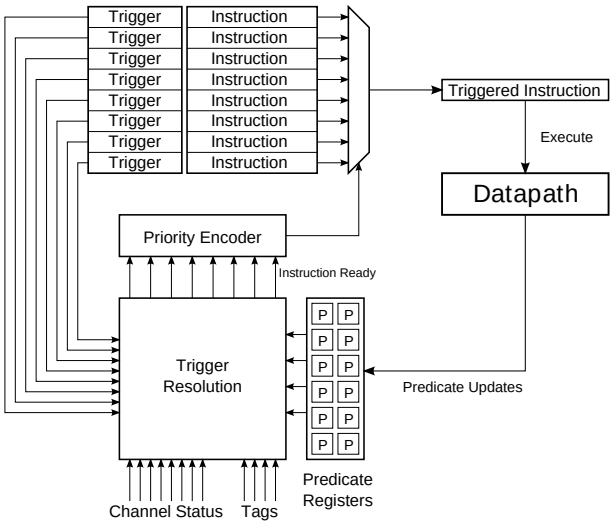


FIGURE 1.26 – Fonctionnement ordonnanceur de *Triggered Instruction* [Parashar et al., 2013].

1.4.4 Méthodes pour la tolérance aux fautes

La littérature propose quelques exemples de méthodes de tolérance aux fautes sur CGRAs. Ces méthodes, brièvement introduites dans la section précédente, sont détaillées et analysées ici.

Tolérance aux fautes transitoires

L'article [Singh et al., 2006] présente pour le CGRA RAW une implémentation de la tolérance aux fautes transitoires qui ne change pas son architecture. Elle s'appuie sur l'utilisation de la triplication temporelle, de la duplication sélective, de la réplication sélective et de points de contrôle (*checkpoints*). La réplication sélective permet de ne pas dupliquer toutes les parties du code et donc d'accélérer son exécution pour des parties moins/non sensibles. La duplication permet de s'assurer qu'une partie de l'exécution entre deux points de contrôle donne bien le même résultat que l'autre version exécutée en parallèle. Si les deux parties dupliquées ne fournissent pas les mêmes résultats, alors la portion de code considérée sera ré-exécutée. La triplication temporelle est utilisée pour les entrées/sorties. Elle permet de s'assurer que les données qui seront traitées sont bien correctes et que les résultats fournis le sont aussi. Ce type de mécanisme n'est pas adapté dans le contexte de cette thèse car il n'est pas possible de garantir le temps de traitement d'une donnée dans l'architecture.

Les travaux autour du CGRA DART possèdent trois extensions traitant d'implémentations de la tolérance aux fautes. La méthode proposée dans [Jafri et al., 2010, Jafri et al., 2013] permet de détecter les fautes transitoires en ligne. Pour cela, elle vérifie l'exactitude des calculs en utilisant une technique de détection d'erreur par le calcul du reste « modulo 3 » [3]. Pour cela, en parallèle du chemin de données classique, des ressources dédiées calculent la valeur attendue du modulo en se basant sur sa valeur précédente. Des opérateurs dédiés sont utilisés pour faire ces calculs (*e.g.* additionneur modulo 3, soustracteur modulo 3, etc.). Les valeurs des modulo 3 sont stockées en mémoire à côté des données correspondantes. À la fin de chaque étage de calcul, la valeur du modulo du résultat est calculée. Elle est alors comparée avec la valeur attendue qui a été calculée en parallèle du résultat. Si elles sont identiques, c'est qu'il ne s'est pas produit d'erreur et l'exécution continue. Si elles ne le sont pas, alors l'étage d'opérations correspondant est ré-exécuté. Dans l'architecture DART, cette technique utilise moins de ressources que la

3. La valeur d'un nombre m modulo n est le reste de la division euclidienne de m par n. Par exemple 5 modulo 3 donne 2.

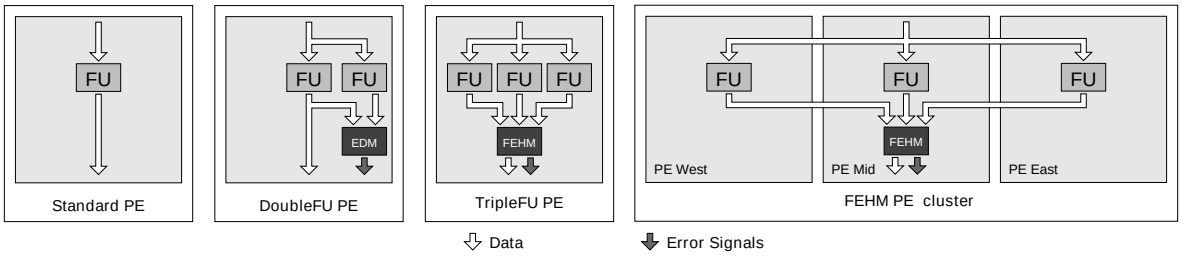


Figure 1.27 – Schéma d'un PE standard et trois implémentations de redondance : duplication, triplication et triplication bas-coût de [Schweizer et al., 2011].

duplication simple et possède une plus grande efficacité énergétique. Cependant, elle introduit un délai non négligeable pouvant limiter la fréquence de fonctionnement.

Dans [Azeem et al., 2011] une méthode pour optimiser la ré-exécution des instructions, indépendante de la méthode de détection et donc compatible avec la précédente, est présentée. Elle définit un contrôleur d'erreur qui va permettre une ré-exécution rapide de l'instruction précédente en mémorisant les valeurs des données du cycle courant et du cycle précédent. La durée d'une faute transitoire est considérée comme inconnue et donc le nombre de répétitions de l'instruction précédente en cas d'erreur peut être supérieur à un. Un seuil de répétition est fixé en fonction de la technologie de fabrication et du type d'erreur attendu. Ce seuil permet de considérer que la faute est permanente et lance la reconfiguration du *Reconfigurable Data Path* (RDP) ainsi que la ré-exécution complète de la tâche.

Dans [Schweizer et al., 2011], plusieurs implémentations de redondance pour la tolérance aux fautes transitoires pour des architectures CGRAs « génériques » sont présentées. La figure 1.27 en présente trois variantes. Le schéma de gauche représente de façon simplifiée une tuile standard non tolérante aux fautes transitoires. Elle est reliée aux autres tuiles par le réseau d'interconnexion. La méthode de duplication dans la tuile est illustrée dans la tuile « DoubleFU PE ». Un module de détection d'erreur ou *Error Detection Module* (EDM) est utilisé pour comparer le résultat des deux voies et déclencher un signal d'erreur quand ils sont différents. Une implémentation de la triplication dans la tuile est schématisée dans la tuile « TripleFU PE ». Les trois parties opératives fournissent leurs résultats à un module de détection et de correction d'erreur appelé *Flexible Error Handling Module* (FEHM). Ce module masque la faute et permet de connaître la provenance de l'erreur. L'implémentation proposée dans cet article consiste à réaliser la triplication non pas en triplant les ressources dans chaque tuile, mais en utilisant les opérateurs des tuiles voisines et le même module de correction d'erreur que dans le cas précédent. Faire ainsi permet de faire coexister deux modes de fonctionnement : le mode tripliqué et le mode normal. Comme l'illustre le schéma de droite de la figure 1.27, seule une tuile sur trois possède le module de correction d'erreur, réduisant le surcoût en cas de non utilisation de la triplication. L'inconvénient de cette approche est la complexité de l'obtention de *mappings* sur l'architecture en mode tripliqué du fait du réseau d'interconnexion.

Tolérance aux fautes permanentes

L'approche proposée dans [Eisenhardt et al., 2011] définit une méthode de tolérance aux fautes permanentes pour les CGRAs. Le flot de la figure 1.28 décrit l'ensemble des étapes réalisées en cas de d'apparition d'une faute permanente. Il se compose des étapes suivantes :

— à l'apparition d'une faute un voisinage de 3×3 est défini ;
— l'ensemble des opérations assignées dans ce voisinage est alors extrait ;
— il essaie de déplacer les opérations *mappées* sur la tuile fautive dans le voisinage ;
— s'il n'y arrive pas, le voisinage est augmenté à 5×5, puis à 7×7 ;
— s'il n'y a toujours pas de solution alors l'opération problématique sera placée dans un

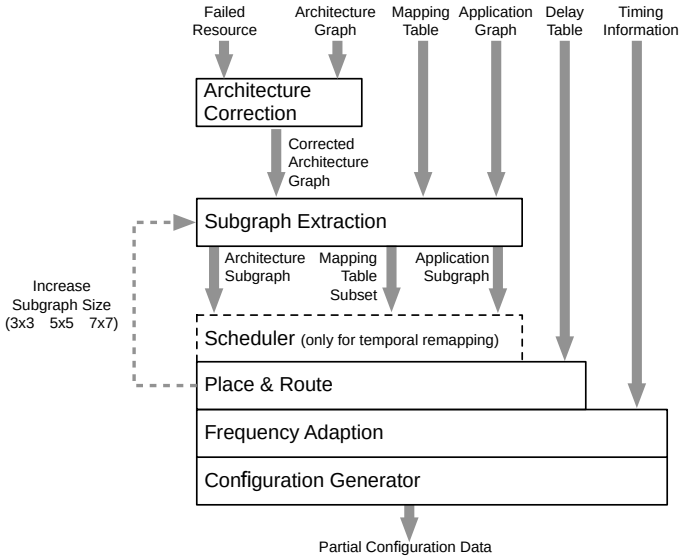


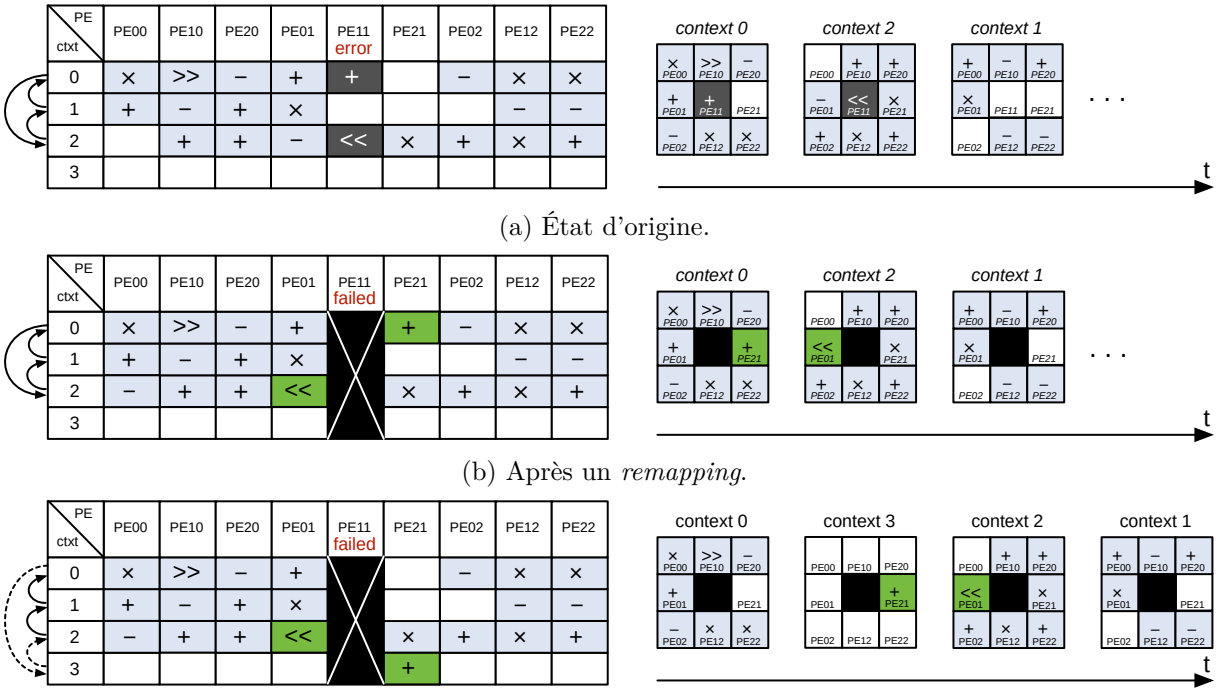
FIGURE 1.28 – Flot pour la tolérance aux fautes permanentes de [Eisenhardt et al., 2011].

nouveau cycle d'exécution (ou contexte) intermédiaire ;
— le processus de placement-routage est alors relancé avec ce nouvel ordonnancement uniquement pour les cycles précédents et suivants au contexte ajouté, permettant ainsi de respecter les dépendances de données.

La figure 1.29a schématise le problème à résoudre avec la tuile 11 fautive. Dans la figure 1.29b, l'opération d'addition du contexte 0 a été décalée dans la tuile 21 et le décalage de bits du contexte 2 a remplacé la soustraction de la tuile 01 qui s'est déplacée dans la tuile 00. La figure 1.29c illustre le cas où le déplacement de l'addition ne serait pas possible à cause des dépendances de données créant un nouveau contexte avec cette opération. L'ordre d'exécution n'est alors plus 0-2-1-0…, mais 0-3-2-1-0… Seulement, le temps de reconfiguration peut être rédhibitoire pour des applications temps réel car la compilation sur CGRA est très complexe. Le système ne fonctionnera plus durant cette reconfiguration, ce qui rend cette méthode non utilisable dans notre contexte. De plus, l'exploration se limite à des voisinages restreints ne permettant pas, par exemple, de simplement déplacer l'intégralité des calculs dans une direction.

Dans [Wang et al., 2011], cette idée de déplacer les calculs sur des lignes ou des colonnes est proposée. Elle transpose à un modèle d'architecture CGRA des techniques existantes sur FPGA comme [Huang and Mccluskey, 2001]. Il est ainsi possible de profiter de ressources libres dans l'intégralité du CGRA comme illustré sur la figure 1.30. Le flot permettant d'obtenir ce résultat reste cependant assez vague et l'approche décrite dans cet article ne permet pas de tolérer plusieurs fautes permanentes les unes à côté des autres, limitant son intérêt, de par le choix de l'interconnexion (un mesh-X). Avec un réseau d'interconnexion plus riche, la méthode permettrait de durer plus longtemps, mais complexifierait aussi le processus de reconfiguration.

Dans [Rakossy et al., 2013] une approche fondamentalement différente est proposée. L'idée est d'effectuer de l'autotest cyclique sur l'architecture en prévoyant dès le départ des ressources libres supplémentaires pour pouvoir l'effectuer en parallèle. L'architecture CGRA considérée est constituée d'un ensemble de lignes contenant des tuiles reliées aux lignes supérieure et inférieure par un *crossbar*, c'est-à-dire un réseau point-à-point complet. La figure 1.31a donne un exemple d'une telle architecture avec quatre tuiles par ligne et pour lequel le réseau d'interconnexion est programmé pour réaliser un papillon de *Fast Fourier Transform* (FFT). L'approche ajoute, en plus de l'interconnexion *crossbar*, une ligne au dessus et en dessous des tuiles avec des *switch boxes* permettant de changer la destination d'un lien vers une autre tuile. Les *switch boxes* ont



(a) État d'origine.

(b) Après un *remapping*.

(c) Après un réordonnancement partiel et un *remapping*.

Figure 1.29 – Illustration des deux moyens disponibles pour tolérer les fautes permanentes de [Eisenhardt et al., 2011].

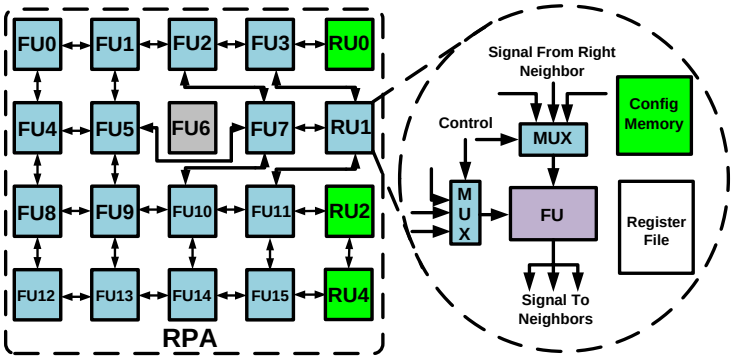
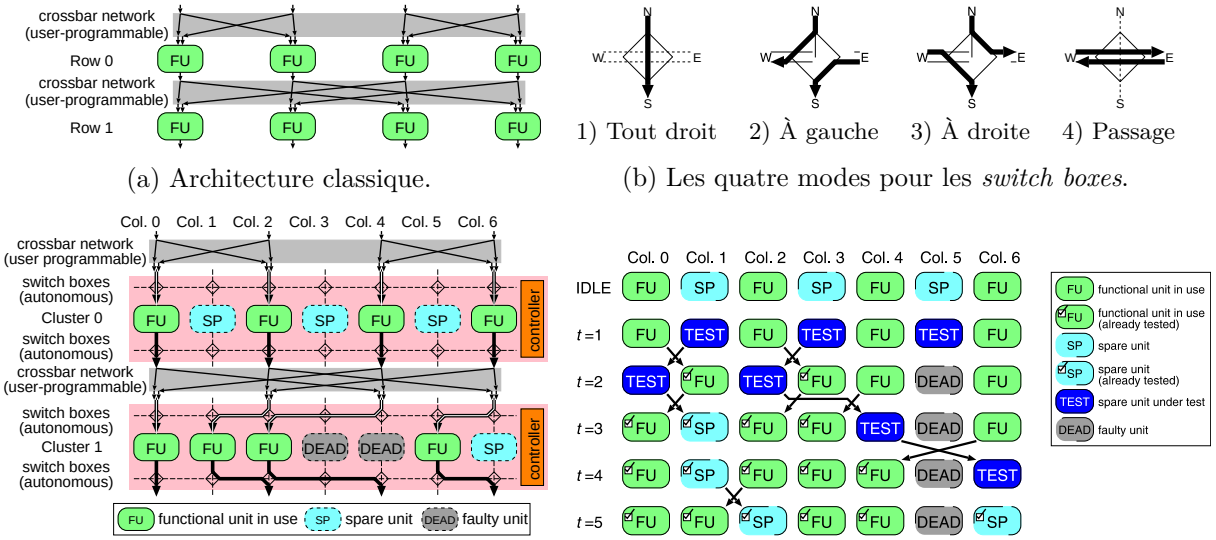


Figure 1.30 – Schéma de principe de la tolérance proposée dans Elastic CGRA [Wang et al., 2011].



(a) Architecture classique.

(b) Les quatre modes pour les *switch boxes*.

(c) Architecture proposée.

(d) Exemple de fonctionnement.

FIGURE 1.31 – Méthode de « *Hot Swapping* » pour la tolérance aux fautes permanentes proposée dans [Rakossy et al., 2013].

quatre états possibles illustrés en figure 1.31b. Le premier, l'état par défaut, permet de relier effectivement la sortie du *crossbar* avec la tuile correspondante. Les deux suivants permettent de changer la destination de la sortie du *crossbar* vers la gauche ou vers la droite. Le dernier permet de ne pas relier la tuile sous-jacente et de faire passer une donnée provenant de droite à gauche ou inversement. En plus de ces *switch boxes*, l'approche ajoute des tuiles supplémentaires à la ligne. L'architecture résultante pour trois tuiles ajoutées est donnée en figure 1.31c. L'intérêt des *switch boxes* est de pouvoir diriger les données vers des tuiles qui fonctionnent. Dans la première ligne de tuiles (Cluster 0), les quatre *switch boxes* sont dans leur état par défaut car les tuiles sont toutes fonctionnelles. Dans la seconde ligne (Cluster 1), les tuiles des colonnes 3 et 4 sont défectueuses. Les *switch boxes* permettent alors de diriger les sorties des colonnes 2 et 4 du *crossbar* vers les tuiles des colonnes 1 et 2 et ainsi ne pas utiliser les tuiles défectueuses. Lors de la phase de compilation et d'assignation, les tuiles supplémentaires ne sont pas considérées comme étant disponibles pour effectuer des calculs. Dans l'exemple, il n'y a que quatre tuiles disponibles par ligne. Durant l'exécution, les tuiles supplémentaires permettent d'exécuter de l'autotest de tuiles, comme illustré dans la figure 1.31d. Au premier cycle, $t = 1$ sur la figure, les tuiles utilisées pour effectuer les calculs sont les tuiles des colonnes 0, 2, 4 et 6. Les tuiles 1, 3 et 5 effectuent de l'autotest. Au cycle suivant, ce sont les tuiles des colonnes 0 et 2 qui effectuent de l'autotest. La tuile de la colonne 5 a été détectée comme étant défectueuse et ne sera plus utilisée. Au troisième cycle, la tuile de la colonne 4 est testée, etc. Lorsque toutes les tuiles ont été testées, l'ordonnanceur d'autotest attend un nombre de cycles défini par l'utilisateur avant de relancer l'autotest. L'implémentation de cette approche possède néanmoins trois limitations majeures :

— dans la figure 1.31d, au troisième cycle, il n'est pas possible d'effectuer le test de la tuile 6 en même temps que celui de la tuile 4 car pour cela il faudrait que la *switch box* de la colonne 4 puisse diriger ses données vers la gauche et en même temps faire passer les données provenant de sa droite (de la colonne 6) à sa gauche ;
— la réactivité de ce type de test est limitée et dépend de la fréquence de test ;
— dans cette architecture, si une ligne possède trois tuiles défaillantes, alors il n'est plus possible de garantir les résultats de l'intégralité du composant.

1.5 Conclusion

L'étude de l'état de l'art de la tolérance aux fautes, en connaissance des contraintes applicatives fixées, nous a amené à choisir d'une part des familles de méthodes tolérance aux fautes transitoires et permanentes et d'autre part un type d'architecture qui semble le plus à même d'implémenter efficacement ces méthodes tout en respectant nos contraintes. Concernant la méthode de tolérance aux fautes transitoires la littérature justifie l'utilisation du masquage de faute pour les parties calculatoires ainsi que la mémorisation des valeurs en registres. La possibilité d'utiliser des codes correcteurs d'erreurs n'est pas écartée pour autant et fera l'objet d'une discussion ultérieure. Pour les fautes permanentes, l'étude de l'état de l'art montre que la reconfiguration dynamique est la seule méthode permettant de ne plus utiliser une ressource défectueuse. L'étude des architectures reconfigurables à gros grains, en particulier de leur flot d'implémentation et des éventuelles implémentations de mécanismes de tolérance nous a permis de voir qu'aucune ne répond à notre besoin.

Le chapitre suivant détaille notre proposition de méthode pour répondre à la problématique de la réalisation d'un système numérique tolérant aux fautes en justifiant nos choix. Les chapitres 3, 4 et 5 décrivent chacun une partie précise de la proposition globale.

Chapitre 2

Motivations et proposition générale

2.1 Introduction

La problématique générale qui est de déterminer comment réaliser un système numérique tolérant aux fautes est extrêmement vaste. Ce chapitre a pour but de justifier les choix qui ont orienté les travaux présentés dans ce manuscrit. L'analyse de l'état de l'art de la tolérance aux fautes a permis de choisir un type d'architecture qui est la plus à même de répondre aux exigences de notre contexte : un CGRA utilisant de la triplication et éventuellement des codes correcteurs d'erreurs. Vouloir être tolérant aux fautes permanentes impose d'être capable de modifier l'utilisation des ressources de l'architecture par le code.

Dans ce chapitre, nous présenterons le modèle d'application que l'on souhaite utiliser et dont le formalisme est nécessaire pour projeter l'application sur l'architecture, permettant ainsi d'obtenir des *mappings*. Nous détaillerons ensuite les problématiques que soulève l'utilisation d'un CGRA dans un contexte de tolérance aux fautes. Puis nous motiverons le choix d'obtenir des *mappings* utilisant des ressources différentes. Enfin nous aborderons les problématiques dépendantes de la méthode de projection d'une application sur une architecture CGRA.

2.2 Modèle d'application

Le modèle *Control and Data Flow Graph* (CDFG) d'une application est un modèle qui représente la combinaison de la partie flot de contrôle de l'application ou *Control Flow Graph* (CFG) et de la partie calculatoire à proprement parler. La partie calculatoire se présente sous la forme d'un ensemble de *Basic Block*s (BBs). Un BB est représentable par un DFGs. Le passage d'un BB à l'autre se fait en fonction du flot de contrôle. Ce modèle permet de représenter des applications plus complexes que celles représentées par un DFG. En effet, dans un DFG, l'intégralité du contrôle est « mis à plat » : les boucles sont déroulées permettant ainsi d'expliciter les valeurs d'indices ; les conditions de type « *if . . . then . . . else* » sont transformées de manière à exécuter à la fois le *then* et le *else*, puis une instruction particulière PHI sélectionne le bon résultat pour la suite de l'exécution en fonction du résultat du test défini dans le *if* ; il n'est pas possible d'avoir du contrôle dynamique, c'est-à-dire du contrôle dépendant d'une valeur lue.

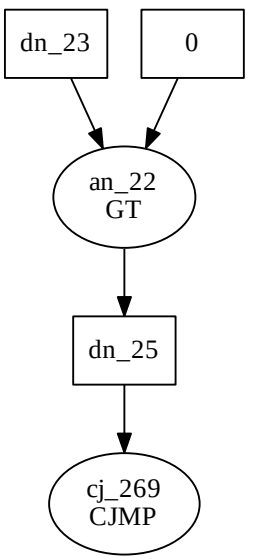
Dans un CDFG, chaque BB se termine par une instruction de saut qui définit le prochain BB à exécuter. Ce saut peut selon les cas être inconditionnel ou conditionnel. Un saut inconditionnel représente un passage forcé d'un BB à un autre. Pour un saut conditionnel, le choix du BB suivant dépend du résultat d'un calcul, généralement un test d'égalité ou d'inégalité.

Par exemple, pour représenter la boucle *for* simple dont le code est donné dans l'algorithme 1, sans optimisation particulière, il faut trois BBs présentés dans la figure 2.1 :

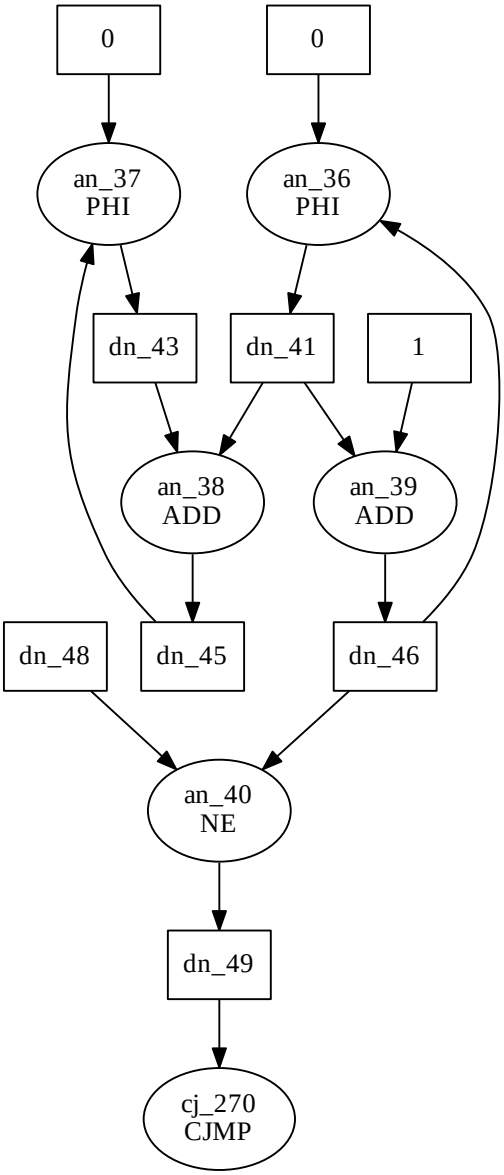
— Le premier (bb_35, figure 2.1(a)) contient les instructions qui permettent de savoir si il faut rentrer dans la boucle. Pour cela le code teste la condition $n > 0$ (n est ici représenté

Algorithme 1 Exemple de code avec une boucle *for* simple

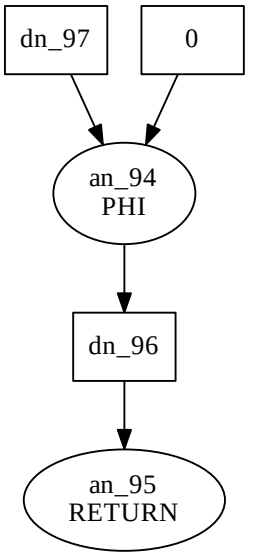
```
1  int boucleForSimple (int n)
2  {
3    int i;
4    int res = 0;              // Variable de resultat
5    for (i = 0; i < n; ++i)
6    {
7      res += i;               // Coeur de boucle
8    }
9    return(res);
10 }
```



(a) Basic Block bb_35.



(b) Basic Block bb_93.



(c) Basic Block bb_116.

FIGURE 2.1 – CDFG de l'algorithme 1.

par le nœud dn_23). Il se termine par un saut conditionnel vers bb_93 si le test est vrai et vers bb_116 si le test est faux.
— Le deuxième BB (bb_93, figure 2.1(b)) contient les instructions exécutées par le cœur de boucle à proprement parler. La variable « res » du code 1 est représentée par les nœuds dn_45 et dn_43. Le nœud dn_45 est celui qui contient le résultat de l'incrémentation et le nœud dn_43 représente l'ancienne valeur. Les nœuds dn_41 et dn_46 quant à eux représentent respectivement la variable « i » avant et après son incrémentation. Le nœud an_40 effectue le test d'arrêt de la boucle et l'instruction de saut cj_270 regarde alors si le test d'arrêt est vérifié pour décider si il faut ré-exécuter ce BB ou passer au BB final (bb_116).
— Le troisième BB (bb_116, figure 2.1(c)) contient les instructions à exécuter après la boucle. Dans l'exemple présent, il faut retourner le bon résultat. Le nœud an_94 permet donc de choisir entre la valeur 0 et la valeur du nœud dn_97 qui représente dans ce BB la variable « res » du code. Il prendra la valeur 0 si le BB précédent était bb_35 et la valeur de « res » si c'était bb_93.

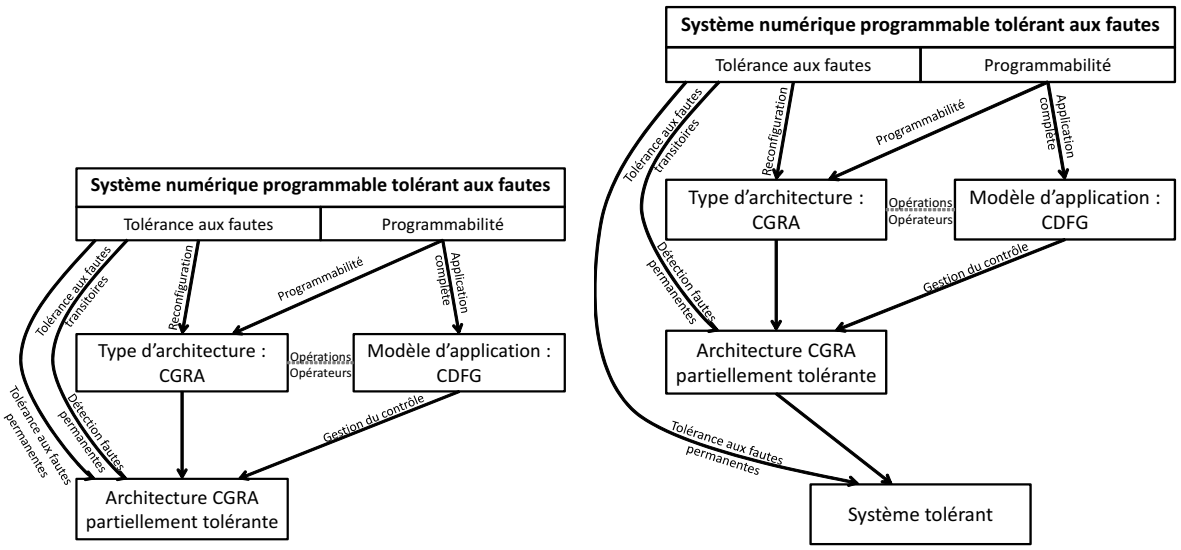
Le nombre et la taille des BBs dépendent beaucoup des options de compilation (et donc des différentes optimisations de codes) comme cela sera illustré dans la section 4.2.3. Les nœuds ont tous des identifiants (IDs) différents car le flot utilise une représentation *Static Single Assignement form* (SSA), c'est-à-dire à identifiant unique.

Dans ce modèle, des variables peuvent être initialisées dans un BB, utilisées dans un deuxième (pour effectuer des *loads* ou des *stores* par exemple), modifiées dans un troisième (pour être incrémentées), etc. Ces variables sont partagées entre les différents BBs. Ces différents aspects ont des conséquences sur la méthode que nous proposons et seront plus particulièrement traités dans le chapitre 4.

2.3 Système global

Comme justifié au chapitre précédent, l'utilisation d'un CGRA semble être le meilleur compromis pour permettre de tolérer à la fois les fautes transitoires et les fautes permanentes. Cependant, dans la littérature, les CGRAs sont quasi-systématiquement accompagnés d'un processeur généraliste. En effet, ils ont initialement été créés pour accélérer des cœurs de boucles. Dans le cadre de la tolérance aux fautes, il faudrait néanmoins que l'intégralité du code soit projeté sur une seule architecture. Cela implique que le CGRA soit capable d'exécuter non pas simplement des flots de données généralement représentés par des DFGs, mais aussi le flot de contrôle (CFG) et donc des CDFGs. Nous proposons donc de définir une architecture CGRA capable d'exécuter l'intégralité d'une application. Ceci nécessite d'intégrer la gestion du contrôle dû au modèle de l'application (figure 2.2a) dans l'architecture mais aussi de définir un flot permettant de projeter l'application complète sur le CGRA.

« Changer le fonctionnement de l'architecture » revient à reconfigurer le composant. Il s'agit d'un changement dynamique qui s'adapte en fonction du besoin et de l'état actuel du système. Pour un CGRA, cela revient à le reconfigurer, c'est-à-dire à changer le *mapping* utilisé. Comme montré dans [Berg et al., 2008] et utilisé dans [Zhao et al., 2011, Bolchini et al., 2011, Eisenhardt et al., 2011], le module qui gère la reconfiguration doit être un élément extérieur à l'architecture pour assurer un fonctionnement correct. C'est pourquoi le système que nous proposons dispose d'un module extérieur au CGRA réalisant la reconfiguration (figure 2.2b). Cependant, l'état de l'art actuel ne permet pas d'envisager que le système détermine la nouvelle configuration « à la volée », c'est-à-dire en effectuant le processus de synthèse/compilation en « temps réel ». Il faut que les configurations soient déterminées à l'avance et, que durant l'exécution, le module de reconfiguration n'ait qu'à choisir la configuration adaptée à l'état du composant. Notre proposition contient donc la définition et la réalisation d'un flot d'implémentation permettant d'obtenir un grand nombre de configurations. Cet ensemble de configurations



(a) Prise en compte de la gestion du contrôle de l'application.

(b) Prise en compte de la tolérance aux fautes permanentes en dehors du CGRA.

FIGURE 2.2 – Illustration du cheminement logique menant à la proposition générale pour la tolérance à différentes étapes (suite de la figure 1.14).

vont servir pour la reconfiguration dynamique de l'approche retenue. Cette approche pour la tolérance aux fautes combine un procédé de masquage de fautes transitoires, de détection de fautes permanentes et de reconfiguration dynamique pour le CGRA. Le code applicatif dans son intégralité s'exécutera sur un CGRA qui implémentera les mécanismes de tolérance définis. Le système est rendu tolérant aux fautes permanentes grâce à un module externe au CGRA qui gère la reconfiguration dynamique en temps réel en se basant sur un ensemble de configurations trouvées en amont. La figure 2.3 schématise la proposition globale de tolérance.

2.4 Obtention de configurations différentes

Le cœur de l'approche repose sur un flot de projection pour lequel il faut définir un objectif. Cet objectif doit permettre de prendre en compte le fait que des fautes permanentes sont susceptibles d'apparaître dans l'architecture. Sur un CGRA – qui possède un grand nombre de ressources de calculs – il est intuitif de penser que pour minimiser la probabilité d'apparition d'erreurs, il faut minimiser le nombre de ressources utilisées, *i.e.* la surface utilisée. Ainsi, si le flot permet de générer des *mappings* qui utilisent le plus petit nombre de ressources tout en respectant les contraintes de l'utilisateur, alors ces configurations devraient permettre de faire durer le plus longtemps possible le composant. Cependant, ce n'est pas suffisant. Considérons le CGRA 4×4 torique de la figure 2.4a. Supposons que les tuiles 1, 2, 4, 9, 10 et 16 sont défectueuses et que le *mapping* utilisant le moins de tuiles est en forme de croix comme celui de la figure 2.4b en vert. Si la tuile 8 devient défectueuse comme dans la figure 2.4c, alors il n'est plus possible d'utiliser ce *mapping* sur l'architecture alors qu'elle possède encore plus de la moitié de ses ressources. Pour pouvoir continuer à fonctionner, il faudrait avoir par exemple une configuration « en ligne » comme celle de la figure 2.4d.

On peut donc conclure qu'avoir uniquement le *mapping* ou un ensemble de *mappings* de taille minimale n'est pas suffisant pour répondre à notre problématique. Il faut avoir à disposition d'autres *mappings*. La littérature ne fournit pas de méthode permettant de déterminer en temps restreint une configuration ou *mapping* sur un CGRA. Il faut donc obtenir avant l'exécution du code applicatif sur l'architecture un grand nombre de configurations différentes pour ce dernier. Pour cela, il faut projeter le code sur l'architecture de sorte qu'il « l'utilise » de différentes façons.

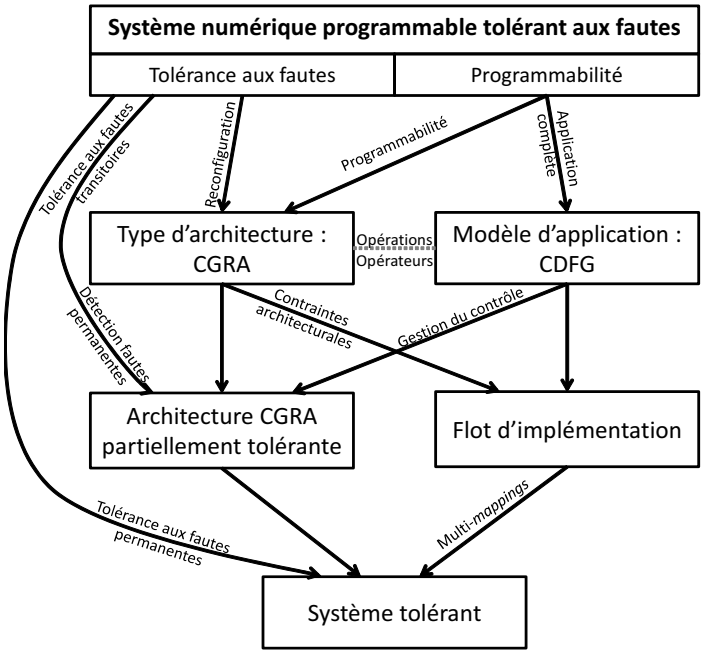
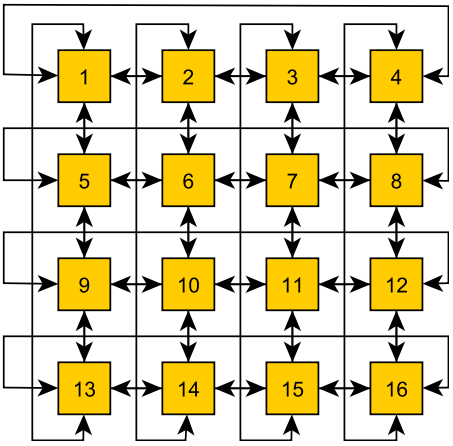
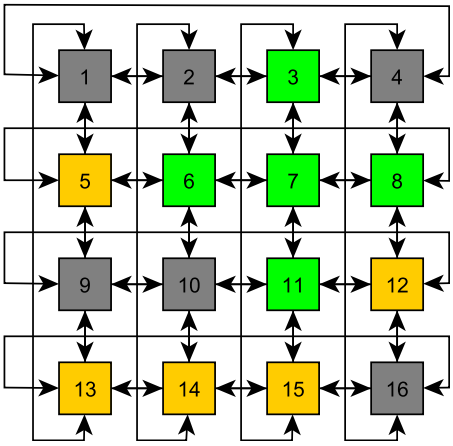


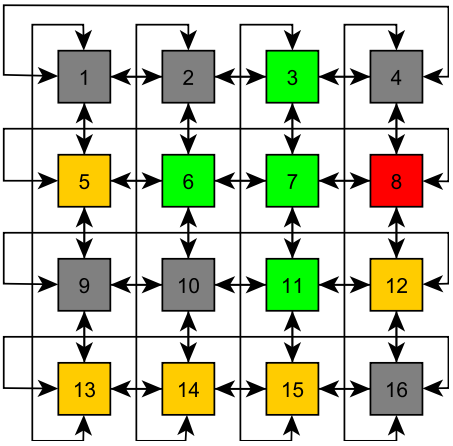
Figure 2.3 – Illustration du cheminement logique de la proposition générale finalisée.



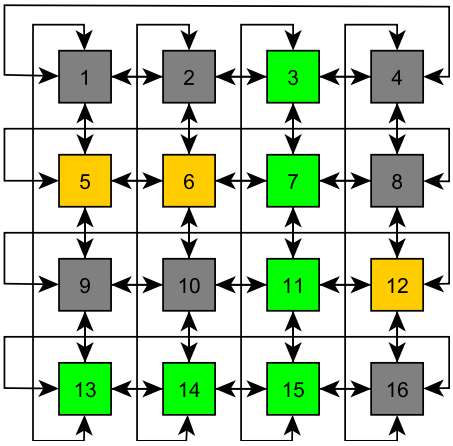
(a) Architecture initiale.



(b) Architecture en cours de fonctionnement après 6 fautes.

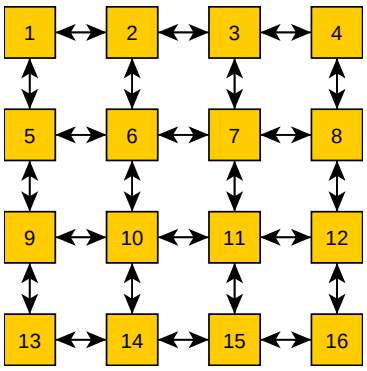


(c) Architecture après une faute sur la tuile 8.

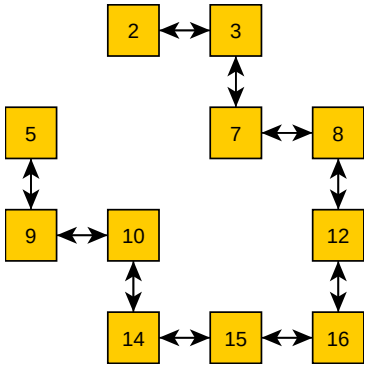


(d) Fonctionnement après changement pour un *mapping* linéaire utilisant 6 tuiles.

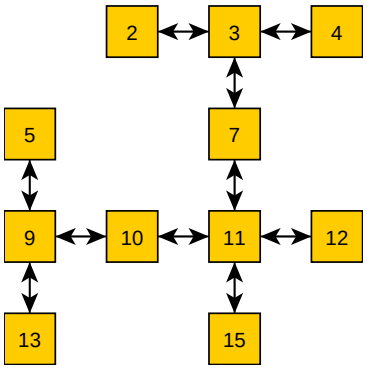
Figure 2.4 – Illustration de la nécessité de *mappings* différents.



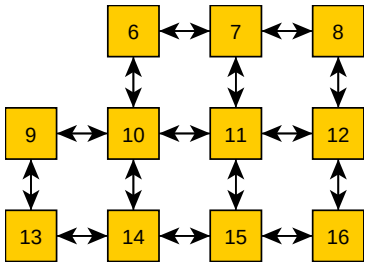
(a) Architecture d'origine.



(b) 1^{er} exemple.



(c) $2^{ème}$ exemple.



(d) $3^{ème}$ exemple.

FIGURE 2.5 – Trois exemples d'apparition de cinq fautes permanentes dans un CGRA 4×4 possédant un mesh-2D. Les tuiles fautives ne sont pas représentées pour plus de lisibilité.

2.4.1 Notion de réseaux

Une « utilisation » correspond à l'ensemble des ressources qui sont utilisées par le code au cours de son exécution (*i.e.* une projection géométrique selon l'axe temporel du *mapping* sur l'architecture). Il ne s'agit pas simplement de la localisation des ressources dans l'architecture, mais aussi de la façon dont elles sont reliées. Dans le cas d'un CGRA, un réseau est constitué par un ensemble de tuiles (ressources) reliées par une sous-partie de l'interconnexion. Cette association « tuiles + interconnexion » est appelée un « réseau ».

2.4.1.1 Exemples

Considérons un CGRA 4×4 possédant une interconnexion de type mesh-2D simple illustré en figure 2.5a. Les figures 2.5b, 2.5c et 2.5d présentent trois possibilités de dégradation de l'architecture par cinq fautes permanentes. Dans le premier cas (figure 2.5b), le réseau restant est équivalent à un réseau linéaire avec chaque tuile pouvant communiquer avec deux voisins à l'exception des deux extrémités qui ne peuvent communiquer qu'avec un seul voisin. Dans le deuxième cas, le réseau restant est assez hétérogène avec six tuiles pouvant communiquer avec un seul voisin, deux avec deux, deux avec trois et une avec quatre. Dans le troisième cas le réseau résultant reste très connecté. Une application qui a été compilée pour un de ces réseaux précis ne pourra pas s'exécuter simplement sur un autre de par les différences de capacité de communication. Il faut donc arriver à générer un ensemble de *mappings* pouvant s'exécuter sur ces sous-réseaux.

Tableau 2.1 – Durée de compilation pour effectuer la projection d'une application sur toutes les possibilités de réseaux de différentes architectures.

Taille du CGRA	Durée totale	
	Hypothèse d'une minute	Hypothèse d'une seconde
2×2	16 minutes	16 secondes
3×3	8,5 heures	8,5 minutes
4×4	45,5 jours	18,2 heures
5×5	63.8 ans	1 an
8×8	35 000 milliards d'années	584 milliards d'années

2.4.1.2 Approche mathématique

Comme cela a été illustré par l'exemple précédent, à partir d'une architecture CGRA possédant un réseau d'interconnexion simple, il est possible d'obtenir, après l'apparition de quelques fautes permanentes, des réseaux complètement différents dans leur capacité à exécuter un code. En fait, chaque tuile possède deux possibilités d'état : non fautive ou fautive. De ce fait, pour un CGRA possédant $nbTuiles$ tuiles, il y a $2^{nbTuiles}$ possibilités de réseaux sur lesquels il faudrait avoir un *mapping*. L'ordre de grandeur de la durée de l'obtention d'un *mapping* pour les méthodes de l'état de l'art est d'environ une minute. Le tableau 2.1 donne pour quelques tailles de CGRA le temps de compilation total pour obtenir un *mapping* par réseau en supposant que le temps de compilation vaut soit une minute, soit une seconde par projection. Au vu des valeurs pour un CGRA 4×4, dans les deux cas, il n'est pas envisageable d'utiliser l'énumération de tous les réseaux qui pourraient être obtenus en cas de fautes et de projeter l'application sur chacun d'entre eux.

De manière à diminuer ce temps, il est possible de remarquer que cette énumération est très redondante. En effet, par exemple dans la figure 2.5b, si la tuile numéro 7 était fautive à la place de la tuile numéro 4, le résultat de la compilation serait identique en termes d'exécution. De même, le résultat de la compilation ne change pas d'un réseau à l'autre si il existe des symétries, des rotations ou des translations permettant le passage de l'un à l'autre. Prendre en compte ces éléments de géométrie permet de réduire le nombre de réseaux pour lesquels il faudrait réaliser les projections et ensuite regénérer les configurations manquantes.

Dans le cas d'un réseau de type mesh-2D, la détermination des réseaux pour lesquels il est intéressant d'effectuer la projection est quasi-équivalente à celle de déterminer les Polyominos à Forme Libre (PFLs). Les PFLs ont été nommés ainsi par Solomon Golomb [Golomb, 1996] qui cherchait à trouver et dénombrer des pavages de l'espace à base de carrés. Les Polyominos à Forme Fixée (PFFs) correspondent aux pavages strictement différents de l'espace par translations et rotations alors que les PFLs correspondent aux pavages qui sont différents par symétries, rotations et translations. La figure 2.6 illustre les différents PFFs possibles avec 4 éléments [1]. Dans une optique d'obtention de *mappings* différents, il n'y a que trois réseaux véritablement différents : le carré, celui en forme de T et la ligne, en bleu sur la figure. Les autres pavages sont semblables à la ligne. Il y a donc sept PFFs à quatre éléments, mais seulement trois PFLs.

2.4.1.3 Obtention des réseaux différents

Le problème de l'obtention des PFFs et des PFLs est très complexe. La génération en se basant sur les solutions précédentes n'est pas aisée non plus pour ne pas oublier des réseaux « inédits ». Il existe néanmoins quelques algorithmes dans la littérature qui ont permis en 8 mois de calcul d'obtenir les PFLs possédant jusqu'à 28 éléments (il y en a 153 511 100 594 603) et les PFFs jusqu'à 56 éléments [Oliveira e Silva, 2007]. Le tableau 2.2 donne quelques uns des

1. Ces réseaux ont été démocratisés par le célèbre jeu vidéo Tetris.

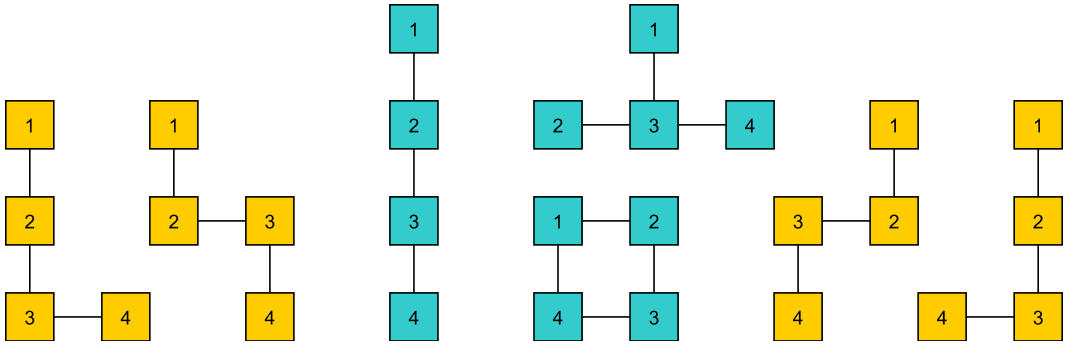


FIGURE 2.6 – PFFs possédant quatre éléments. Les PFLs sont bleus.

Tableau 2.2 – Illustration du nombre de Polyominos à Forme Fixée (PFFs) et à Forme Libre (PFLs) [Jensen and Guttmann, 2000, Guttmann, 2009].

Nombre d'éléments	Nombre de PFFs	Nombre de PFLs
2	2	1
3	6	2
4	19	5
5	63	12
6	216	35
7	760	108
8	2 725	369
9	9 910	1 285
16	104 592 937	13 079 255
25	20 457 802 016 011	2 557 227 044 764
56	69 150 714 562 532 896 936 574 425 480 218	?

nombres de polyominos pour différents nombre d'éléments et illustre que même en tenant compte des propriétés mathématiques des réseaux, au delà de quelques éléments ce nombre devient non raisonnable. Par exemple pour des réseaux à 16 éléments, les 13 079 255 possibilités entraîneraient un temps de compilation de plusieurs années en considérant un temps unitaire d'une seconde. Une méthode se basant sur une génération exhaustive de réseaux ne passe donc pas à l'échelle.

2.4.2 Implications sur le flot

Obtenir des « utilisations » différentes pour un code, et donc pouvoir s'adapter à un ensemble de fautes permanentes, peut se faire de deux façons. La première consiste à énumérer les différents réseaux possibles et pour chacun d'eux lancer un processus de projection qui serait donc sous contrainte de réseau. Mais comme montré au paragraphe précédent, ce n'est pas une solution réaliste. La seconde façon de faire consiste à non pas être sous contrainte de réseau, mais sous objectif de réseau. Dans cette approche, le processus de projection doit essayer d'obtenir le plus de réseaux différents possibles, sans avoir connaissance *a priori* de ceux-ci. Pour la mettre en œuvre, il est possible de définir une heuristique ou une meta-heuristique. Cette méthode doit alors réaliser la projection de l'application sur le CGRA de manière à obtenir le plus d'utilisations différentes à la fin. C'est ce type d'approche que nous nous proposons d'explorer dans ce manuscrit.

2.4.3 Implications sur l'interconnexion

Le choix du réseau d'interconnexion n'est pas sans conséquence sur la méthode permettant d'obtenir des utilisations différentes et sur le nombre de réseaux de tuiles qui peuvent être

obtenus. Dans les paragraphes précédents, l'hypothèse faite était que le réseau d'interconnexion des tuiles était un mesh-2D simple. Soient quatre CGRAs A, B, C et D possédant 16 tuiles organisées en 4×4 différant uniquement par leur réseau d'interconnexion :

— le CGRA A possède un réseau d'interconnexion de type mesh-2D simple (figure 2.7a) ;
— le CGRA B possède un réseau d'interconnexion de type mesh-2D torique (figure 2.7b) ;
— le CGRA C possède un réseau d'interconnexion de type mesh-plus (figure 2.7c) ;
— le CGRA D possède un réseau d'interconnexion de type mesh-X (figure 2.7d).

Lorsque l'on considère des fautes permanentes sur certaines tuiles, le réseau formé peut considérablement changer. La figure 2.8 donne l'architecture résultante après l'apparition de huit fautes permanentes (en enlevant les tuiles défectueuses) :

— dans le cas du mesh-2D simple (figure 2.8a), trois tuiles se retrouvent isolées et ne pourront être utilisées que si l'application possède des parties indépendantes et peu gourmandes en ressources. Le plus grand réseau possède alors uniquement cinq tuiles ;
— dans le cas du mesh-2D torique (figure 2.8b), une des tuiles qui était précédemment isolée reste reliée, ce qui porte à six tuiles le plus grand réseau. Le coût à payer est l'ajout des liens entre les tuiles de côté qui va augmenter le temps de propagation et limiter la fréquence de fonctionnement en fonction de la dimension du CGRA ;
— dans les cas des mesh-plus et mesh-X (respectivement en figures 2.8c et 2.8d), l'ensemble des tuiles reste relié dans cet exemple en formant un seul réseau. Mais ceci au coût d'une interconnexion à huit voisins au lieu de quatre.

Une tuile isolée étant moins à même d'exploiter le parallélisme d'une application qu'une tuile reliée à d'autres, il est préférable, dans l'architecture, de disposer d'un maximum de tuiles reliées. Cependant, il faut prendre en compte les coûts en surface et temps de propagation de tels réseaux.

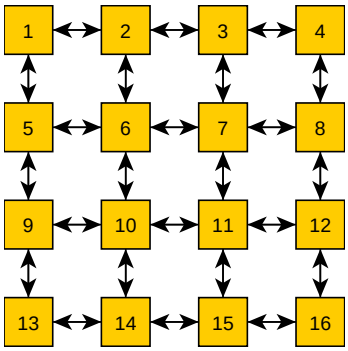
2.5 Problématiques liées à la méthode de projection

Pour répondre à la problématique d'obtention de configurations différentes, le flot doit trouver plusieurs *mappings* pour le code sur l'architecture. C'est ce que l'on appellera dans ce manuscrit de la « projection multiple ». Pour obtenir plusieurs *mappings*, il faut déjà arriver à en obtenir un. Projeter une application sur un CGRA consiste, pour chaque cycle d'exécution, à définir quelles sont les opérations qui seront exécutées, quelles sont les ressources qui les exécuteront, comment sera utilisée l'interconnexion et où seront stockés les résultats des calculs. Le premier point est appelé ordonnancement. Les deuxième et quatrième correspondent à l'assignation. Le troisième est appelé routage, mais lorsque l'on assigne une opération à une ressource, il n'est pas forcément difficile de vérifier si le routage est possible. Dans le cas du flot que nous proposons, ces deux étapes, parfois séparées, seront réunies. Comme cela a été montré dans l'état de l'art, il existe un grand nombre de méthodes permettant de résoudre les problèmes d'ordonnancement et d'assignation. Il existe cependant deux problématiques majeures avec l'ordonnancement qui nous ont poussé à explorer certaines voies plutôt que d'autres. Ces choix feront l'objet d'une discussion dans le chapitre détaillant notre flot (chapitre 3).

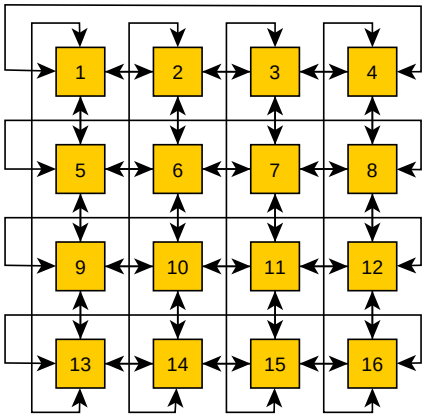
2.5.1 Ordonnancement et assignation séparés

Considérons le graphe d'opérations de la figure 2.9a. Supposons qu'il doit être ordonnancé sur le réseau de la figure 2.9b. Le tableau 2.3 en donne un ordonnancement en cinq cycles :

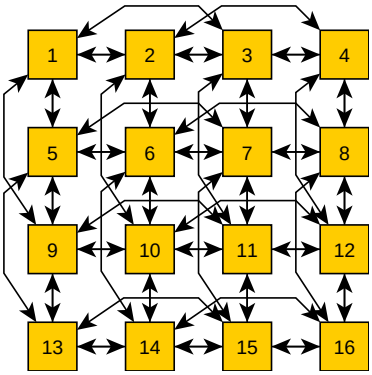
— les opérations 1, 2, 3, 4 et 5 sont ordonnancées au premier cycle ;
— les opérations 6, 7, 8, 9 et 10 sont ordonnancées au deuxième cycle ;
— les opérations 11, 12, 13 et 14 sont ordonnancées au troisième cycle ;
— les opérations 15 et 16 sont ordonnancées au quatrième cycle ;
— et la dernière opération (17) est ordonnancée au cinquième cycle.



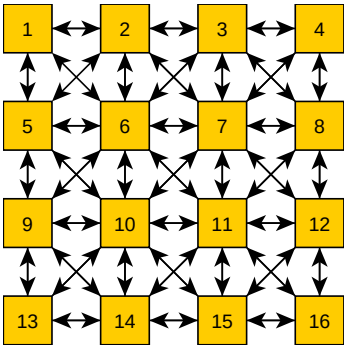
(a) CGRA A avec un mesh-2D simple.



(b) CGRA B avec un mesh-2D torique.

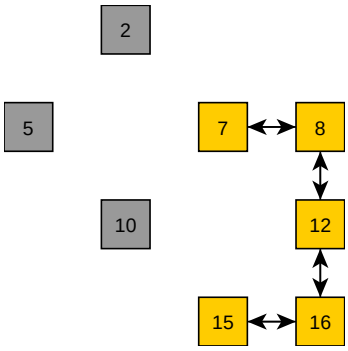


(c) CGRA C avec un mesh-plus.

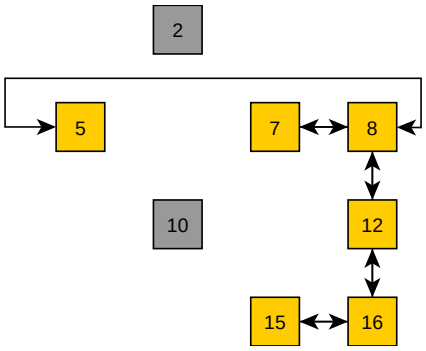


(d) CGRA D avec un mesh-X.

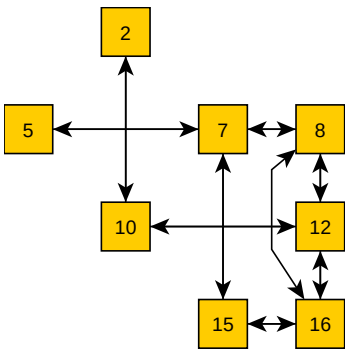
FIGURE 2.7 – Exemples de CGRAs 4×4 possédant des réseaux d'interconnexion différents.



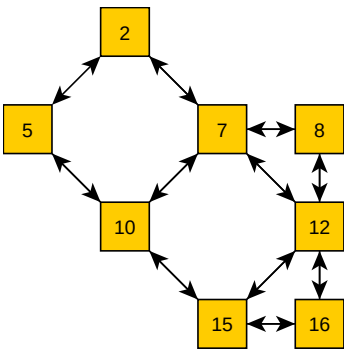
(a) CGRA A résultant.



(b) CGRA B résultant.

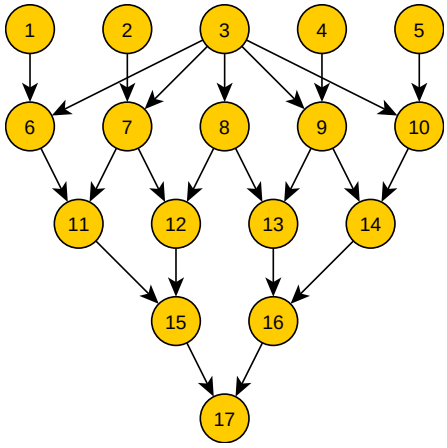


(c) CGRA C résultant.

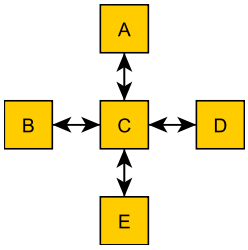


(d) CGRA D résultant.

FIGURE 2.8 – Architectures de la figure 2.7 après apparition de huit fautes sur les tuiles 1, 3, 4, 6, 9, 11, 13 et 14.



(a) Graphe d'opérations



(b) Réseau cible.

Figure 2.9 – Exemple problématique d'un ordonnancement séparé du placement.

Tableau 2.3 – Résultat sous forme de tableaux de l'ordonnancement du graphe d'opération de la figure 2.9a sur l'architecture de la figure 2.9b.

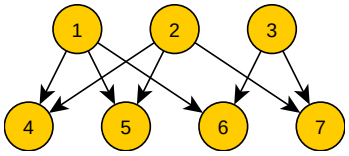
Cycle	Opérations ordonancées				
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	
4	15	16			
5	17				

Cet ordonnancement respecte les dépendances « producteurs-consommateurs », c'est-à-dire les dépendances de données. Il respecte le nombre d'opérateurs disponibles dans l'architecture. Il semble donc valide *a priori*. Cependant, les deux tentatives d'assignation présentées dans le tableau 2.4 illustrent l'impossibilité d'obtenir un *mapping* sur ce réseau avec cet ordonnancement. Pour le premier cycle, la tuile C est nécessairement occupée par l'opération 3 car c'est la seule qui offre une capacité de communication suffisante et permet d'accéder à l'ensemble des tuiles du réseau. En effet, l'opération 3 diffuse son résultat aux opérations 6, 7, 8, 9 et 10 qui doivent toutes être sur des tuiles différentes. Pour les autres opérations du cycle, il n'y a pas de contrainte particulière. Au second cycle, il y a deux possibilités : soit les opérations filles de 1, 2, 4 et 5 sont toutes exécutées sur les mêmes tuiles (tentative de gauche), soit il y a une permutation. La tentative de droite présente la cas d'une permutation de placement entre l'opération 7 et l'opération 8, mais le raisonnement reste le même avec toutes les autres permutations. Au

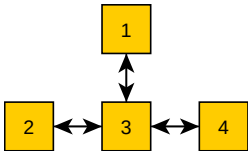
Tableau 2.4 – Résultat sous forme de tableaux de deux tentatives de placement des opérations ordonnancées de la figure 2.9a sur l'architecture de la figure 2.9b.

Cycle	Tuile				
	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3		12	?	13	
4					
5					

Cycle	Numéro de tuile				
	A	B	C	D	E
1	1	2	3	4	5
2	6	8	7	9	10
3	11	12	?		
4					
5					



(a) Graphe d'opérations



(b) Réseau cible.

FIGURE 2.10 – Exemple problématique d'un ordonnancement utilisant un tri topologique avant.

Tableau 2.5 – Résultat sous forme de tableaux de l'ordonnancement du graphe d'opérations de la figure 2.10a sur l'architecture de la figure 2.10b.

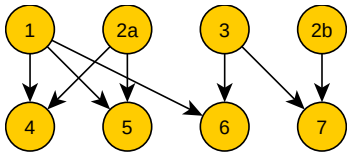
Cycle	Opérations ordonancées			
1	1	2	3	
2	4	5	6	
3	7			

troisième cycle, il apparaît nécessairement un conflit d'accès aux ressources. Dans le cas de la première tentative, l'opération 11 issue de 6 et 7 ne peut aller que sur la tuile C pour respecter la dépendance de données (car 6 et 7 sont placées sur des tuiles à la périphérie du réseau). Pour la même raison, l'opération 14, issue de 9 et 10, ne peut se placer que sur la tuile C. Les opérations 11 et 14 sont donc en conflit sur la tuile C, ce qui fait échouer la projection. Avec la seconde hypothèse, ce sont les opérations 13 – issue de 8 et 9 – et 14 – issue de 9 et 10 – qui sont en conflit sur la tuile C. Bien que seulement deux exemples soient présentés, il n'est en réalité pas possible de trouver un *mapping* en cinq cycles avec ce graphe d'opérations. Réaliser l'ordonnancement séparé de l'assignation peut donc conduire à ne pas trouver de solution. C'est pour cette raison que nous utiliserons un ordonnancement et une assignation simultanés.

2.5.2 Ordonnancement et sens du parcours de graphe

Lorsque l'on veut ordonnancer un graphe orienté, deux choix principaux se présentent sur le sens de parcours. Le premier choix consiste à parcourir les nœuds selon un tri topologique simple, ce qui revient à d'abord considérer les opérations qui n'ont pas de prédécesseur ou dont les prédécesseurs ont déjà été traités. Le second choix consiste à parcourir les nœuds selon un tri topologique inverse. Ce qui revient à considérer en premier les nœuds qui n'ont pas de successeurs ou dont les successeurs sont déjà tous ordonnancés. Une autre manière de voir ces parcours est de commencer par effectuer un ordonnancement sans contrainte « au plus tôt » ou *As Soon As Possible* (ASAP) et « au plus tard » ou *As Late As Possible* (ALAP). Puis, pour le tri topologique simple, il faut considérer les nœuds par valeur d'ASAP croissante. A contrario, pour le tri topologique inverse, il convient de considérer les nœuds par valeur d'ALAP décroissante.

Le choix de l'ordre dans lesquelles les opérations sont ordonnancées a aussi un impact sur la qualité du résultat. Considérons l'exemple du graphe d'opérations en 2.10a devant s'exécuter sur le réseau en 2.10b. Supposons que l'ordonnancement est réalisé simultanément à l'assignation, c'est-à-dire que l'existence d'une solution est directement vérifiée. Le tableau 2.5 donne le résultat de l'ordonnancement de ce graphe d'opération. Au premier cycle, toutes les opérations ordonnançables sont ordonnancées. Au deuxième cycle, les opération 4, 5 et 6 sont ordonnancées et placées, mais l'opération 7 ne peut pas être ordonnancée à ce cycle-là. En effet, l'opération 1 doit fournir son résultat à trois opérations et ne peut donc être placée que sur la tuile 3. De ce fait, l'opération 2 doit être sur une des tuiles périphériques et ne peut donc fournir son résultat qu'à deux opérations (4 et 5). L'opération 7 est donc retardée et ordonnancée au troisième cycle.



(a) Graphe transformé par « recalcul ».

Cycle	Opérations ordonancées			
1	1	2a	3	2b
2	4	5	6	7

(b) Résultat d'ordonnacement après transformation.

Figure 2.11 – Graphe d'opérations et résultat d'ordonnancement post-transformation.

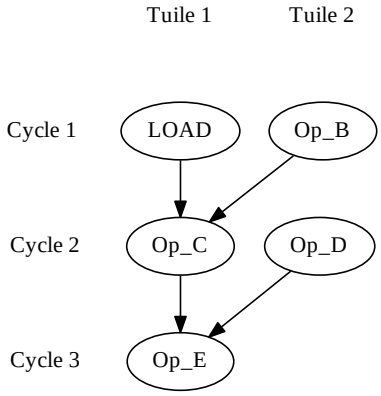
Dans [Hamzeh et al., 2012], il est montré que pour obtenir une meilleure latence, il est parfois nécessaire d'effectuer ce qui est appelé du « recalcul ». Il s'agit d'une transformation du graphe qui duplique une opération pour réduire le nombre de successeurs et ainsi permettre de la placer plus facilement car la pression sur le réseau d'interconnexion est moins forte. Dans cet exemple, pour pouvoir réduire le nombre de cycles d'exécution, il faudrait appliquer cette transformation sur l'opération 2. Le résultat de la transformation et de l'ordonnancement associé est donné en figure 2.11. Seulement, pour cela, il faudrait pouvoir se rendre compte que la capacité de communication du réseau d'interconnexion est dépassée. Or, dans le cas d'un ordonnancement basé sur un tri topologique simple, il n'est pas possible d'identifier les opérations qui seront réellement ordonnancées au cycle suivant. De ce fait, pour pouvoir obtenir la meilleure latence, il faut que le flot traite les opérations suivant un tri topologique inverse. En ordre inverse, les opérations 4, 5, 6 et 7 sont ordonnancées, puis l'opération 1 au cycle précédent. Quand l'algorithme tente d'ordonnancer et placer l'opération 2, il échoue et peut réagir en conséquence. C'est pour cette raison que le flot que nous proposons parcourt les opérations en ordre inverse.

2.6 Problématique de la gestion du contrôle de l'exécution sur CGRAs

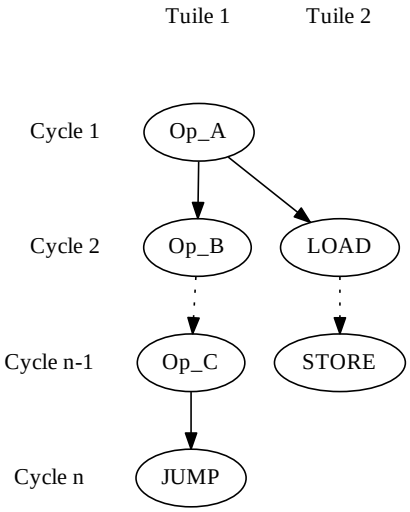
Dans la littérature relatant des CGRAs, il n'est jamais fait mention de la possibilité qu'une instruction d'accès à la mémoire (une lecture/*load* ou une écriture/*store*) ne se termine pas au moment où l'instruction est réalisée [2]. En effet, ces opérations sont généralement représentées par un simple nœud d'opération ayant une durée de un cycle. Or selon les mémoires (en particulier la mémoire *Dynamic Random Access Memory* (DRAM)), il peut exister un temps strictement supérieur à un cycle d'horloge processeur entre le moment où l'on demande une donnée et le moment où elle est disponible sur le bus de sortie de la mémoire. Cette durée est appelée *Column Address Strobe (CAS) latency*. Son ordre de grandeur est par exemple d'environ 8 cycles pour les mémoires DDR3 cadencées à 1 600 MHz et d'environ 10 cycles pour d'autres cadencées à 2 400 MHz. La mémoire peut aussi poser un autre problème : certaines technologies mémoires nécessitent un rafraîchissement pendant lequel il n'est pas possible d'accéder aux données. Ce temps de rafraîchissement peut d'ailleurs être très long (des centaines, voire des milliers de cycles d'horloge). C'est au concepteur de s'assurer que la donnée sera bien disponible au moment où elle est demandée. Cependant, pour pouvoir utiliser ce type de technologie mémoire, il faut pouvoir gérer le fait que le *load* ne soit pas terminé au moment où il devrait l'être. Négliger cet aspect peut s'avérer néfaste pour l'exécution du programme et pour la véracité des résultats.

Prenons par exemple le graphe partiel d'application présenté en figure 2.12a. Le résultat du *load* est utilisé par l'opération C avec le résultat de l'opération B. Ensuite, le résultat de C est utilisé par E avec le résultat de l'opération D. Supposons que l'outil de projection ait placé le *load* et les opérations C et E sur la tuile 1 et les autres opérations sur la tuile 2. Si tout va bien, le *load* et B seront exécutés au premier cycle, puis C et D et enfin E. Si le résultat du *load* arrive

2. Dans [Parashar et al., 2013], ce problème de la gestion des accès mémoire bloquants ne se présente pas. Si une donnée n'est pas disponible, alors l'instruction qui l'utilise ne pourra s'exécuter car son prédicat n'est pas validé. Ce problème est déporté sur la génération de prédicat et sur la distribution des instructions dans les tuiles.



(a) Exemple d'un graphe avec un *load* suivi de deux opérations.



(b) Exemple d'un graphe utilisant un *load* sur une branche indépendante.

FIGURE 2.12 – Graphes partiels problématiques de Basic Blocks utilisant un *load*.

seulement un cycle après, si rien n'est fait, C puis E donneront des résultats faux. Une solution simple serait d'empêcher l'exécution des instructions suivantes de la tuile tant que le *load* n'est pas terminé. Mais cette solution ne fonctionne pas si les autres tuiles continuent leur exécution alors qu'une tuile est bloquée. Dans l'exemple, nous obtiendrions le fonctionnement suivant :

— au premier cycle, l'opération B s'exécute en parallèle du *load* qui n'est pas terminé ;
— au deuxième cycle, le *load* se termine et l'opération D s'exécute ;
— au troisième cycle, l'opération C s'exécute mais prend comme opérande le résultat de D et non pas le résultat de B ;
— au dernier cycle, E prend une valeur non déterminée provenant de la tuile 2.

Cette exécution n'est pas acceptable et donc la solution ne peut donc pas être de simplement bloquer l'exécution d'une tuile. Un second exemple de fonctionnement problématique est donné dans la figure 2.12b. Dans cet exemple, le graphe se sépare en deux branches indépendantes dont l'une possède un *load* et l'autre se termine par un saut inconditionnel (JUMP dans la figure). Avec un code applicatif réel, ce pourrait être d'une part une branche qui calcule la nouvelle valeur de l'indice de boucle et qui vérifie si la condition d'arrêt est vérifiée, et d'autre part une branche chargeant des données à partir de cet indice et les stockant ensuite via le *store*. L'ordonnancement est tel que le saut s'effectue après la dernière instruction de toutes les branches. Cependant, si la branche effectuant le *load* est bloquée pour attendre que les données soient effectivement disponibles, alors le changement de graphe pourrait être effectué avant que la totalité des opérations de toutes les branches ne soit terminée. Ce comportement n'est évidement pas tolérable et il faut donc trouver de véritables solutions à ces problèmes.

2.7 Proposition générale

L'analyse de ces différents points a permis d'établir notre proposition pour répondre à la problématique de concevoir un système numérique tolérant aux fautes.

Cette proposition se compose de plusieurs parties distinctes mais interdépendantes :

— la première est la définition d'un flot permettant d'obtenir un ensemble de configurations/*mappings* différents d'une application complète pour un modèle d'architecture en supposant qu'il implémente les mécanismes de tolérance ;

— la deuxième est la définition d'une architecture CGRA capable d'exécuter une application complète et donc capable de gérer le flot de contrôle.

— la troisième est la définition d'un système tolérant aux fautes en se basant d'une part sur l'architecture CGRA définie précédemment, rendue tolérante aux fautes transitoires et capable de détecter les fautes permanentes et d'autre part sur un module de reconfiguration utilisant les *mappings* obtenus par le flot.

Ce manuscrit commencera dans un premier temps par présenter un flot capable de « multi-*mapper* » une application sur un CGRA que l'on suppose *a priori* capable de détecter les fautes permanentes et de corriger les erreurs transitoires. Étant donné la complexité de la réalisation d'un tel flot, le chapitre 3 décrira la solution que nous proposons pour des applications simples, c'est-à-dire sans flot de contrôle (DFG). Puis le chapitre 4 détaillera comment d'une part étendre ce flot pour des applications plus complexes, représentées par des CDFGs, et d'autre part comment l'utiliser dans le cadre de la tolérance aux fautes pour obtenir davantage de configurations et arriver à les exploiter en cours de fonctionnement. Enfin, le chapitre 5 détaillera tout d'abord notre proposition d'architecture CGRA capable d'exécuter un code complexe possédant du contrôle. Puis il décrira comment mettre en œuvre dans cette architecture les différents mécanismes de tolérance et de détection de fautes et comment utiliser ce CGRA pour réaliser un système complet tolérant aux fautes transitoires et permanentes.

Chapitre 3

Multi-projection de DFG sur CGRA

3.1 Introduction

Ce chapitre présente la méthode permettant d'effectuer la projection multiple de DFG sur CGRA que nous avons définie. Elle projette une application afin d'obtenir différentes utilisations d'une architecture CGRA. Cette méthode repose sur une représentation formelle de l'application et de l'architecture sous forme de graphes. Ces représentations sont rendues homomorphes de façon à rechercher tous les sous-graphes communs maximaux, ce qui revient à trouver tous les *mappings* possibles pour un ordonnancement donné. Pour un jeu de contraintes donné, cette approche nous permet de trouver l'ensemble des projections en une seule compilation.

Ce chapitre commence par présenter notre proposition de flot permettant de projeter une application sur une architecture de manière multiple ainsi que son évaluation. Puis, il décrit comment améliorer les performances de ce flot pour des applications et/ou des architectures de plus grandes tailles.

3.2 Multi-projection semi-exhaustive

Dans notre approche, nous proposons le flot de projection multiple dont la figure 3.1 présente une vue générale. Il est dit « semi-exhaustif » car contrairement aux approches exactes telles que la Programmation Linéaire en nombre Entier ou ILP [Raffin et al., 2010], il explore l'espace de solutions en combinant une heuristique et une approche exacte : une heuristique d'ordonnancement et une méthode exacte pour l'assignation. Ses points d'entrée sont d'une part une description fonctionnelle de l'application et d'autre part la description de l'architecture du CGRA ciblé. La description fonctionnelle de l'application est compilée pour fournir un DFG. Cette étape, réalisée par un front-end basé sur GCC, inclut des transformations de haut niveau comme par exemple des déroulages totaux de boucles, des passes de propagation de constantes, etc. Un générateur permet d'obtenir le modèle de l'architecture en fonction des paramètres fournis dans la description du CGRA ciblé. L'algorithme réalisant la projection multiple se base sur ces deux modèles pour générer l'ensemble des solutions et est composé de quatre étapes principales et d'une étape optionnelle d'optimisation améliorant ses performances quand elle est applicable. Il réalise pour chacun des nœuds du DFG l'ordonnancement et l'assignation de manière simultanée. En cas d'échec, il transforme le graphe en fonction du problème rencontré. À la fin de chaque cycle, une passe d'élagage est réalisée.

Cette section va présenter les modèles de l'application et du CGRA utilisés par notre algorithme puis détailler ses différentes étapes avant d'en évaluer les performances.

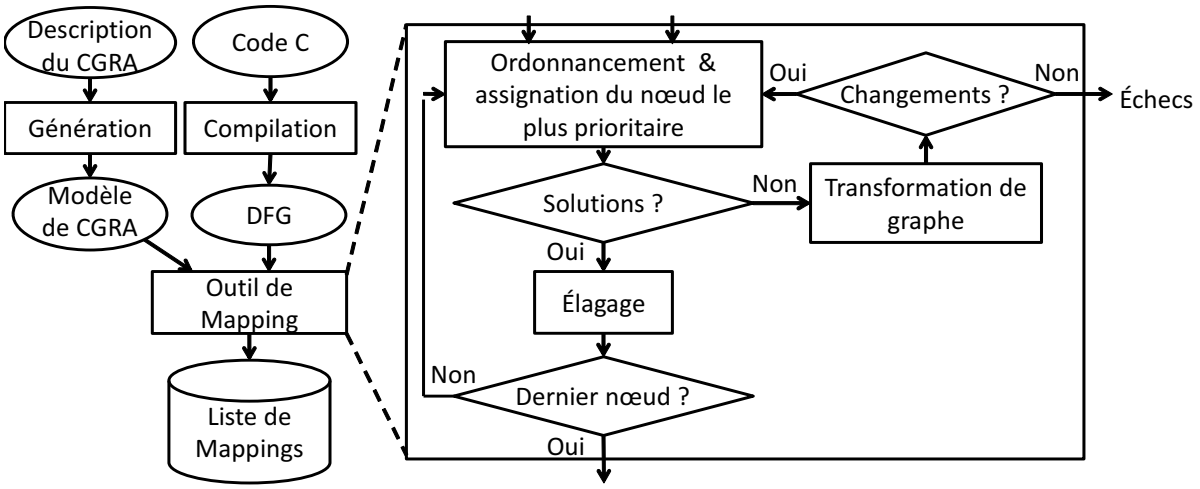


FIGURE 3.1 – Vue générale du flot de conception proposé.

3.2.1 Modèles utilisés

3.2.1.1 Modèle flot de données des applications

Notre modèle pour l'application est très semblable à celui classiquement défini pour un DFG. Il s'agit d'un graphe orienté biparti et acyclique. L'idée est d'introduire une notion temporelle dans ce DFG au travers du processus d'ordonnancement et d'assignation. L'obtention du modèle de l'application nécessite plusieurs étapes. L'application décrite en langage de haut niveau doit être compilée afin d'obtenir une représentation intermédiaire. La représentation intermédiaire que l'on utilise est appelée CDFG. Il s'agit d'une représentation versatile sous forme de graphe qui contient à la fois les successions d'opérations (flot de données) et le contrôle. Cette représentation intermédiaire est ensuite transformée de manière à supprimer le contrôle et n'avoir plus qu'un seul DFG.

Mathématiquement, notre DFG est un graphe $G_{DFG}(V_{op}, U_{var}, E_{DFG})$ orienté, acyclique et biparti. Il est composé d'un ensemble de nœuds d'opérations V_{op}, d'un ensemble de nœuds de variables U_{var} et d'un ensemble d'arcs orientés E. Un nœud d'opération ne possède qu'un seul arc sortant. Réciproquement, un nœud de variable ne possède qu'un seul arc entrant. L'enchaînement des nœuds est toujours alterné : opération → variable → opération. Il y a deux sortes de nœuds d'opérations : les opérations « classiques » et les opérations de mémorisation.

Dans notre modèle de DFG, il faut obligatoirement que les nœuds de variables soient explicites contrairement au modèle utilisé dans [Hamzeh et al., 2012]. De plus, nous définissons et ajoutons un nouveau type de nœuds : les nœuds de mémorisation. Les mémorisations sont des opérations à une seule entrée. Elles prennent en entrée une variable et génèrent en sortie une variable qui possède la même valeur. Concrètement, dans le DFG possédant un aspect temporel, la mémorisation permet d'explicitement conserver la variable au fil des cycles. Par exemple, dans la figure 3.2a, le nœud 2 produit un résultat utilisé par les nœud 3 et 4. Le respect des dépendances de données fait que ces nœuds ne peuvent pas être ordonnancés au même cycle. Le nœud 2' ajouté dans la figure 3.2b permet d'expliciter la conservation du résultat du nœud 2 pendant un cycle. Dans les représentations graphiques, les nœuds de mémorisation seront toujours représentés avec des cercles pointillés.

Le modèle peut être étendu afin de considérer les opérations multicycles. La figure 3.3 donne une illustration graphique de DFG avec une multiplication s'exécutant en deux cycles d'horloge, à la manière de [Hartenstein and Kress, 1995].

(a) DFG initial. (b) DFG après ajout d'une mémorisation.

Figure 3.2 – Exemple d'un DFG simple avant et après ajout d'un nœud de mémorisation.

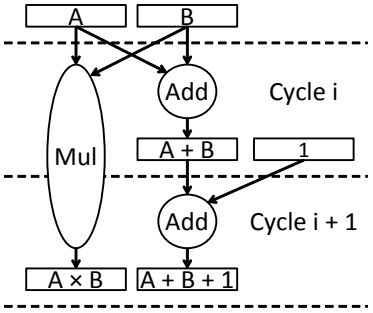


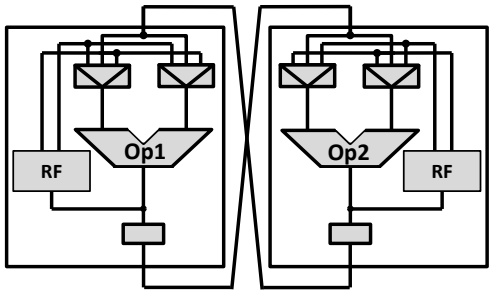
Figure 3.3 – Exemple de DFG utilisant une multiplication multicycle (sur deux cycles) et d'autres opérations ne nécessitant qu'un seul cycle.

3.2.1.2 Modèle d'architecture

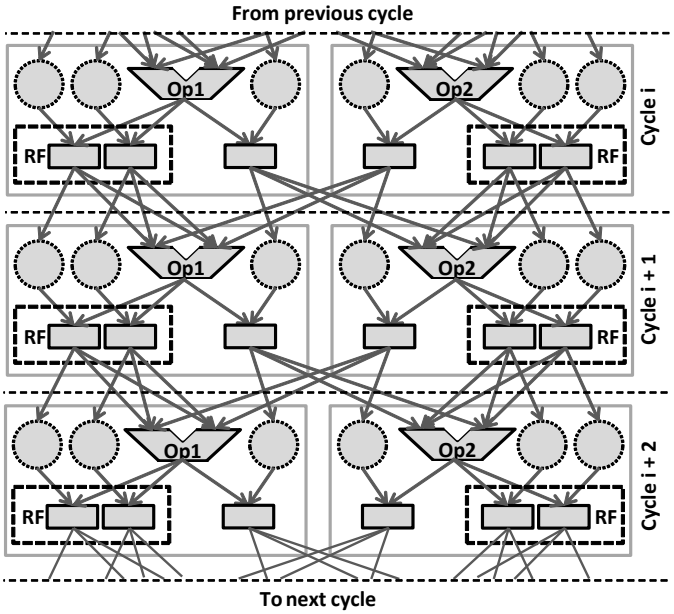
L'obtention du modèle de l'architecture se fait au travers d'un générateur dans lequel est décrite la taille de la matrice de tuiles voulue, la taille de la RF interne, le type de réseau d'interconnexion ainsi que les opérateurs présents dans les tuiles. L'interconnexion est un élément central dans un CGRA. Elle peut être régulière ou non, simple ou complexe, mais dans tous les cas, elle définie la façon dont communiquent les tuiles. Contrairement à un « réseau sur puce » ou *Network On Chip* (NOC), ce sont surtout les connexions entre les tuiles voisines qui sont prédominantes dans un CGRA. De plus cette communication s'effectue généralement en un seul cycle. Le registre de sortie d'une tuile permet de communiquer son résultat auprès des tuiles voisines (les quatre plus proches par exemple).

Dans notre modèle, dont un exemple est donné en figure 3.4b, l'interconnexion est représentée « à plat », c'est-à-dire uniquement par des liaisons point-à-points. Les multiplexeurs d'entrée des opérateurs ne sont pas représentés explicitement. Ils sont remplacés par des ports d'entrée dans le nœud de l'opérateur. Une tuile possédant un registre de sortie et qui est capable de communiquer avec ses quatre voisins qui possèdent chacun un opérateur à deux entrées aura donc huit arcs sortants, un pour chaque port d'opérateur atteignable. Cette représentation peut sembler *a priori* restreindre les possibilités de représenter des réseaux d'interconnexion complexes. Cependant, elle supporte les interconnexions suivantes :

— les liaisons point-à-points de tout type (*e.g.* un mesh 2D, mesh torique, mesh X à huit voisins, mesh plus, etc.) ;
— les liaisons type bus partagé autorisant plusieurs éléments à communiquer simultanément, comme par exemple un bus à « accès multiple à répartition dans le temps » ou *Time Division Multiple Access bus* (TDMA), en remplaçant le bus par son équivalent en point-à-point.



(a) Schéma d'un CGRA à deux tuiles possédant chacune une RF de deux registres.



(b) Modèle équivalent sur trois cycles d'exécution.

FIGURE 3.4 – Exemple simple du modèle d'architecture. Dans (b), les cadres sont présents pour faciliter le repérage des éléments. Les opérateurs de mémorisation sont en pointillés. Les registres sont rectangulaires et les opérateurs de calcul sont en forme d'ALU.

Mathématiquement, ce modèle est donc un graphe $G_{archi}(V_{Op}, U_{Reg}, E_{archi})$ orienté biparti composé d'un ensemble d'opérateurs V_{Op}, d'un ensemble de registres U_{Reg} et de liens de communications orienté E_{archi}. L'orientation des arcs permet de représenter l'écoulement des données dans l'architecture : registre → opérateur → registre. Les registres de la RF d'une tuile sont tous explicités à plat dans le modèle (sans niveau hiérarchique particulier).

Notre modèle pour l'architecture est donc similaire au TEC de [Hamzeh et al., 2012], mais en explicitant les registres de ses RFs. Cette explicitation permet de prendre en compte l'intégralité des registres et surtout de prendre en compte la durée de vie des variables. La mémorisation prend la forme d'un opérateur de mémorisation. Cet opérateur artificiel explicite l'action de conserver une valeur en registre. Il « externalise » cette conservation en dehors du registre. L'introduction de la temporalité dans le modèle d'architecture se fait implicitement grâce aux registres (plus exactement entre le registre et l'opérateur suivant). Ils forment la frontière entre deux cycles d'exécution. La façon dont sont connectés les arcs dépend de l'architecture de la tuile et de l'interconnexion.

La figure 3.4 donne un exemple de modèle équivalent à un CGRA élémentaire de deux tuiles possédant en interne un opérateur de calcul (représenté ici avec la forme classiquement donnée à une ALU), des opérateurs de mémorisation (un par registre), un registre de sortie et une RF interne de deux registres pendant trois cycles d'exécution. On observe en particulier que le registre de sortie (le troisième de chaque ligne de registres) peut recevoir une donnée soit de son opérateur de mémorisation, soit de l'opérateur de calcul de sa tuile. Il ne peut donner sa valeur qu'à son opérateur de mémorisation ou à l'opérateur de l'autre tuile. À l'inverse, un registre de la RF ne peut donner sa valeur qu'à l'opérateur de la tuile ou son opérateur de mémorisation. Dans le contexte de tolérance aux fautes, la notion de tuile n'est pas indispensable par rapport au modèle. Elle permet simplement de définir la granularité des éléments utilisables ou défectueux.

Le modèle peut également intégrer la représentation des opérateurs « lents » qui sont soit multicycles soit pipelinés :

— Un opérateur multicycle est un opérateur pour lequel il faut attendre un nombre de cycles donné entre le moment où il reçoit les données en entrée et le moment où le résultat est

disponible en sortie de l'opérateur. Il n'est donc pas disponible pour un nouveau calcul et est « bloqué ».
— Un opérateur pipeliné, comme un opérateur multicycle, ne fournira son résultat qu'un nombre de cycle donné après que les opérandes aient été fixés en entrée. Mais contrairement au précédent, à chaque nouveau cycle d'horloge, il peut recevoir de nouvelles entrées. Il augmente donc la latence, mais ne change pas la cadence.

Le modèle de l'architecture doit représenter à chaque cycle d'exécution le démarrage de l'opérateur long en plus du démarrage de l'opérateur court. Il suffit d'avoir un nœud opérateur par durée d'exécution. Lors de la phase d'ordonnancement, il faudra vérifier si l'opérateur désiré est libre à cet instant ou bien si il est déjà utilisé pour effectuer un calcul multicycle.
Cette représentation n'est par contre pas en mesure de représenter un bus partagé à accès unique. Cependant, moyennant quelques aménagements dans le flot, il serait possible de prendre en compte le fait que si un arc est utilisé, les autres arcs appartenant au même bus physique ne sont plus disponibles. Pour se faire, il faudrait définir des groupes d'arcs ne permettant qu'une seule utilisation simultanée.

3.2.1.3 Équivalence et utilisation des modèles

Dans [Hamzeh et al., 2012], il a été démontré que résoudre le problème du Sous-graphe Commum Maximal ou *Maximum Common Subgraph* (MCS) entre deux graphes homomorphes était équivalent à effectuer la projection de l'un sur l'autre dans le cas particulier où le plus grand graphe commun est un des deux graphes.

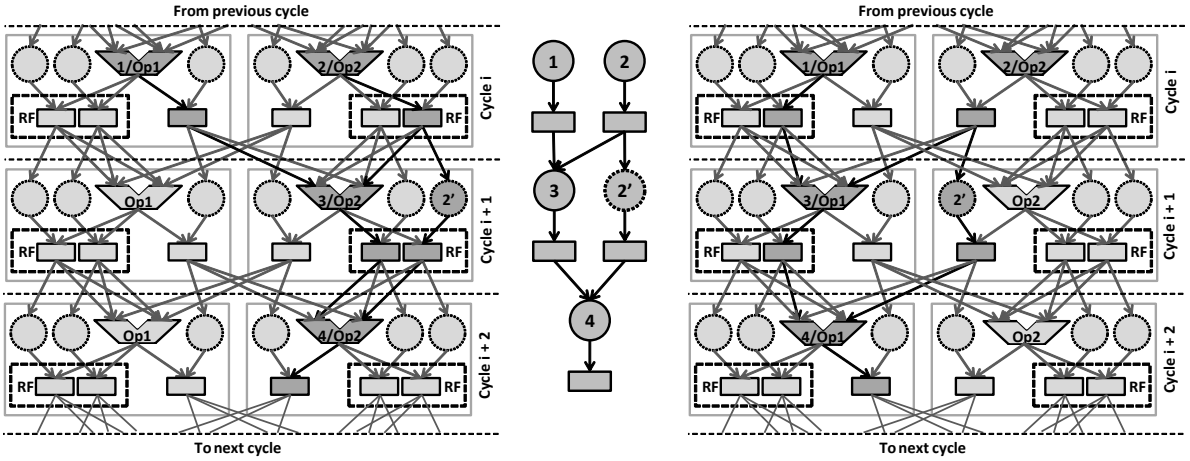
Définition Soit $G_1(V_1, E_1)$ et $G_2(V_2, E_2)$ deux graphes dont les nœuds sont notés V_1 et V_2 et les arcs E_1 et E_2 ; une application $f : V_1 \rightarrow V_2$ qui envoie les nœuds de G_1 sur ceux de G_2 est un homomorphisme si $\forall e \in E_1, f(e) \in E_2$. En d'autres terme, f est un homomorphisme si l'image de tout arc de G_1 est un arc de G_2. Dans le cas d'arcs orientés, il faut que les arcs soient orientés dans le même sens dans l'espace de départ et d'arrivée.

Définition Le problème du Sous-graphe Commun Maximal MCS de $G_1(V_1, E_1)$ et $G_2(V_2, E_2)$ se définit comme étant le plus grand sous-graphe de G_1 pour lequel il existe un homomorphisme f vers un sous-graphe de G_2.

Pour obtenir un *mapping*, il faut que l'homomorphisme soit de $G_{DFG}(V_{op}, U_{var}, E_{DFG})$ vers $G_{archi}(V_{Op}, U_{Reg}, E_{archi})$ et soit surjectif, c'est-à-dire que tous les éléments de l'espace départ (le DFG) aient une image dans l'espace d'arrivée (l'architecture). Il s'agit d'un problème NP-Complet. L'algorithme de Levi [Levi, 1973] est un algorithme permettant de résoudre ce problème en établissant entre autres des matrices de compatibilité entre les deux graphes et de construire petit à petit la plus grande solution. Le formalisme que nous avons introduit pour définir les modèles permet d'utiliser ce genre d'approche à condition de définir un homomorphise permettant de passer de l'un l'autre. Le paragraphe suivant définit les différentes règles permettant de construire cet homomorphisme.

Équivalences entre les deux modèles

De manière à pouvoir utiliser les outils de la théorie des graphes et en particulier l'algorithme de Levi [Levi, 1973], nous définissons des équivalences, ou plutôt des listes de compatibilité. La première compatibilité est celle des arcs. Dans les deux modèles, les graphes sont orientés et ils ne sont compatibles que si ils vont dans le même sens. Concrètement, cela signifie que les compatibilités vont dans le même sens par rapport au parcours des graphes. La deuxième compatibilité est celle entre les variables et les registres : les nœuds de variable ne sont compatibles qu'avec les nœuds de registre. La troisième compatibilité est celle entre les opérations et les



(a) Première variante de *mapping* avec mémorisation dans la RF.

(b) DFG.

(c) Deuxième variante avec mémorisation dans le registre de sortie.

FIGURE 3.5 – Deux exemples de *mappings* pour le DFG de la figure 3.2b. Les ressources utilisées sont d'un gris plus foncé que les ressources libres.

opérateurs. Parmi les opérations, deux grandes familles se distinguent : les opérations « classiques » dites « de calcul » et les opérations de mémorisation. Les opérations classiques d'un DFG sont l'addition, la multiplication, les opérations de logiques combinatoires etc., ainsi que les opérations d'accès à la mémoire en lecture (*load*) et en écriture (*store*) qui ne sont pas des opérations de mémorisation. Les opérations de mémorisation sont uniquement celles que l'on a ajoutées dans le graphe comme le nœud 2' de la figure 3.2b. Une opération de calcul n'est compatible qu'avec un opérateur capable d'effectuer ce calcul. Une opération de mémorisation est quant à elle compatible avec tous les opérateurs. Les opérateurs de mémorisation ont été spécialement introduits pour cela et n'importe quel opérateur est capable d'effectuer une égalité entre la sortie et l'entrée (« $Sortie = Entree$ »). Cette seconde équivalence est très importante dans la méthode car c'est elle qui permet de déplacer des variables stockées dans l'architecture et ainsi diminuer les problèmes de localité. Une opération multicycle n'est quant à elle compatible qu'avec un opérateur long.

Utilisation

Le processus de projection consiste à transformer le DFG de manière à le trouver dans le modèle d'architecture du CGRA. La figure 3.5 donne deux exemples de *mappings* pour le DFG de la figure 3.2b sur l'architecture simple de la figure 3.4. Le modèle du DFG contient un certain nombre de nœuds qui seront éventuellement modifiés lors du processus de projection comme expliqué dans la section 3.2.2.2. Pour le modèle de CGRA, il y a plusieurs moyens de l'utiliser. La première serait de l'étendre comme le TEC d'*EPIMap* [Hamzeh et al., 2012] sur l'ensemble des cycles d'exécution, ce qui reviendrait à modifier sa taille régulièrement car il n'est pas simple de connaître à l'avance le nombre de cycles dont aura besoin un DFG pour s'exécuter. Un moyen de réduire ce surcout est d'utiliser une représentation « factorisée » de l'architecture. Pendant la projection, il n'est pas forcément nécessaire de connaître l'intégralité du passé pour pouvoir traiter les nœuds courants. Une façon d'en tirer parti est de ne représenter que le nombre minimal de cycle d'exécution sur le graphe de l'architecture et de le faire boucler sur lui même en respectant bien les capacités de l'interconnexion.

3.2.2 Algorithme de projection multiple

L'algorithme 2 décrit le fonctionnement global de notre méthode de projection multiple. Il a en entrée un modèle d'architecture, le DFG à projeter et un paramètre pour l'optimisation.

Algorithme 2 Multi-projection de DFG.

Nécessite : dfg : le DFG de l'application à projeter
Nécessite : $cgra$: le modèle de l'architecture CGRA
Nécessite : $virtualisation$: le niveau auquel il est possible de virtualiser

```
1: lnAOrdonnancer ← Noeuds(dfg)                          // ln* signifie listeNoeuds*
2: lnOrdonnancables ← ∅
3: lnOrdonnancesPrec ← ∅
4: lnOrdonnancesCourant ← ∅
5: arbreProjections ← ∅
6: tant que lnAOrdonnancer.estNonVide() faire
7:     lnOrdonnancables ← getNoeudsOrdonnancables(lnAOrdonnancer, lnOrdonnancesPrec)
8:     AjoutMemPourNoeudsNonOrdonnancables(lnOrdonnancePrev, lnOrdonnancables)
9:     tant que lnOrdonnancables.estNonVide() faire
10:        op = NoeudPlusPrioritaire(lnOrdonnancables)
11:        nbSolution = AssignationIncrementale(op, cgra, arbreProjections, virtualisation, ...
12:                                             lnOrdonnancesCourant)
13:        si nbSolutionPartielles < 1 alors
14:            TransformationNoeud(op, dfg, lnAOrdonnancer, lnOrdonnancables)
15:        sinon
16:            lnOrdonnancesCourant.ajout(op)
17:            lnOrdonnancables.suppression(op)
18:        fin si
19:    fin tant que
20:    lnAOrdonnancer.suppression(lnOrdonnancesCourants)
21:    lnOrdonnancesPrec ← lnOrdonnancesCourants
22:    lnOrdonnancesCourants ← ∅
23:    Elagage(arbreProjections)
24: fin tant que
25: retourne arbreProjections
```

3.2.2.1 Ordonnancement arrière et assignation simultanés

La méthode d'ordonnancement et d'assignation est un point central de notre flot et est constituée des lignes 6 à 24 de l'algorithme à l'exception des lignes 8, 14 et 23. Comme expliqué précédemment, notre flot ordonnance et assigne les nœuds d'opérations un par un, et ce tant qu'il reste des nœuds non-traités. Un exemple illustrant le fonctionnement sera donné en section 3.2.2.3.

Ordonnancement

Notre approche utilise un ordonnancement de type « list-scheduling » qui est une heuristique dans laquelle les nœuds ordonnançables sont triés par ordre de priorité.

L'algorithme commence par déterminer l'ensemble des nœuds ordonnançables à ce cycle (ligne 7). Les nœuds du DFG étant considérés en suivant un ordre topologique inverse, une opération n'est ordonnançable que si tous ses enfants ont déjà été ordonnancés. Ce choix permet un plus grand nombre de transformations de graphe comme cela est expliqué dans la sous-section suivante (3.2.2.2).

Ensuite, tant que la liste des nœuds ordonnançables n'est pas vide, l'algorithme choisit le plus prioritaire (ligne 10). Notre fonction de priorité utilise en premier critère la mobilité des nœuds, qui est la différence d'instant d'exécution entre l'ordonnancement ASAP et l'ordonnancement ALAP [Paulin and Knight, 1989], puis, pour les nœuds ayant la même mobilité, le nombre d'arcs sortants. En effet, un nœud possédant un nombre d'arcs sortants élevé est *a priori* plus difficile à assigner en respectant les dépendances de données. Commencer par ces nœuds est une stratégie raisonnable pour obtenir la latence la plus petite possible. Nous avons aussi ajouté des moyens de changer la priorité de certains types de nœuds. Par exemple, les nœuds de mémorisation

Algorithme 3 Assignation incrémentale.

Nécessite : *op* : le nœud à assigner
Nécessite : *cgra* : le modèle de l'architecture CGRA
Nécessite : *virtualisation* : le niveau auquel il est possible de virtualiser
Nécessite : *arbreProjections* : la structure qui stocke l'ensemble des solutions partielles de projection
Nécessite : *lnOrdonnancesCourant* : liste des opérations déjà ordonnancées à ce cycle

```
1: nbSolutionPartielles = 0
2: pour tout contexte ∈ arbreProjections faire
3:     OpCandidats ← ObtentionOperateursCompatibles(op, cgra, virtualisation, arbreProjections, ...
4:                                                  lnOrdonnancesCourant)
5:     pour tout Operateur ∈ OpCandidats faire
6:         si EnfantsAccessibles(op, Operateur, cgra, arbreProjections) alors
7:             arbreProjections.ajout(op, Operateur, variable_op, registres)
8:             nbSolutionPartielles = nbSolutionPartielles + 1
9:         fin si
10:    fin pour
11: fin pour
12: retourne nbSolutionPartielles
```

peuvent avoir une priorité plus grande, égale ou inférieure autres nœuds d'opération. Augmenter leur priorité permettra de favoriser le déplacement des données dans l'architecture alors qu'à l'inverse la diminuer favorisera le placement d'opérations et donc aura pour effet de diminuer la latence. Dans le cadre de cette thèse, nous avons choisi de favoriser les nœuds d'opérations par rapport aux mémorisations. Cependant, l'effet de ce changement de priorité dépend des dépendances de données et mériterait une étude complète.

Les lignes 16 à 24 de l'algorithme sont assez classiques d'un *list-scheduling* : elles suppriment le nœud traité des nœuds ordonnançables, puis lorsque tous les nœuds ordonnançables ont été traités, elles mettent à jour les différentes listes.

Assignation

L'étape d'assignation est réalisée par les lignes 11-12 de l'algorithme principal. Contrairement à la méthode présentée dans [Hamzeh et al., 2012, Hamzeh et al., 2013], qui utilise une version régulière de l'algorithme de Levi [Levi, 1973] (*i.e.* l'algorithme cherche des solutions en profondeur d'abord), notre méthode d'assignation est une version incrémentale de l'algorithme de Levi. Elle consiste à considérer toutes les solutions partielles précédentes et pour chacune d'elles à trouver l'ensemble des solutions d'assignation pour le nœud ordonnancé, comme présenté dans l'algorithme 3.

Concrètement, pour chaque solution partielle précédemment trouvée, l'algorithme commence par lister quels sont les opérateurs compatibles avec l'opération, c'est-à-dire l'ensemble des opérateurs libres capable d'exécuter l'opération (ligne 3). C'est à cette étape qu'il est possible de limiter le nombre d'opérations d'un certain type et en particulier les opérations d'accès à la mémoire. Il n'est en effet pas raisonnable de penser que toutes les tuiles puissent accéder à la mémoire centrale en même temps. Il est beaucoup plus réaliste de limiter ces accès à quelques uns (*e.g.* 2 ou 4). Si cette liste est vide pour l'ensemble des solutions précédentes, alors il n'y a plus de ressources disponible dans ce contexte. Si elle ne l'est pas, l'algorithme vérifie qu'à partir de ces opérateurs il est bien possible d'atteindre l'ensemble de ses successeurs et qu'il existe bien des registres libres pour stocker le résultat de l'opération (le nœud de variable associé) sur les différents chemins menant aux endroits où sont placés les successeurs. Si il n'y a plus de candidat pour l'ensemble des solutions précédentes, c'est qu'il n'est pas possible pour le nœud d'attendre tous ses enfants. Cette information sera utilisée lors de la transformation du nœud. Quand le sous-graphe considéré contient l'intégralité des nœuds, le DFG est entièrement ordonnancé, assigné et routé. Les solutions trouvées sont alors des *mappings* complets.

3.2.2.2 Transformations dynamiques de graphe

Comme précisé précédemment, lorsque l'algorithme de placement n'arrive pas à trouver de solution, le flot transforme le graphe (algorithme 2, ligne 14). Plus précisément, l'algorithme transforme le nœud qui n'a pas pu être placé. Les transformations de graphe sont donc réalisées de manière dynamique et en connaissance de cause, contrairement à celles effectuées dans [Hamzeh et al., 2012, Hamzeh et al., 2013] qui sont statiques et *a priori*, ou celles effectuées dans les algorithmes de recuits simulés comme dans [Mei et al., 2002] et ses évolutions qui sont dynamiques mais aléatoires. Après une transformation, l'algorithme met à jour les listes de nœuds en enlevant éventuellement le nœud courant de la liste des nœuds ordonnançables si la transformation l'impose.

Les différentes transformations

Nous avons défini quatre transformations pour un nœud qui sont représentées dans la figure 3.6 :

1. La première transformation : la « scission d'opération », consiste, pour une opération possédant plusieurs successeurs, à dupliquer l'opération (et son nœud de variable associé) et à répartir équitablement les successeurs entre les deux. Ainsi, chacun des nœuds possède un nombre inférieur de successeurs ce qui facilitera leur placement. Dans la figure, les nœuds qui précédent le nœud 1 ne sont pas représentés, mais s'ils existent, il faut créer les liens de dépendance de données pour le nœud dupliqué. Cette transformation est donc « à double tranchants » dans le sens où elle divise par deux le nombre d'arcs sortants pour un nœud (le nœud transformé), facilitant son placement, mais augmente pour les nœuds parents le nombre d'arcs sortants. Cette transformation est l'équivalent pour notre flot du « *recomputing* » proposé dans [Hamzeh et al., 2012, Hamzeh et al., 2013].
2. La deuxième transformation est le « routage simple » et consiste à repousser dans le temps l'exécution d'une opération. Au cycle courant, c'est une opération de mémorisation qui sera exécutée et au cycle précédent l'opération transformée. Cette transformation ne change pas le nombre d'arcs sortants ou entrants des différent nœuds, mais peut augmenter la latence du graphe en ajoutant un nœud de mémorisation sur le chemin critique. Cette transformation est équivalente au « *routing* » décrit dans [Hamzeh et al., 2012, Hamzeh et al., 2013].
3. La troisième transformation : la « scission de mémorisation », est l'équivalent de la scission d'opération mais effectuée sur un nœud de mémorisation. Elle permet de diminuer le nombre d'arcs sortants des mémorisations, facilitant leur placement, mais aussi permettant au résultat de l'opération de se déplacer dans l'architecture pour atteindre des successeurs dont les placements sont éloignés les uns des autres.
4. La quatrième transformation : le « routage et scission combinés », est la réalisation successive d'un routage et d'une scission de mémorisation.

Les cas d'utilisation

Ces différentes transformations ne doivent pas être appliquées de manière aléatoire au risque de détériorer la latence globale. Nous avons défini différentes règles permettant de choisir quelle est la meilleure transformation en fonction de la situation courante. En effet, l'algorithme d'ordonnancement et de placement, permet de connaître la cause de l'échec et donc d'appliquer la transformation la plus adéquate :

— lorsqu'un nœud d'opération n'est pas ordonnançable par manque de ressource, il faut utiliser du routage simple (en tout cas, dans un premier temps : il faudra éventuellement scinder le nœud généré si besoin, mais si ce n'est pas nécessaire, il vaut mieux ne pas le faire). Un manque de ressource peut survenir pour deux raisons : la première est qu'il n'y a plus d'opérateur capable d'effectuer l'opération, ce qui peut être le cas dans une

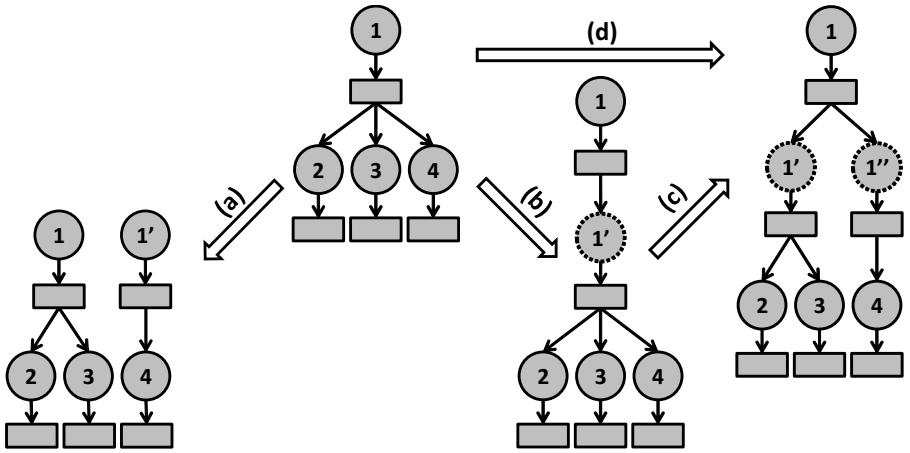


FIGURE 3.6 – Les quatre transformations de graphe : **(a)** scission d'opération sur le nœud 1, **(b)** routage simple sur le nœud 1, **(c)** scission de mémorisation sur le nœud 1', **(d)** routage et scission combinés sur le nœud 1.

architecture hétérogène ; la seconde est que l'intégralité des opérateurs du CGRA est utilisée ;

- lorsqu'un nœud de mémorisation n'est pas plaçable parce qu'il n'y a pas de solution de placement qui lui permette d'atteindre tous ses successeurs, il faut scinder le nœud ;
- lorsqu'un nœud d'opération n'est pas plaçable parce qu'il n'y a pas de solution de placement lui permettant d'atteindre tous ces successeurs, le choix est plus complexe car il y a trois transformations applicables au nœud d'opération. Une fonction de coût détaillée ci-après permet de prendre la décision ;
- le dernier cas de figure est celui d'une opération de mémorisation ne possédant qu'un seul successeur et qui est non-plaçable. Cela peut arriver quand l'intégralité des registres des tuiles est occupée avec des variables différentes de celle à mémoriser. Dans ce cas, il n'existe pas de transformation pertinente, il n'y a donc pas de changement et, comme indiqué dans la figure 3.1, c'est l'échec de la projection.

La fonction permettant de choisir la transformation du nœud d'opération prend en compte le nombre et le type d'opérations restantes à *mapper*, le nombre et le type de ressources de calcul libres, le nombre d'opérandes de l'opération, sa mobilité et le nombre de successeurs. Ces paramètres sont ceux qui nous ont parus les plus discriminants entre les nœuds, mais cette liste n'est pas exhaustive et d'autres paramètres pourraient éventuellement améliorer l'heuristique de choix comme le nombre de successeurs des parents du nœud, etc.

Notre première politique de choix reposait sur la prise en compte du nombre d'arcs entrants : si le nœud possède deux arcs entrants, il est routé ; sinon, il est scindé. La justification de cette approche vient du fait que scinder un nœud possédant plus d'un arc entrant va augmenter le nombre d'arcs et augmenter la pression sur le réseau d'interconnexion. Cependant, cette première approche donnait de très mauvais résultats en termes de latence et bande passante mémoire du fait qu'elle scindait des nœuds d'accès à la mémoire et qu'elle scindait des nœuds qui ensuite étaient tous deux routés.

Une politique plus élaborée a été mise en place, elle applique les règles suivantes :

- si il n'y a plus de ressource libre, le nœud est routé ;
- si le nœud est un nœud d'accès à la mémoire, il est systématiquement routé et scindé (quatrième transformation) car si un *load* n'est pas plaçable et qu'il reste des ressources, c'est parce qu'il n'arrive pas à accéder à ses enfants ;
- si il reste plus de ressources dans l'architecture que d'opérations à placer, alors le nœud est scindé ;
- si ce n'est pas le cas, il sera routé.

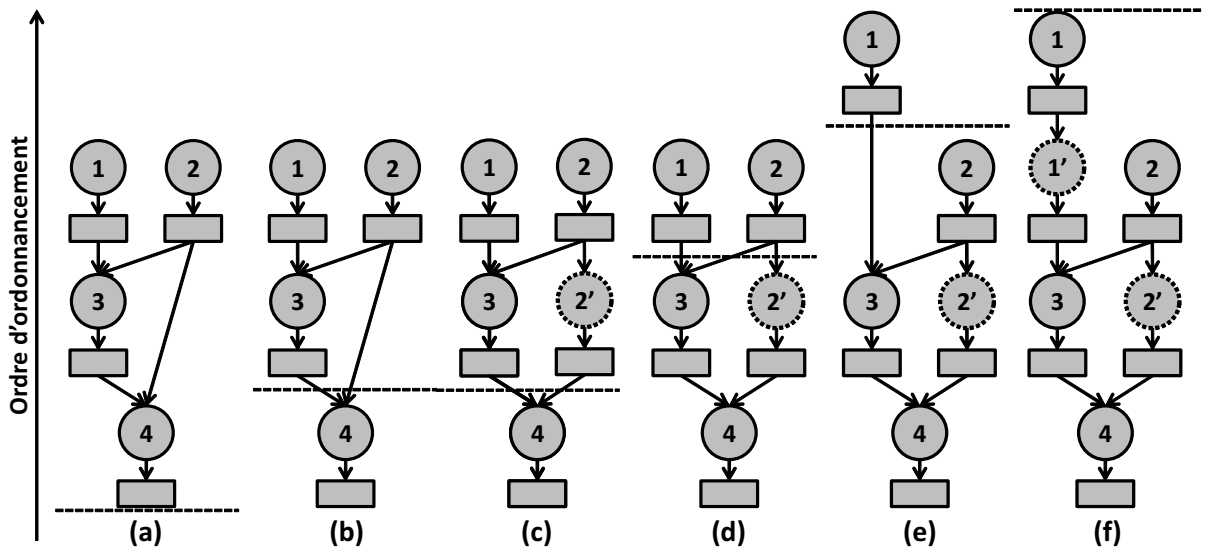


Figure 3.7 – Exemple illustratif du processus d'ordonnancement sur un CGRA ne possédant qu'une seule tuile. La ligne en pointillées délimite les nœuds ordonnancés (en dessous) des nœuds non-ordonnancés. **(a)** DFG initial. **(b)** Après ordonnancement du nœud 4. **(c)** Après ajout de la mémorisation pour le nœud 2 qui n'est pas ordonnançable. **(d)** Après ordonnancement des nœuds 3 et 2'. **(e)** Après ordonnancement du nœud 2. **(f)** DFG final.

Le choix d'un ordonnancement arrière permet d'appliquer toutes ces différentes transformations de graphe. Avec un ordonnancement avant (parcourant les nœuds en suivant un tri topologique), la seule transformation pertinente est le routage simple car il n'est pas possible de connaître les opérations filles qui seront réellement ordonnancées au cycle suivant. La répartition des arcs sortants entre les deux nœuds serait alors purement arbitraire et ne permettrait pas d'obtenir un véritable gain par rapport au routage simple.

Autre cas d'utilisation

À la ligne 8 de l'algorithme, la fonction AjoutMemPourNoeudsNonOrdonnancables() est exécutée. Son but est d'ajouter des nœuds de mémorisation dans le cas particulier d'une opération dont le résultat a été utilisée mais qui n'est pas ordonnançable car d'autres successeurs n'ont pas encore été traités. Les mémorisations qu'elle intègre au graphe sont ajoutées à la liste des nœuds ordonnançables.

3.2.2.3 Exemples illustratifs

Exemple simple

De manière à mieux visualiser le fonctionnement de l'algorithme d'ordonnancement/assignation, cette sous-section décrit un exemple pas à pas de ce processus pour le DFG de la figure 3.7(a) sur un CGRA ne possédant qu'une seule tuile. L'intérêt de cet exemple est qu'il permet déjà d'utiliser les différentes étapes de l'algorithme (ordonnancement, assignation et transformation). La ligne en pointillées délimite les nœuds déjà ordonnancés et placés (en dessous) des nœuds non-ordonnancés.

1. Tout d'abord, comme l'ordonnancement s'effectue en arrière, le seul nœud ordonnançable au premier cycle est le nœud 4. Le CGRA possède un opérateur et un registre compatibles, donc ces ressources seront affectées aux nœuds 4 et ils sont alors retirés de la liste des nœuds à ordonnancer. Le résultat est donné en (b).
2. Au cycle suivant, un seul nœud est ordonnançable : le 3. Mais le nœud 4 utilise un résultat

fourni par le nœud 2 qui n'est pas ordonnançable. Il faut donc ajouter une mémorisation (nœud 2') et l'ajouter à la liste des nœuds à ordonnancer et des nœuds ordonnançables. Cette étape est illustrée en (c).

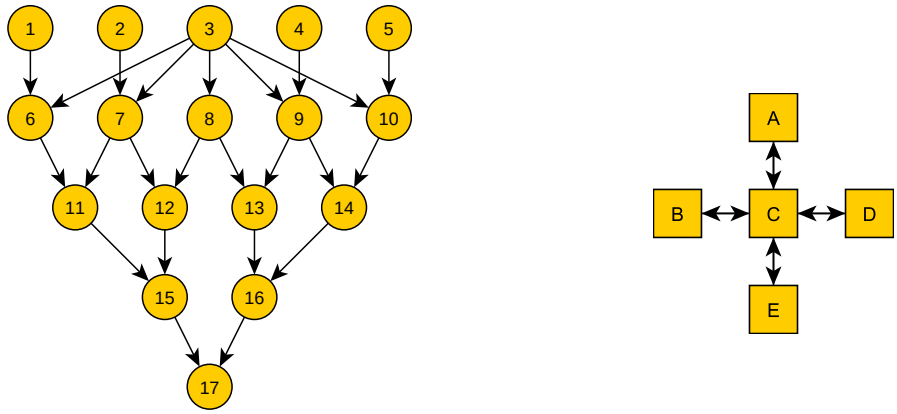
3. Toujours dans le même cycle, le nœud 3 ne peut utiliser que l'opérateur classique de la tuile. Le nœud 2', lui peut utiliser l'opérateur classique et un opérateur de mémorisation d'un des registres de la RF (on suppose que la RF possède plus d'un registre). Le nœud 3 est donc assigné à l'opérateur de calcul et le nœud 2' sur l'opérateur de mémorisation. Il n'y a plus d'autre nœud à ordonnancer à ce cycle ce qui amène à l'illustration (d).
4. Au cycle suivant deux derniers nœuds sont ordonnançables. Le nœud 2 possède plus d'arcs sortants, il est donc prioritaire et va être assigné en premier (figure 3.7(e)).
5. Le nœud 1 ne peut pas être placé par manque de ressource. Il est donc routé. Le nœud 1' est ajouté à la liste des opérations à ordonnancer et est ensuite assigné dans un opérateur de mémorisation d'un registre de la RF. Enfin, au cycle suivant le nœud 1 est ordonnancé et assigné. Le DFG en (f) est celui obtenu à la fin du processus.

Exemple problématique du chapitre 2

Dans la section 2.5.1, un exemple justifiant qu'il ne fallait pas séparer l'ordonnancement et l'assignation pour pouvoir obtenir un *mapping* sur certain réseau a été présenté. Il s'agit de l'exemple rappelé dans la figure 3.8. La figure 3.9 présente le graphe résultant des différentes transformations ainsi qu'une des solutions de *mapping*.

Nous présentons maintenant pas à pas un des *mappings* qu'obtient notre flot. Dans cet exemple, il commence par ordonnancer et assigner l'opération 17, par exemple sur la tuile C, mais les quatre autres possibilités d'assignation donnent aussi des solutions. Au deuxième cycle, les opérations 15 et 16 sont ordonnancées et assignées sur les tuiles B et C (cf. figure 3.9a). Au troisième cycle, l'opération 15 ayant deux parents (11 et 12), l'un d'eux doit être assigné sur la tuile B et l'autre sur la tuile C. Pour l'exemple, ce sera l'opération 11 qui sera assignée sur la tuile B et l'opération 12 assignée sur la tuile C. L'opération 16 étant sur la tuile C, ses parents peuvent être sur les tuiles périphériques, par exemple 13 sur D et 14 sur E.

Le quatrième cycle est plus complexe. Trois opérations (7, 8 et 9) ont une priorité plus grande que les autres (6 et 10). L'algorithme les traite donc en premier. L'opération 7 doit avoir accès aux opérations 11 et 12 et donc peut être placée sur la tuile B ou C et nous ne présentons que le cas où elle est assignée sur la tuile B. De même l'opération 8 peut être assignée sur la tuile C ou D. Les possibilités d'assignation de l'opération 9 dépendent de l'assignation de l'opération 8. Si l'opération 8 est assignée sur la tuile C, alors l'opération 9 ne peut pas communiquer avec ses deux enfants. En revanche, si l'opération 8 est assignée sur la tuile D, alors l'opération 9, en

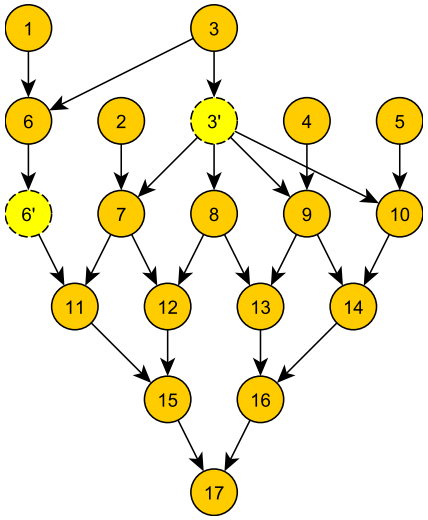


(a) Graphe d'opérations. (b) Réseau cible.

FIGURE 3.8 – Exemple problématique d'un ordonnancement séparé du placement.

Cycle	Numéro de tuile				
	A	B	C	D	E
1		1	3		
2		6	2 + 3’	4	5
3		7 + 6’	9	8	10
4		11	12	13	14
5		15	16		
6			17		

(a) *Mapping* résulant.



(b) Graphe d’opérations transformé par le flot.

Figure 3.9 – Une solution de l’exemple problématique de la figure 3.8. Les nœuds de variables, indispensables pour l’algorithme, ne sont pas représentés ici pour ne pas alourdir la figure.

se plaçant sur la tuile C peut communiquer avec ses deux enfants. L’algorithme étant exhaustif pour l’assignation, cette solution sera trouvée. Pour le reste des opérations, l’opération 10 sera assignée sur la tuile E et l’opération 6 ne peut pas être assignée à ce cycle car la seule tuile libre restante (la tuile A) ne communique pas avec la tuile B. Le graphe est donc transformé par routage simple. L’opération 6’ résultante est assignée sur la tuile B.

Au début du cinquième cycle, l’algorithme détermine que l’opération 3 n’est pas ordonnançable mais possède des enfants déjà assignés. Il va donc effectuer du routage simple pour « relier » ses enfants à l’opération 3 qui ne pourra être ordonnancée qu’au cycle suivant. Une mémorisation 3’ est donc ajoutée et placée sur la tuile C (seule possibilité pour relier tous ces successeurs. L’opération 6 est ensuite prioritaire car elle a une mobilité nulle. Elle est donc placée sur la tuile B. Puis l’opération 2 est assignée sur la seule tuile lui permettant de respecter sa dépendance de donnée : la tuile C. L’opération 4 peut être assignée sur la tuile D ou A ; et l’opération 5 sur uniquement la tuile E. Enfin, au sixième cycle, l’opération 3 est placée sur la tuile C et l’opération 1 sur la tuile B.

Le *mapping* résultant possède donc 6 cycles, soit un de plus que l’ASAP qui n’est pas atteignable pour ce réseau. On remarquera que le *mapping* de la figure 3.9a n’utilise pas la tuile A, mais aurait pu si l’opération 4 n’était pas sur la tuile D, mais sur la tuile A.

3.2.2.4 Élagage exact

Comme l’algorithme de placement est exhaustif, il permet de s’assurer que si une solution existe pour le nœud considéré, elle sera trouvée. Mais il pose des problèmes de passage à l’échelle. Pour limiter l’explosion de *mappings* partiels, nous avons dû introduire une étape permettant de supprimer les solutions « redondantes ». Cette étape s’exécute uniquement lorsque tous les nœuds ordonnançables ont été *mappés*, c’est-à-dire à la fin de l’ordonnancement pour le cycle courant (ligne 23 de l’algorithme 2).

On définit alors un *mapping* redondant comme étant un *mapping* qui utilise exactement les mêmes opérateurs pour effectuer les mêmes opérations à un cycle donné qu’importe les cycles précédents. On ne tient compte que des opérateurs (classique et de mémorisation). Le placement des variables en registres n’importe pas. La figure 3.10 illustre ce phénomène. Dans cette figure, au cycles $i + 1$ et $i + 2$ les opérateurs utilisés sont différents et il fallait donc conserver les deux variantes. En revanche, au cycle i, les opérateurs A et B sont respectivement utilisés pour

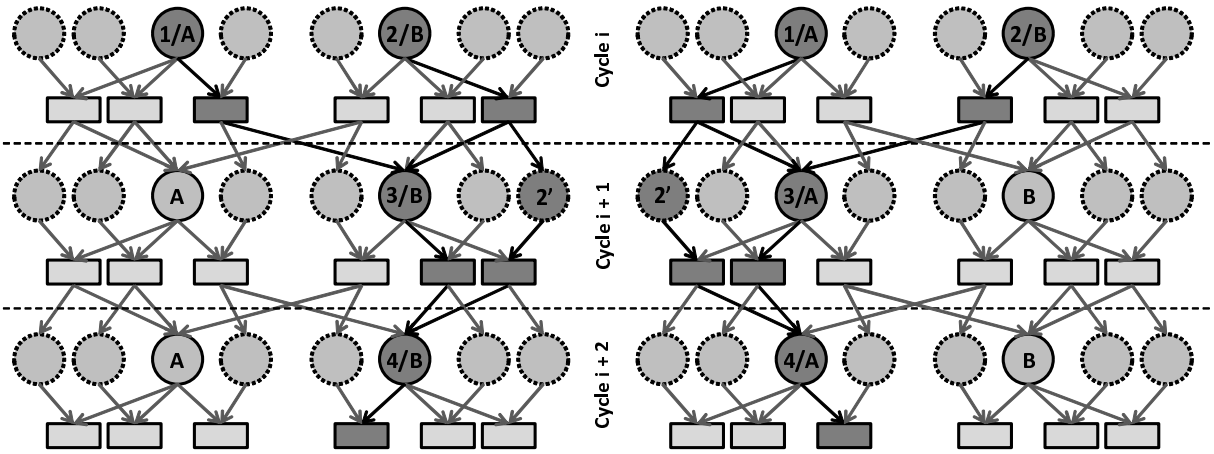


FIGURE 3.10 – Illustration de deux *mappings* redondants au cycle i sur une architecture avec deux tuiles et deux registres par RF. L'utilisation des ressources est en gris foncé.

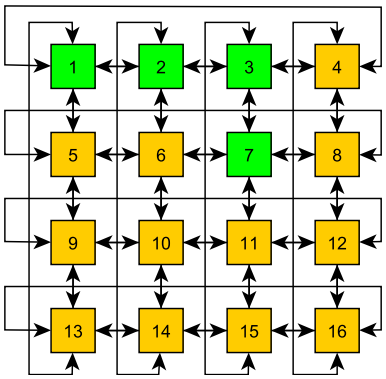
les opérations 1 et 2 dans les deux cas. Ils sont donc redondants car les nœuds qui seront ordonnancés dans la suite du processus de projection « verront » la même base dans les deux cas, ce qui produira exactement les mêmes solutions.

L'étape d'élagage sert à supprimer tous les *mappings* redondants pour ne conserver qu'un seul exemplaire de chaque utilisation. Du fait du contexte de tolérance aux fautes, nous avons décidé de conserver la variante qui offre la plus petite utilisation globale de tuiles. Par exemple si au cycle considéré seules les tuiles A, B et C sont utilisées et qu'une des variantes utilise dans un cycle précédent en plus la tuile D alors que l'autre non, cette variante sera supprimée. Par contre, si un *mapping* utilise les tuiles A, B, C et D, et qu'un autre *mapping* utilise les tuiles A, B, C et E et qu'il n'existe pas de solution n'utilisant que les tuiles A, B et C, alors les deux variantes seront conservées à ce cycle. Cette redondance sera probablement supprimée aux cycles suivants en fonction des réutilisations des tuiles.

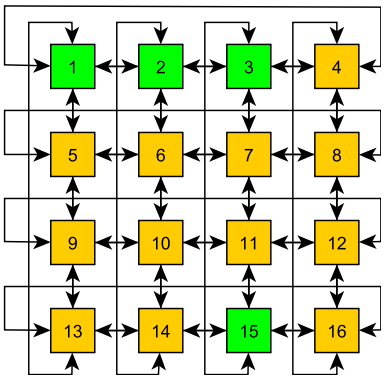
Cette étape d'élagage préserve donc la diversité des *mappings* finaux tout en limitant leur nombre. C'est pour cette raison que ce flot est malgré tout qualifiable de semi-exhaustif dans son exploration de l'espace de solutions. L'inconvénient de cet élagage, qui sera discuté dans la section suivante, est qu'il possède une complexité en $O(n^2)$.

3.2.2.5 Virtualisation des mappings

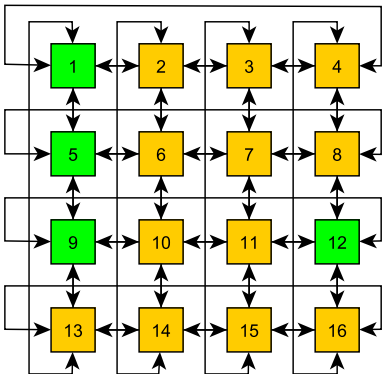
Comme l'étape d'élagage, l'étape facultative de notre flot permet de réduire le nombre de solutions partielles tout en préservant l'exhaustivité du placement. Elle consiste à tirer partie de la régularité du CGRA et dépend donc grandement de son architecture que ce soit au niveau des tuiles ou de l'interconnexion. Concrètement, cette étape va limiter les redondances dues aux symétries, aux rotations et aux invariances par translation lorsque c'est possible au moment de la détermination des opérateurs compatibles (algorithme 3, ligne 3-4). Le cas le plus favorable est celui d'un CGRA homogène (toutes les tuiles sont identiques), possédant un réseau d'interconnexion torique (sans effet de bord) et dont la largeur égale la hauteur comme dans la figure 3.11. Dans ce cas, il est possible, sans perdre de généralité, de fixer la position du premier nœud sur l'opérateur d'une seule tuile car toutes les autres positions seront déterminables par translation grâce à l'aspect torique du réseau d'interconnexion et de l'homogénéité des tuiles. L'utilisation de la figure 3.11d est incluse dans celle de la figure 3.11a par translation. Avec seulement un CGRA torique et hétérogène mais régulier, il est possible de restreindre les positions de départ à simplement une part type de tuiles et par voisinage de tuiles. Ensuite, si les dimensions du CGRA sont égales, il est possible de ne pas conserver pour le couple « premier nœud deuxième



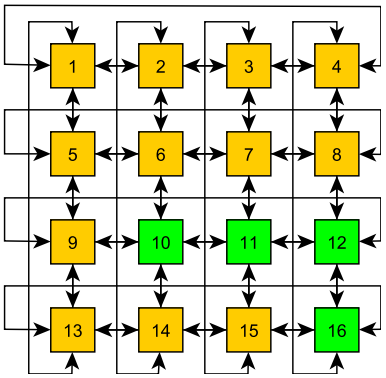
(a) Utilisation originale.



(b) Virtualisée par une symétrie horizontale.



(c) Virtualisée par une rotation horaire.



(d) Virtualisée par une translation.

Figure 3.11 – Exemples d'utilisations virtualisées avec le cas le plus favorable.

nœud »les positions se situant par exemple dans le quartier droit supérieur de l'architecture. En effet, les autres positions pourront être déduites par rotation autour du premier nœud comme celle de la figure 3.11c. D'ailleurs, tant qu'il n'y a qu'une seule tuile utilisée il est possible de considérer les rotations (si le deuxième nœud utilise le même opérateur que le premier). Le troisième niveau consiste à considérer les symétries axiales. Il est ainsi possible de ne pas conserver certaines positions du troisième nœud si il existe la position symétrique par rapport à l'axe formé par les deux premiers nœuds. Pour cela, il faut que les deux premiers nœuds soient sur une même ligne, une même colonne ou une même diagonale. D'ailleurs, si le troisième nœud est aussi sur l'axe de symétrie, cette étape s'applique au quatrième nœud et ainsi de suite tant que les utilisations sont alignées comme avec la figure 3.11b.

Le gain apporté par cette virtualisation peut s'avérer très important. Prenons par exemple un CGRA homogène, possédant un mesh-2D torique et de dimension 4×4. Le premier niveau de virtualisation diminuera le nombre de solutions partielles d'un facteur égal au nombre de tuiles soit 16. Le deuxième niveau permet encore de diviser par un facteur 4 le nombre de solutions partielles. Soit un facteur global de 64 si les deux opérations sont ordonnancées au même cycle. Et le troisième niveau permet, dans ce cas précis, avec un CGRA 4×4, de diviser par 2 le nombre de solutions partielles pour la troisième utilisation de tuile dans $\frac{4}{5}^{ème}$ des cas (les cas où les deux premières utilisations forment un axe de symétrie). Dans le cas d'un CGRA de taille supérieure, le gain est encore plus grand.

3.2.3 Évaluation

Cette section décrit le protocole suivi pour évaluer le flot et présente les résultats obtenus.

3.2.3.1 Protocole expérimental

Notre méthode est intégralement automatisée et implémentée en Java en utilisant l'environnement de développement *Eclipse Modeling Framework* (EMF). Les CDFGs sont générés à partir des codes C intégralement déroulés en utilisant le compilateur GCC-4.7.2. Neuf algorithmes issus du traitement du signal ont été utilisés pour les expériences, à savoir : une transformée en cosinus discret deux dimensions ou *Discrete Cosine Transform* (DCT), un produit de matrice, une transformée de Fourier Rapide ou FFT, un calcul de la distance de Manhattan, un filtre à Moyenne Glissante Exponentielle ou *Exponential Moving Average* (EMA), une Déconvolution à Fenêtre Glissante ou *Moving Window Deconvolution* (MWD), un filtre trapézoïdal, un masque flou (*unsharp mask*) et un filtre passe-bas (DC-Filter). Les tests ont été réalisés sur un PC possédant un processeur Intel Xeon et 8 GByte de mémoire vive. Pour obtenir une large gamme de résultats, nous avons fait varier plusieurs contraintes architecturales : la taille du CGRA (3×3 et 4×4), le nombre de registres dans la RF (4 et 8) et le nombre de tuiles que les *mappings* peut d'utiliser (1, 2, 3 et 4 tuiles) ; soit seize jeux de contraintes par code applicatif et par méthode.

L'approche proposée est comparée à deux autres approches de l'état de l'art. La première, « Méthode 1 », consiste à résoudre l'ordonnancement et l'assignation séparément comme dans la première étape de [Lee et al., 2011]. Elle utilise un « *list-scheduling* » parcourant les nœuds du DFG triés par un tri topologique simple pour les ordonnancer puis l'algorithme de Levi pour l'assignation. Le DFG ne peut être transformé que pendant la phase d'ordonnancement en utilisant du re-routage comme expliqué dans le paragraphe 3.2.2.2. La deuxième, « Méthode 2 », réalise des transformations statiques du DFG, *i.e. a priori* pour réaliser l'ordonnancement et essaie de trouver des *mappings* en utilisant l'algorithme de Levi comme proposé dans [Hamzeh et al., 2012] et [Hamzeh et al., 2013]. Ces deux méthodes ont montré qu'elles donnent de meilleurs résultats que [Mei et al., 2002] et [De Sutter et al., 2008], nous ne nous comparerons donc pas à ces méthodes. Quatre métriques sont considérées pour déterminer la qualité des différentes approches :

1. le taux de succès : défini comme étant le pourcentage de fois qu'une méthode trouve une solution quand au moins une des trois méthodes a trouvé une solution (il ne s'agit pas du taux de succès absolu car selon les contraintes, il n'existe parfois pas de solution : *e.g.* une FFT sur une seule tuile avec seulement 4 registres dans la RF) ;
2. le taux d'obtention de la meilleure latence : défini comme étant le pourcentage de fois qu'une méthode trouve la meilleure latence (ou égale) parmi les trois méthodes ;
3. la diversité : définie comme étant le nombre de *mappings* différents obtenus, c'est-à-dire la capacité d'une méthode à explorer largement l'espace des solutions. Deux *mappings* sont différents l'un de l'autre s'ils n'utilisent pas les mêmes tuiles et/ou s'ils utilisent le réseau d'interconnexion d'une manière différente (il ne s'agit pas de la même notion que la redondance de *mappings* utilisé dans la section 3.2.2.4) ;
4. l'efficacité de l'exploration : définie comme étant le débit de *mappings* différents c'est-à-dire le nombre de *mappings* différents générés par unité de temps.

Dans le contexte de multi-projection où il faut obtenir le plus de *mappings* différents, ces deux dernières métriques donnent une indication particulièrement pertinente sur la capacité d'une méthode à transformer le graphe de l'application « juste comme il faut ». En effet, étant donné le nombre limité de ressources d'un CGRA, l'augmentation du nombre de nœuds du DFG implique une diminution du nombre de possibilités de les placer sur l'architecture. De ce fait, si la méthode ajoute plus de nœuds au graphe que nécessaire, le nombre de solutions partielles que l'algorithme de Levi est capable d'obtenir diminue. De plus, le débit de *mappings* différents permet de savoir si la recherche est efficace ou si l'algorithme perd son temps à essayer des placements qui ne donneront pas d'autres solutions.

3.2.3.2 Résultats

Les figures 3.12 à 3.15 présentent, pour les différentes métriques, les résultats obtenus pour chacun des codes d'application considérés.

Dans la figure 3.12, on observe que résoudre les problèmes d'ordonnancement et d'assignation séparément, comme le fait la méthode 1, conduit à un faible taux de succès (environ 37%). Ceci justifie l'utilisation d'un ordonnancement arrière et d'une assignation simultanée. Le fait d'effectuer des transformations de graphes statiques, comme la méthode 2, augmente le taux de succès (environ 62%), mais ne donne pas d'aussi bons résultats que notre méthode (environ 99%). Les faibles taux de succès des méthodes 1 et 2 sont dus en grande partie à l'aspect glouton de l'algorithme d'ordonnancement qui, lorsqu'il parcourt les nœuds par un tri topologique, engorge le réseau d'interconnexion du CGRA, empêchant toute assignation.

Concernant le taux d'obtention de la meilleure latence, comme la méthode proposée se base en partie sur une heuristique (pour l'ordonnancement), il est normal qu'elle ne trouve pas toujours la meilleure latence (comme pour le filtre passe-bas). Cependant, comme illustré dans la figure 3.13, elle obtient plus de deux fois plus souvent la meilleure latence parmi les trois méthodes testées, soit environ 90% des cas. Quand elle ne trouve pas la meilleure latence, elle l'augmente en moyenne de 1,5 cycles, soit une augmentation d'environ 15% de ces latences.

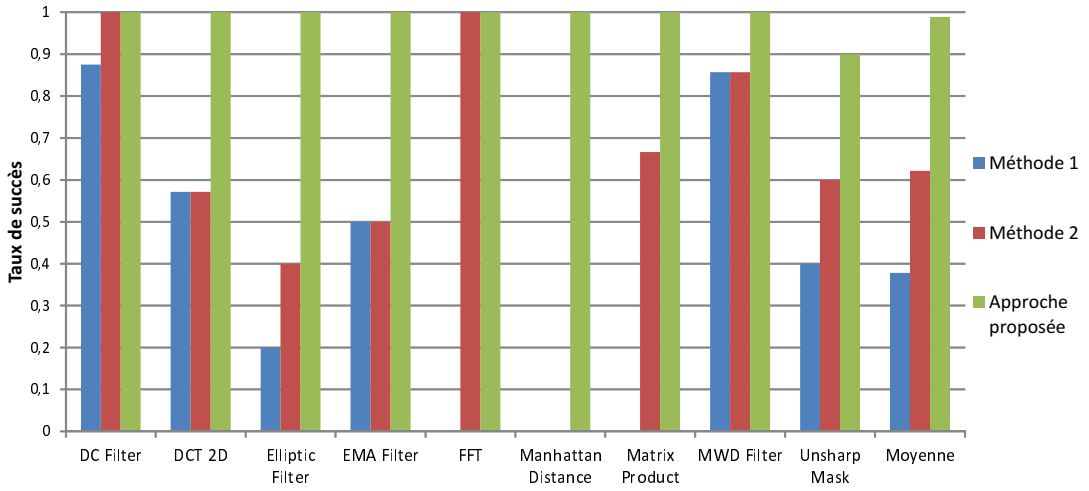


Figure 3.12 – Taux de succès.

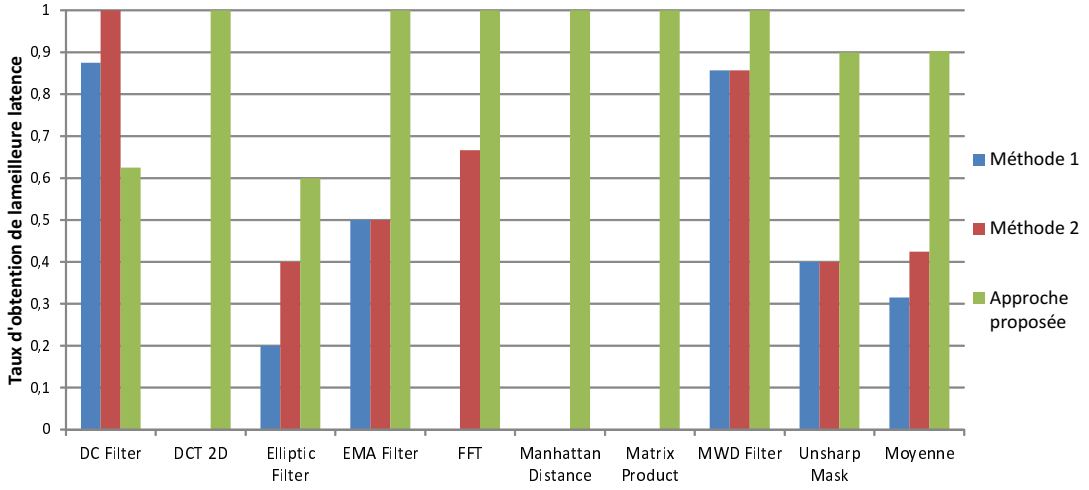


Figure 3.13 – Taux d'obtention de la meilleure latence.

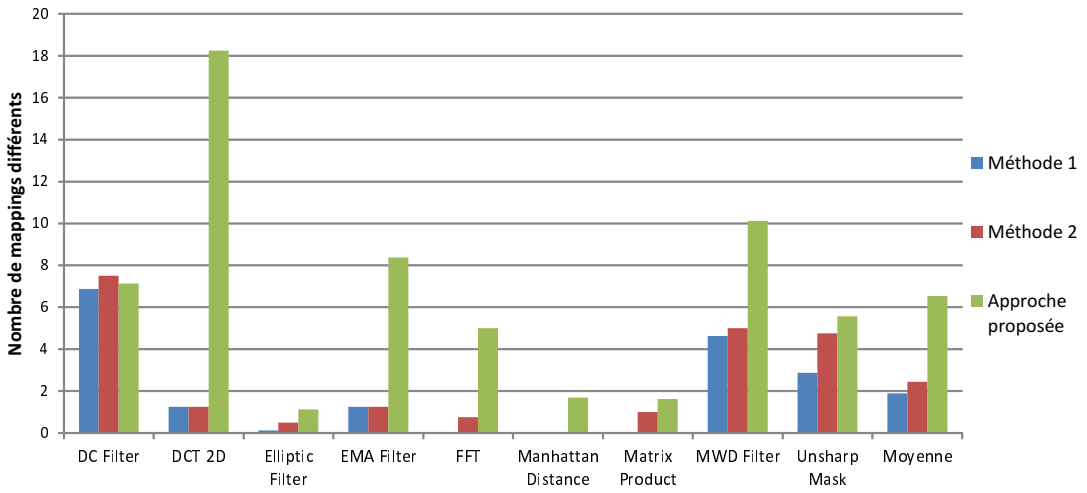


FIGURE 3.14 – Nombre moyen de *mappings* différents.

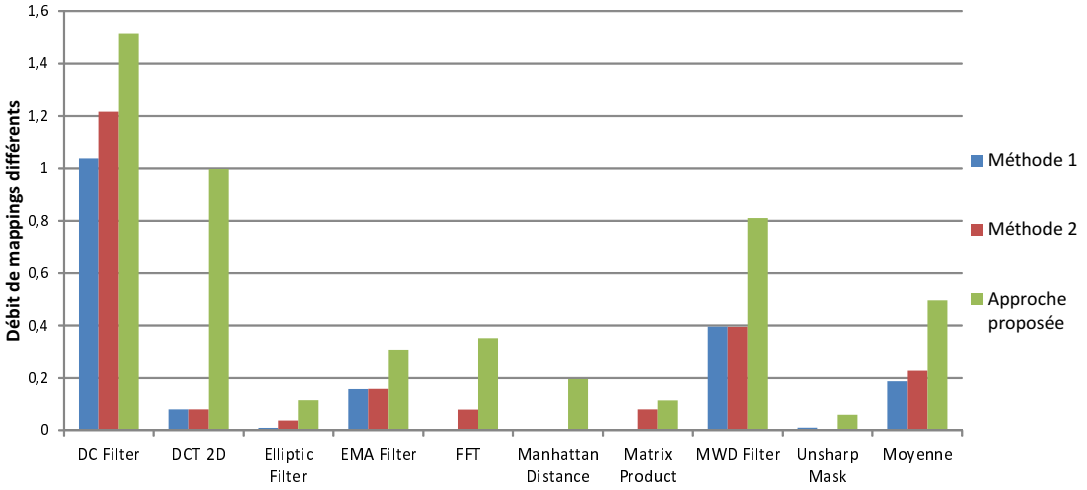


FIGURE 3.15 – Débit moyen de *mappings* différents.

La figure 3.14 montre que l'approche proposée permet d'obtenir en moyenne un nombre plus important de *mappings* différents que les méthodes 1 et 2 (respectivement 3,7 et 2,4 fois plus grand) ce qui est l'objectif de notre flot.

De plus, la figure 3.15 illustre le fait que l'exploration est plus efficace car le temps passé par *mapping* est plus faible comparativement aux autres méthodes (respectivement d'un facteur 2,6 et d'un facteur 2,2).

3.2.4 Bilan

Nous avons présenté dans cette section une première version d'un flot permettant d'obtenir directement plusieurs solutions de *mappings* différentes. Il se base sur deux modèles formels très versatiles. Notre modèle de DFG ajoute l'opération de mémorisation à celles classiquement utilisées et impose l'explicitation des variables. Le modèle de l'architecture permet de représenter des architectures aux caractéristiques très variées :

— possédant des tuiles homogènes ou hétérogènes ;
— possédant ou non une RF ;
— dont la taille de la RF peut varier d'une tuile à l'autre ;
— une ou plusieurs RFs partagées entre plusieurs tuiles : il s'agit alors simplement d'une tuile

« classique » mais ne possédant pas d'opérateur de calcul, mais à la place un opérateur de mémorisation, et dont les sorties des registres sont reliées aux entrées des opérateurs de calcul des autres tuiles ;
— un réseau d'interconnexion régulier ou spécifique.

Le fait de ne pas représenter les multiplexeurs dans le modèle de l'architecture a des avantages et des inconvénients :

— le premier avantage est de ne pas avoir besoin de rajouter des nœuds « multiplexeurs un-vers-un » dans le modèle de l'application. En effet, pour que les deux modèles soient homomorphes, si le modèle d'architecture possède des multiplexeurs, il faudrait le pendant dans le modèle de l'application et donc les ajouter dans le modèle de DFG. Ainsi, le nombre de nœuds à traiter est moindre, limitant la complexité de l'algorithme de Levi ;
— l'inconvénient principal est que le contrôle du nombre de sorties utilisées des RFs doit être fait par le flot de projection augmentant légèrement sa complexité. Néanmoins, s'il n'y a pas de RF partagée entre les tuiles, mais uniquement des RFs internes, comme les opérateurs ont généralement au plus deux entrées, il n'y aura qu'au plus deux sorties utilisées par RF ce qui est un fonctionnement très classique (RF à une entrée et deux sorties).

L'utilisation d'un ordonnancement arrière et d'une assignation simultanée permet de transformer le graphe de l'application si besoin et dynamiquement sans remettre en cause les nœuds déjà traités. Les expériences ont montré que ce flot permet d'obtenir plus de *mappings* différents que d'autres approches, justifiant sa pertinence.

Limitations du flot

Le protocole expérimental qui a été suivi ne fait varier les contraintes que jusqu'à quatre tuiles sur le CGRA. La raison est que sur le PC utilisé, l'implémentation Java ne permettait pas d'utiliser plus de quelques Gigaoctets de mémoire vive. Or, avec une contrainte de 5 tuiles, le nombre de *mappings* partiels générés est supérieur à plusieurs dizaines de millions. De même, si le DFG de l'application comporte plus de quelques dizaines d'opérations, le nombre de *mappings* partiels dépasse les capacité du programme. Bien que, comme nous le verrons dans la chapitre suivant, les CDFGs ne possèdent en moyenne que quelques dizaines de nœuds d'opérations par bloc et donc pourrait fonctionner, il existe une véritable difficulté de passage à l'échelle en termes de nombre de nœuds et de taille de l'architecture considérée. L'étape d'élagage est indispensable (sans elle, le programme n'arrive pas à se terminer dès que le DFG contient plus d'une quinzaine d'opérations sur un CGRA 3×3), mais cet élagage est assez coûteux en termes de temps car il nécessite d'effectuer des comparaisons entre les *mappings* partiels. Nous verrons dans la section suivante (section 3.3) comment améliorer les performances du flot en changeant l'étape d'élagage et ainsi réussir à passer suffisamment à l'échelle pour pouvoir projeter des codes possédant des centaines de nœuds.

3.3 Méthode stochastique

Comme nous l'avons vu dans la section précédente, la première version du flot a des difficultés à passer à l'échelle en termes de nombre de nœuds, que cela soit le nombre de nœuds du DFG ou la taille du CGRA. Cette partie présente une évolution permettant pour des applications comportant un grand nombre de nœuds ou un grand nombre de ressources d'avoir des temps de d'exécution du flot qui ne soient pas rédhibitoires.

3.3.1 Élagage aléatoire et pseudo aléatoire

Le point problématique, que l'on utilise ou non l'étape d'élagage dans le flot précédent, est le nombre de *mappings* partiels manipulés. En effet, si l'étape d'élagage n'est pas utilisée, le nombre de possibilités données par l'algorithme de Levi est combinatoire (bien que limité par les dépendances de données). Cette explosion va nécessairement, lors de l'exécution du programme sur une plateforme réelle, finir par saturer la mémoire vive et entraîner ce que l'on appelle du *swapping*, c'est-à-dire le déplacement de données depuis la mémoire vive vers le disque dur et inversement.

Si l'on est dans le cas où l'étape d'élagage est utilisée, alors le nombre de solutions partielles croit nettement moins vite (en particulier si la « virtualisation » des *mappings* est possible). Mais dans ce cas, c'est le temps d'élagage qui devient rédhibitoire. En effet, à la fin de chaque cycle d'ordonnancement, les solutions redondantes sont supprimées. La détermination de la redondance nécessite un très grand nombre de comparaisons. Sans optimisation particulière, pour n éléments, il faut faire n^2 comparaisons. Il est possible de réduire le nombre de comparaisons nécessaires jusqu'à au mieux de l'ordre de $O(n \log_2(n))$ (complexité optimale pour des algorithmes de tri comme le *quick sort*), par exemple en utilisant du « diviser pour régner » ou en classant les *mappings* partiels dans une structure d'arbre comme énoncé dans la section 3.2.2.4.

Le tableau 3.1 donne le pourcentage du temps total passé dans l'étape d'élagage pour plusieurs tailles de CGRA et pour différents codes applicatifs. On observe qu'en général plus on augmente la taille de l'architecture et plus ce temps devient prédominant. Dans certains cas il est même écrasant ($> 90\%$). De plus l'évolution n'est absolument pas linéaire. Pour le calcul de la distance de Manhattan par exemple, sur un CGRA 3×3 avec 4 tuiles autorisées pour la projection, ce temps s'élève à 0,74% alors qu'il attend près de 50% dans le cas 4×4.

Tableau 3.1 – Pourcentage du temps d'exécution passé dans l'étape d'élagage pour des CGRAs 3×3 et 4×4 toriques avec des RFs de 8 registres.

Taille CGRA	Nom du code	Contrainte de nombre de tuiles			
		1	2	3	4
3×3	DC Filter	0	0	0,15	7,33
	DCT 2D	0	0	0,07	5,15
	EMA Filter	0	0	0,83	47,45
	Manhattan Distance	0	0	0,30	0,74
	Matrix Product	0,01	0,04	43,47	24,25
	MWD Filter	0,01	0,02	0,10	5,87
	Unsharp Mask	0	0,43	85,42	90,28
4×4	DC Filter	0,01	0,01	0,49	21,93
	DCT 2D	0	0,02	31,17	92
	EMA Filter	0	0	2,04	93,28
	Manhattan Distance	0	0	18,62	46
	Matrix Product	0	0,01	13,54	86,53
	MWD Filter	0	0	4,71	85,51
	Unsharp Mask	0	0,38	17,74	92,64

Pour remédier à ce point problématique, nous avons décidé de changer la façon par laquelle les *mappings* sont élagués. Au lieu d'utiliser un élagage conservant l'exhaustivité du résultat, nous avons introduit une heuristique de sélection. L'idée est de ne conserver qu'un nombre « raisonnable » de *mappings* partiels. Pour cela, après l'obtention de l'ensemble des solutions de placements pour un nœud (et non pas à la fin de chaque cycle comme avec la précédente version de l'élagage), l'heuristique s'exécute pour déterminer quels sont les *mappings* partiels conservés pour la prochaine étape. Cette heuristique est basée sur un tirage aléatoire ce qui garantit un traitement équitable des différents *mappings* partiels. Un seuil est défini et pour chaque *mapping* partiel, un tirage aléatoire donne une valeur. Si elle est inférieure ou égale au seuil, alors cette solution partielle est conservée. Sinon, le *mapping* partiel correspondant sera supprimé.

Nous avons étudié plusieurs variantes qui diffèrent par leurs manières de calculer le seuil. Les paragraphes suivants présentent certaines des grandes fonctions de seuillage possibles et leurs impacts sur les résultats. Dans le paragraphe 3.3.1.4, il sera aussi discuté de l'opportunité d'un élagage qui ne soit pas aléatoire.

3.3.1.1 Seuillage à fonction constante

La première variante de l'élagage aléatoire consiste à utiliser un seuil fixe (*e.g.* 50% pour n'en conserver qu'un sur deux). Mais cette solution s'est révélée « instable » :

— elle peut supprimer trop de *mappings* partiels en tout début de processus, voire tous quand la virtualisation est utilisée, entraînant un échec après seulement quelques nœuds si le seuil est trop bas ;
— elle peut ne pas supprimer suffisamment de *mappings* si le seuil est trop haut entraînant soit le crash de l'application soit un temps de calcul rédhibitoire.

Le problème étant que la frontière entre les deux est très mince voire inexistante en fonction du nombre de nœuds de l'application et la taille de l'architecture. Par exemple un seuil qui permet sept fois sur dix de terminer le *mapping* pour la DCT 2D pour une architecture 3×3 avec quatre registres dans la RF et une contrainte de quatre tuiles utilisables, n'a permis de trouver de solutions qu'une fois sur dix avec la même architecture mais une contrainte de six tuiles utilisables. Les raisons des échecs étaient les suivantes : deux échecs par suppression de tous les *mappings* et pour les autres c'est le *timeout* qui s'est déclenché.

Nous n'avons pas trouvé de moyen simple de stabiliser cette variante. Nous avons donc essayé une autre version mais qui permettait de ne supprimer des *mappings* que lorsque ce nombre était supérieur à une valeur définie par l'utilisateur (*e.g.* 5 000). Cette variante donnait de bien meilleurs résultats car elle n'échouait pas par excès de suppressions. Cependant, elle ne permettait pas de résoudre le problème quand le nombre de *mappings* partiels devenait important. À titre d'illustration, supposons que l'architecture soit un CGRA 4×4 possédant un mesh-torique. Nous sommes au premier nœud d'un nouveau cycle et il y a 5 000 solutions partielles. L'opération courante doit fournir son résultat à seulement une autre opération. Comme il n'y a que cette contrainte pour le placement, pour chaque solution partielle précédente, cette opération aura cinq solutions (la tuile où est mappée l'opération fille et ses quatre voisines). Il y aura donc 25 000 solutions partielles. Si on conserve plus d'un *mapping* sur cinq, alors le nombre de solutions partielles pourra exploser. C'est même bien pire si le nœud est une opération *store* qui n'a aucune contrainte d'assignation (pas de successeur). Dans cet exemple, le nombre de *mappings* passerait à 80 000. À l'inverse, si on ne conserve qu'un *mapping* sur dix et si il y a juste 5 001 *mappings* partiels, alors il n'en restera plus que environ 500. Cela qui peut s'avérer insuffisant pour permettre de trouver des solutions pour tous les nœuds surtout en fin de cycle quand l'architecture est proche de la saturation. Ce n'est donc pas l'approche que nous avons retenue.

3.3.1.2 Seuillage à fonction exponentielle décroissante

Face à cet effet de seuil, nous avons testé un seuil qui s'adapte en fonction du nombre de solutions partielles trouvées. L'idée est de conserver un grand nombre de *mappings* quand il y en a peu, et d'en supprimer plus quand il y en a beaucoup. Mathématiquement, la fonction de seuil doit donc respecter les propriétés suivantes :

— elle doit être décroissante par rapport au nombre de *mappings* partiels obtenus et plus précisément par rapport au quotient entre ce nombre et un nombre de référence défini par l'utilisateur ;
— elle doit valoir 1 quand le nombre de *mappings* est nul.

La fonction exponentielle décroissante est classiquement utilisée dans les recuits simulés pour autoriser ou non la conservation d'une solution. En poursuivant l'analogie, le nombre de *mappings* partiels correspond à la température actuelle et le nombre de référence est la température de référence. Nous avons mis en œuvre et testé deux variantes de cette fonction exponentielle décroissante.

Version « linéaire »
La première est une version simple qui répond à l'équation 3.3.1.

$$Seuil = \exp\left(\frac{-NbMappings}{NbReference}\right) \tag{3.3.1}$$

Son évolution est donnée en figure 3.16a. On observe que lorsque le nombre de *mappings* partiels vaut le nombre de référence, la probabilité de conserver une solution est de $\exp(-1) \approx 0,37$.

Cette version pour calculer le seuil est beaucoup plus efficace pour ne pas avoir trop de *mappings* partiels que la version avec un seuil constant. Cependant, elle peut très facilement quasiment supprimer l'intégralité des *mappings* en un seul coup et par conséquent est assez instable. Par exemple, supposons qu'il y ait 1 800 *mappings* partiels à l'origine. Supposons que le nœud que l'on traite n'ait pas de lien avec les autres nœuds (par exemple un nœud *store*). Il aura donc pour chaque solution partielle précédemment trouvée un nombre de solutions égal au nombre de ressources libres. Pour un CGRA 4×4, cela fait potentiellement un facteur 16 et donc $1\,800 \times 16 = 28\,800$ *mappings*. La valeur du seuil calculé sera alors de $\exp\left(\frac{-28\,800}{5\,000}\right) \approx 0,0032$. Sur les 28 800 *mappings* partiels trouvés, il ne sera donc conservé qu'environ 90 *mappings*, ce qui s'avérera souvent insuffisant pour trouver un bon *mapping* voire un *mapping* tout court.

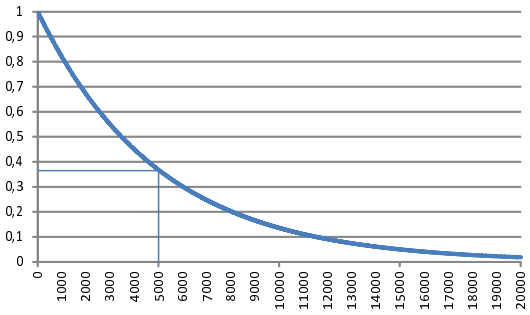
Version « avec *Offset* »
Pour remédier à ce problème, nous avons ajouté un « plancher » à la fonction de seuil. Elle répond alors à l'équation 3.3.2 avec $Offset \in]0..1[$. L'allure de cette fonction est donnée en figure 3.16b avec un $Offset$ de 10%.

$$Seuil = Offset + (1 - Offset)\exp\left(\frac{-NbMappings}{NbReference}\right) \tag{3.3.2}$$

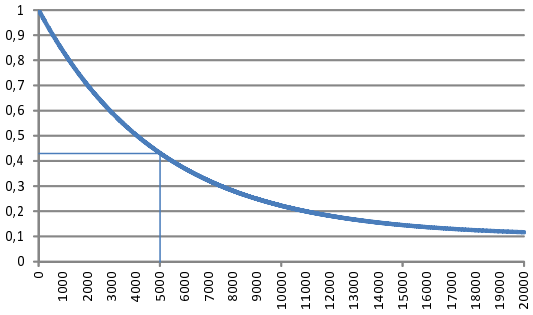
Ainsi, dans le cas de l'exemple précédent, il n'y a plus le problème qu'en partant de 28 800 *mappings*, il en reste moins d'une centaine après élagage. Avec un $Offset$ de 10%, il restera environ 2 960 *mappings*. Cette nouvelle fonction de seuil a cependant les mêmes travers à l'usage que la version à seuil constant.

3.3.1.3 Fonction inverse saturée

Comme nous le verrons dans la partie évaluation (3.3.2), nous n'avons pas réussi à obtenir de résultat satisfaisant avec les fonctions exponentielles décroissantes pour le calcul du seuil et ce malgré des essais sur une grande plage de valeur pour $NbReference$ et l'$Offset$. En effet,



(a) Seuil exponentiel décroissant simple.



(b) Seuil exponentiel décroissant avec un *offset* de 10%.

Figure 3.16 – Allures des fonctions de seuil exponentielles décroissantes avec $NbReference = 5\,000$.

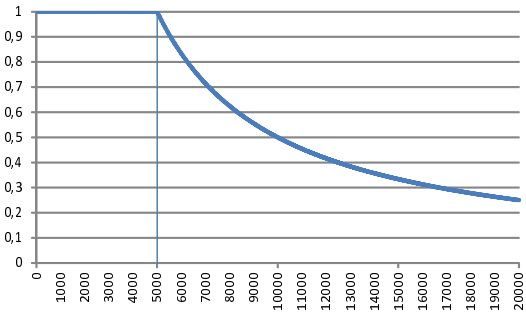


Figure 3.17 – Fonction de seuil inverse saturée avec $NbReference = 5\,000$.

quand un réglage convenait pour une application avec un CGRA de taille donnée, il suffisait de changer sa taille ou le DFG pour que ce réglage ne soit plus performant.

Nous avons donc choisi une autre fonction pour le calcul du seuil qui présente l'avantage d'être moins « agressive » en termes de décroissance. La fonction que nous avons retenue est une fonction inverse saturée. Elle répond à l'équation 3.3.3 et son allure est donnée en figure 3.17.

$$Seuil = \begin{cases} 1 & \text{si } NbMappings \leq NbReference \\ \frac{NbReference}{NbMappings} & \text{si } NbMappings > NbReference \end{cases} \quad (3.3.3)$$

Elle est moins « agressive » car le calcul de l'espérance du nombre de *mappings* restant après élagage donne $NbReference$. Autrement dit :

— si il y avait moins de $NbReference$ solutions partielles, elles seront toutes conservées ;

— si il y en avait plus, il restera, après élagage, environ $NbReference$ solutions.

À l'utilisation, cette méthode de calcul du seuil ne présente pas les comportements problématiques constatés avec les autres fonctions de calcul du seuil. La section 3.3.2 donne plus en détail les performances des deux méthodes de calcul de seuil évolutives (exponentielle et inverse) ainsi que la comparaison par rapport à l'état de l'art.

3.3.1.4 Autres fonctions et élagage non aléatoire

D'autres méthodes sont envisageables et devraient être évaluées pour améliorer l'étape d'élagage. Nous n'avons malheureusement pas eu le temps de les mettre en œuvre pendant le temps imparti à ces travaux de thèse. Dans cette section, les pistes de réflexions que nous avons eues sont présentées. Elles se classent en deux catégories : l'utilisation d'autres fonctions pour le calcul du seuil dans le cas d'un élagage stochastique ; et l'utilisation d'un élagage non aléatoire.

Autres fonctions

Une fonction candidate est une fonction qui varie entre 0 et 1 et qui est décroissante avec le nombre de *mappings*. Nous avons introduit un nombre de *mappings* de référence qui permet d'utiliser les fonctions mathématiques classiques (grâce à l'utilisation de variables sans dimension). Par exemple, les fonctions suivantes répondent aux critères :

— la tangente hyperbolique :

$$1 - \tanh\left(\frac{NbMappings}{NbReference}\right) \ ;$$

— des fractions plus complexes que l'inverse simple :

$$\frac{1}{1 + \left(\frac{NbMappings}{NbReference}\right)^n} \text{ avec } n \in \mathbb{N}^* \ ;$$

Autres paramètres

Jusqu'alors, nous avons simplement pris comme paramètre un nombre de *mappings* de référence. Cependant, nous avons remarqué lors de nos expériences qu'il y avait une grande variabilité de performance en fonction du code d'une part et de l'architecture d'autre part.

Il serait intéressant de tester l'influence de certains autres paramètres directement dépendant soit du code soit de l'architecture, par exemple :

— le nombre de nœuds de l'algorithme : intuitivement, plus il y a de nœuds à *mapper*, plus il faut laisser l'algorithme explorer pour qu'au final il trouve une solution ;

— le nombre de nœuds restants à *mapper* : comme pour le paramètre précédent, intuitivement, plus il reste de nœuds, plus il faudrait laisser l'algorithme explorer l'espace de solutions et *a contrario* moins il en reste, moins il est nécessaire de conserver un grand nombre de solutions. D'un autre côté, il ne faut pas trop diminuer le nombre de solutions à conserver quand l'algorithme approche de la fin au risque de ne plus avoir de solution du tout ;

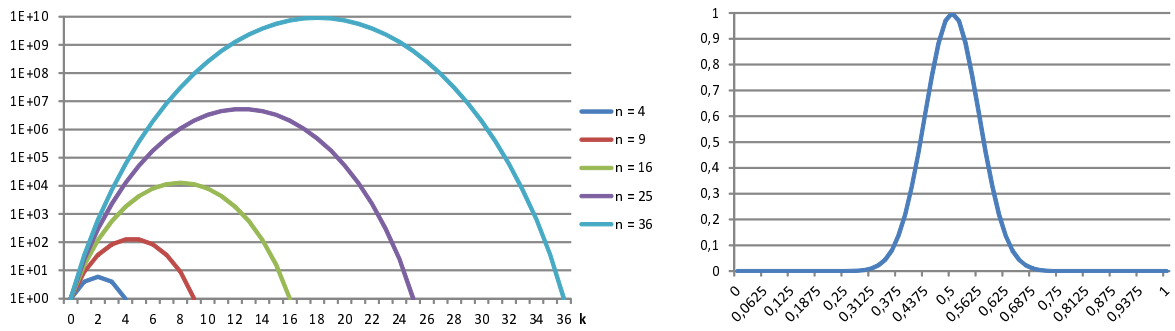
— le nombre de tuiles de l'architecture : plus l'architecture possède d'éléments et plus il y aura de solutions qui ne sont potentiellement que des permutations entre les unes et les autres. Pour avoir des solutions vraiment différentes, il faudrait augmenter le nombre de solutions à conserver, au risque encore une fois d'exploser en mémoire...

— la contrainte de nombre maximal de tuiles utilisables : moins on autorise un *mapping* à s'étendre, et plus ce sera compliqué pour la méthode de converger et il faudrait donc augmenter le nombre de solutions mémorisées.

Ces paramètres pourraient être combinés de manière à rendre la fonction de seuillage encore plus performante. Par exemple, une variable intéressante serait $r = \frac{contrainteNbTuiles}{nbTotalTuiles}$: le ratio entre la contrainte de nombre maximal de tuiles utilisables et le nombre de tuiles disponibles dans l'architecture. En effet, plus ce ratio est faible, plus le processus de projection est contraint et plus il faudrait conserver de solutions partielles. Si l'on ne tient pas compte de l'interconnexion, et si l'on considère que la contrainte du nombre de tuiles maximum est atteinte, alors il y a $\binom{n}{k}$ possibilités d'utiliser ces tuiles où n est le nombre de tuiles disponibles dans l'architecture et k le nombre utilisées, comme illustré dans la figure 3.18a. La figure 3.18b illustre l'influence du ratio de la contrainte de nombre de tuiles et du nombre total de ressources sur le nombre maximal de solutions atteignables (il s'agit d'une borne supérieure, une approximation plus réaliste tient compte de l'interconnexion).

Élagage non/moins aléatoire

En poursuivant plus loin la démarche précédente, on pourrait penser qu'un élagage basé sur une heuristique serait capable d'améliorer les performances sans introduire de caractère aléatoire ou alors de façon plus restreinte et contrôlée. En effet, il serait intéressant de pouvoir évaluer



(a) Valeur de $\binom{n}{k}$ pour différentes valeurs de n correspondant à des tailles classiques de CGRA.

(b) Évolution de $\binom{n}{k}$ par rapport à sa valeur maximal en fonction du ratio $\frac{k}{n}$.

Figure 3.18 – Allure de coefficients binomiaux

la pertinence d'une solution partielle par rapport aux autres ; le véritable problème étant de définir ce critère de pertinence. Notre but étant d'obtenir des solutions différentes au final, il faudrait pouvoir prédire quelles sont les solutions partielles qui convergeront vers une même utilisation globale des ressources. Mais il faudrait y arriver sans faire de comparaison entre tous les *mappings*.

Une première idée que nous n'avons pas eu le temps d'explorer ni d'évaluer complètement consiste à placer les solutions partielles dans un arbre en fonction des tuiles utilisées. La figure 3.19 illustre les différentes possibilités de classement des *mappings* utilisant entre une et quatre tuiles. Après ce classement, il faut effectuer un tirage aléatoire pour chacun des ensembles d'utilisation en utilisant par exemple la fonction inverse saturée. Le problème est alors de correctement régler le nombre de référence $NbReference$ pour chacun des sous-ensembles. En effet, ce nombre est censé représenter le nombre de solutions partielles que l'on souhaite conserver entre chaque étape. Il se pose alors un grand nombre de questions de mise en œuvre :

- si il y a des *mappings* partiels dans une grande partie de ces sous-ensembles, doit-on adapter $NbReference$ proportionnellement au nombre dans le sous-ensemble par rapport à la totalité ?
- doit-on favoriser les *mappings* les plus petits ? Ou au contraire les plus grands ?
- le $NbReference$ doit-il changer en fonction de l'avancement du *mapping* ?
- faut-il effectuer cette heuristique après le traitement de chaque nœud ou uniquement en fin de chaque cycle comme l'élagage exact ?

En effet, si cette heuristique est effectuée à la fin de chaque cycle, alors elle traite l'ensemble des nœuds de manière équitable ; *a contratio*, si elle est effectuée après chaque nœud, alors les nœuds n'auront pas tous subi le même nombre de tirages. Cela peut introduire un biais dans la sélection. Au final, nous n'avons pas trouvé d'élagage pertinent qui n'utilise pas d'aléatoire. Cela ne veut cependant pas dire qu'il n'en existe pas et cette recherche mérite d'être approfondie.

3.3.2 Évaluation

Dans cette section, nous évaluons les performances de cette seconde version du flot pour les DFGs qui inclut un élagage aléatoire. Nous commencerons par déterminer quelle fonction d'élagage donne les meilleurs résultats et aussi l'impact du paramètre $NbReference$ sur les performances. Nous nous intéresserons aussi à la pertinence d'associer les deux méthodes d'élagage ensemble (aléatoire, exécuté après chaque nœud et exhaustif, exécuté à la fin de chaque cycle). Enfin nous comparerons les performances globales pour la meilleure avec ceux de la version de la section précédente utilisant uniquement un élagage exhaustif.

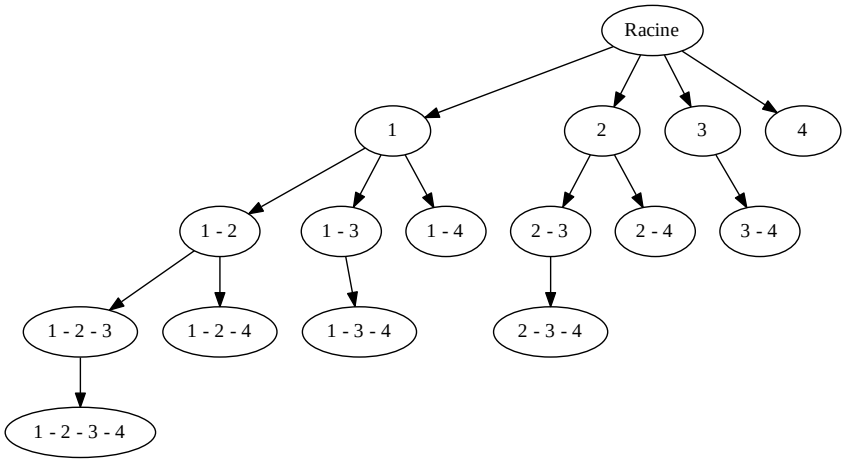


FIGURE 3.19 – Illustration de classement sur un arbre d'utilisation dans le cas d'une architecture possédant quatre tuiles numérotées de 1 à 4. Par exemple le nœud « 1 - 3 - 4 » correspond à l'utilisation des tuiles 1, 3 et 4 mais pas de la tuile 2.

3.3.2.1 Comparaison entre les fonctions

Nous avons implémenté et testé plusieurs variantes présentées dans la section précédente afin de déterminer quelle est la plus pertinente à utiliser. Nous avons tout d'abord comparé la version exponentielle décroissante avec et sans $Offset$ (la version avec un seuil fixe ne résolvant pas les problèmes de passage à l'échelle, nous ne présentons pas ces résultats ici). La version avec $Offset$ ne présente pas l'instabilité décrite précédemment, mais malgré cela elle obtient un taux de succès inférieur en fonction de la valeur de $NbReference$ qui varie entre -2% et -6% avec une moyenne à -5%. Cette écart est dû aux projections avec des contraintes moins fortes qui favorisent l'explosion du nombre de solutions partielles et donc peuvent provoquer leurs échecs.

Nous avons ensuite comparé les résultats obtenus avec l'élagage exponentiel et ceux avec l'élagage inverse saturé pour différentes valeurs de $NbReference$. Pour cela nous avons observé le temps de d'exécution, le taux de succès, le nombre de *mappings* différents générés ainsi que le débit de *mappings* différents (nombre de *mappings* différents générés par unité de temps) pour les mêmes applications et architecture que lors de l'évaluation de la méthode précédente. Les figures 3.20, 3.21, 3.22 et 3.23 présentent ces différentes observations. Sur la première (3.20), on observe que le temps de compilation est croissant avec $NbReference$ et qu'il est supérieur avec l'élagage inverse. Ce comportement était attendu car l'élagage inverse saturé conserve plus de *mappings* en moyenne car il en conserve systématiquement environ $NbReference$. On aura donc tendance à choisir une valeur plutôt petite de $NbReference$. La figure 3.21, qui présente le taux de succès des deux méthodes, montre qu'il n'y a pas d'impact pour l'élagage inverse du nombre de référence. En revanche, le taux de succès de l'élagage exponentiel croit avec ce nombre jusqu'à attendre la même limite que pour l'autre élagage. Pour ce critère, il est donc préférable pour l'élagage exponentiel d'avoir une valeur élevée de $NbReference$. La figure 3.22 donne l'influence de $NbReference$ sur le nombre de *mappings* différents : plus $NbReference$ est grand, plus il y a de *mappings* différents. Cette évolution n'est pas linéaire et tend à plafonner. L'élagage inverse saturé permet d'obtenir en moyenne plus de *mappings* différents que l'élagage exponentiel pour une valeur de $NbReference$ donnée. Pour ce critère, il faudrait donc choisir une valeur plutôt élevée de $NbReference$ pour maximiser le nombre de *mappings* différents et ce quel que soit l'élagage utilisé. La figure 3.23 montre qu'il existe des maxima pour le débit de *mappings* différents en fonction de l'élagage utilisé. Pour l'élagage exponentiel, l'optimum se situe vers 10 000 et pour l'élagage inverse saturé, il se situe entre 2 500 et 5 000.

Ces différentes observations permettent de conclure que l'élagage le plus pertinent pour obtenir rapidement un grand nombre de *mappings* différents est l'élagage inverse saturé. Il faut alors

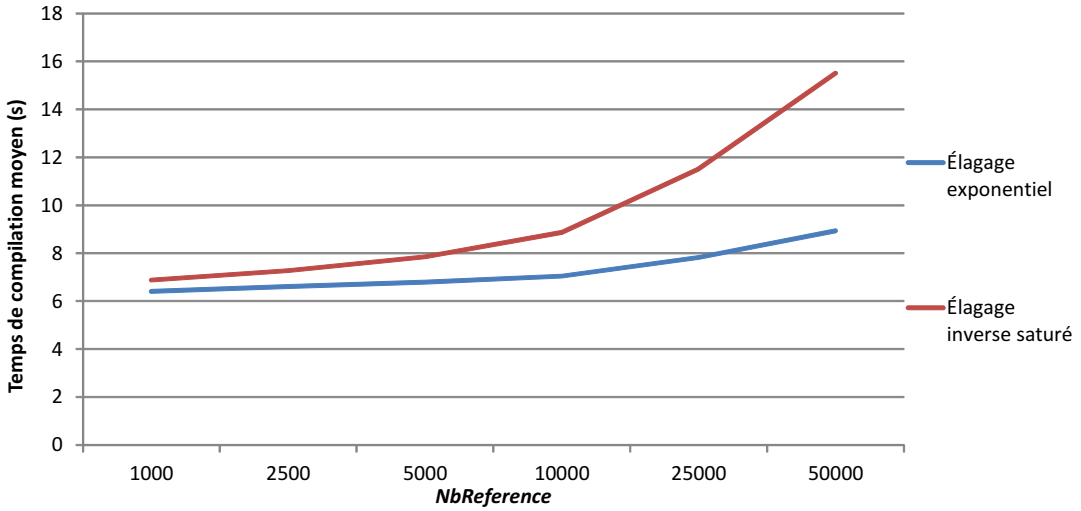


Figure 3.20 – Temps de compilation moyen pour les 2 élagages en fonction de $NbRefence$.

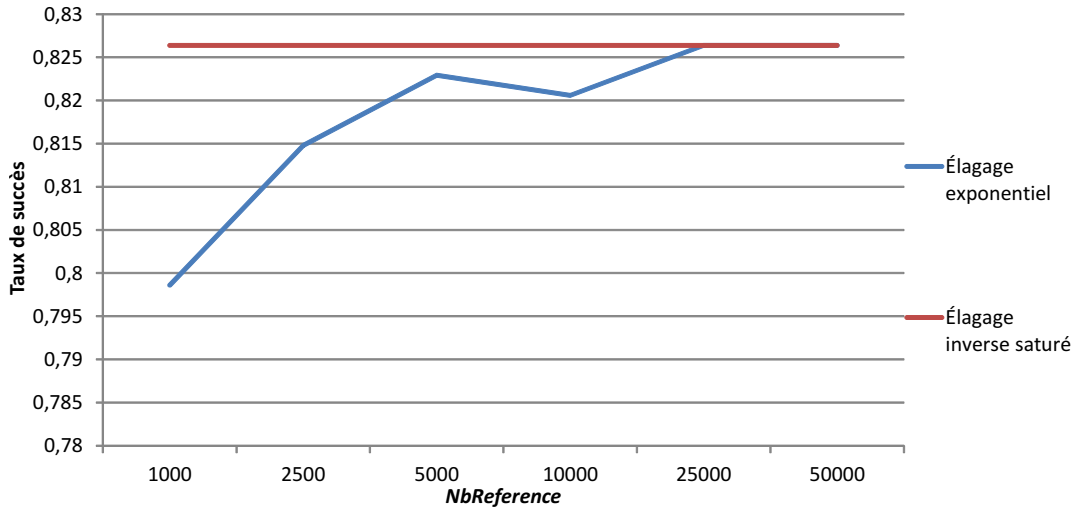


Figure 3.21 – Taux de succès pour les deux élagages en fonction de $NbRefence$.

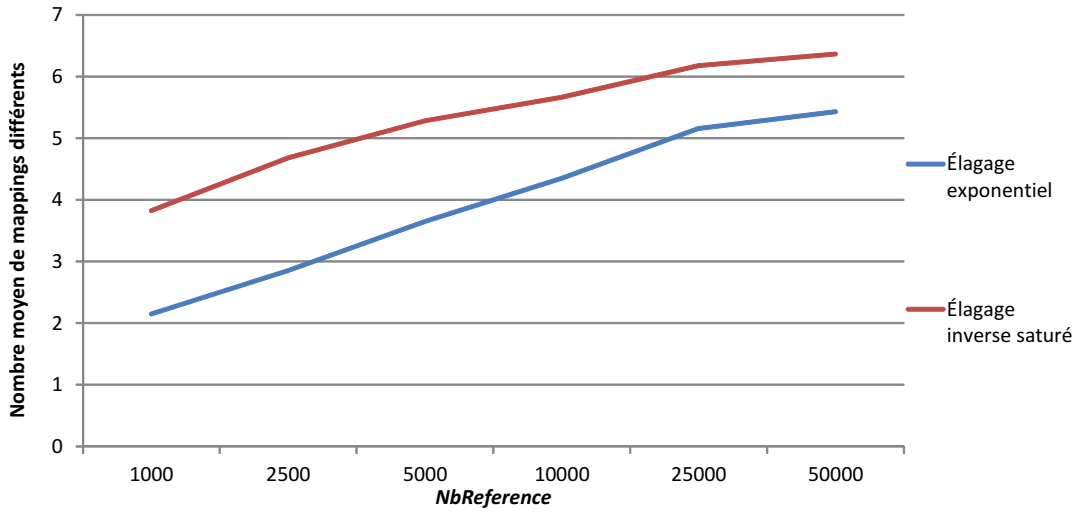


Figure 3.22 – Diversité des *mappings* pour les deux élagages en fonction de $NbRefence$.

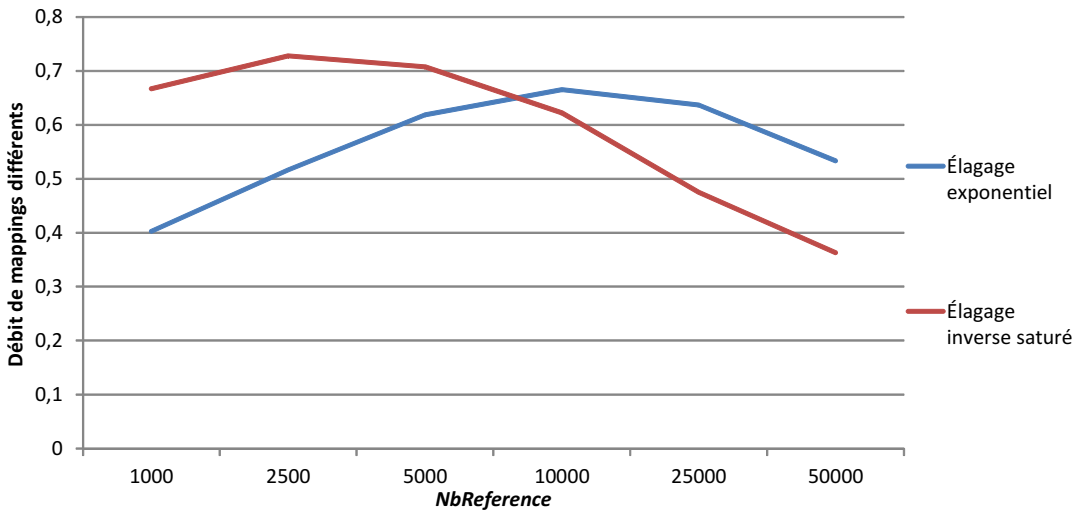


FIGURE 3.23 – Débit de *mappings* différents des deux élagages en fonction de $NbRefence$.

choisir un compromis entre temps de compilation et qualité du résultat. Comme notre premier critère est la qualité, la valeur retenue pour la suite pour $NbReference$ n'est pas l'optimum de la figure 3.23 : nous avons choisi 10 000. Ce paramètre fixé, la section suivante présente les performances de l'élagage inverse saturé.

3.3.2.2 Performances

L'analyse des performances de cette version du flot est faite en deux parties. La première consiste à vérifier qu'il est bien possible avec ces modifications de passer à l'échelle en termes de nœuds et de taille du CGRA. La seconde permet de vérifier que la qualité des résultats est toujours bonne. Quand l'élagage aléatoire est utilisé, c'est le tirage médian (et non pas moyen) qui est utilisé dans les résultats pour plus d'équité. En effet, si les résultats sont identiques 9 fois sur 10, le tirage médian donnera le cas identique alors que le tirage moyen sera soit inférieur, soit supérieur sans que cela ne reflète véritablement une tendance.

Passage à l'échelle

Pour évaluer la capacité à passer à l'échelle du flot avec un élagage inverse saturé, nous avons comparé les temps de compilation moyens pour l'ensemble des codes utilisés pour l'évaluation du flot semi-exhaustif sur une architecture possédant seize tuiles (un CGRA 4×4 torique). Nous avons fait varier la contrainte de nombre de tuiles maximales autorisées entre 1 et 16. La figure 3.24 donne les performances de cette méthode comparée à la version avec un élagage exact et par rapport à deux méthodes issues de l'état de l'art (respectivement Ordonnancement puis assignation et EPIMap) décrites précédemment (voir section 3.2.3). Contrairement aux trois autres méthodes, celle possédant un élagage inverse saturé est la seule à conserver un temps de compilation raisonnable (de l'ordre de la dizaine de secondes). Ce temps semble être relativement linéaire avec l'augmentation du nombre de ressources utilisables du CGRA, mais comme il s'agit d'une échelle logarithmique, l'évolution réelle reste exponentielle. Nous avons poursuivi ces expériences avec succès sur des architectures possédant jusqu'à 256 tuiles de calcul (CGRA 16×16).

Le deuxième élément permettant de vérifier la capacité à passer à l'échelle de notre flot inverse saturé est d'augmenter le nombre de nœuds de l'application. Le flot semi-exhaustif n'était pas vraiment capable de dépasser la trentaine de nœuds pour une architecture où au maximum 4 tuiles étaient utilisables.

Les tableaux 3.2 et 3.3 donnent quelques-uns des temps de compilation obtenus pour diffé-

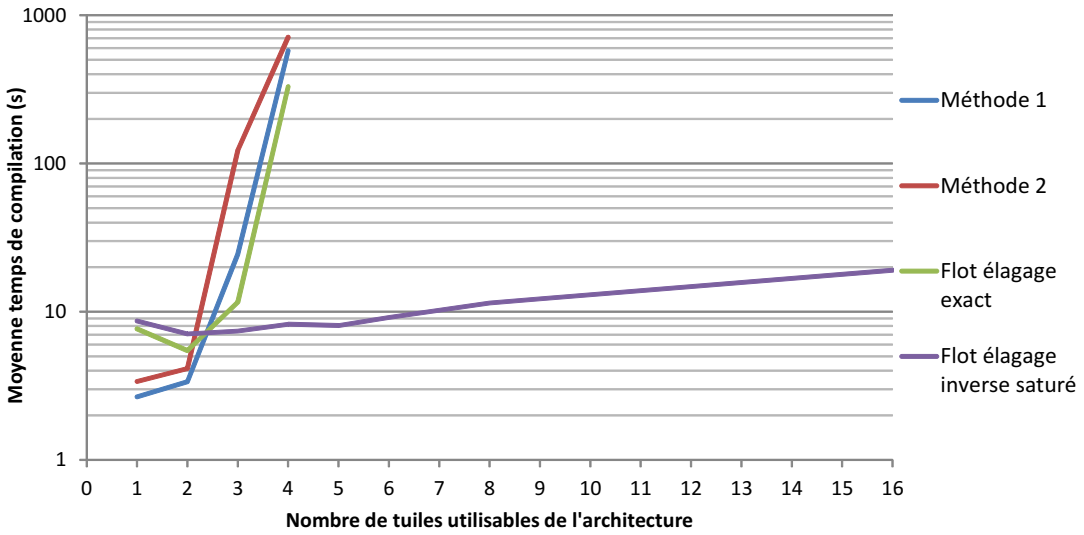


Figure 3.24 – Illustration du passage à l'échelle pour le temps de compilation de la méthode possédant un élagage inverse saturé sur un CGRA 4×4 torique. Le set d'applications considéré est le même que pour l'évaluation du flot semi-exhaustif.

rents codes en fonction de leur taille, de la taille du CGRA et de la contrainte de nombre de tuiles maximal. Les temps d'exécution obtenus croissent avec le nombre de tuiles autorisées et de tuiles disponibles, mais finissent par tendre vers une valeur maximale relativement raisonnable (ici, pour un DFG d'une cinquantaine de nœuds, on arrive à deux minutes). Cette valeur dépend du parallélisme maximal et des contraintes de dépendance de données qui limitent « l'étalement » du code sur l'architecture. On observe aussi que pour certains codes (*e.g.* 30 pour 5×5), le temps de compilation obtenu avec la contrainte 16 tuiles est supérieur à celui obtenu avec toutes les tuiles. Ce comportement étant reproductible, il n'est pas dû à l'aléatoire de la sélection. L'explication vient du fait que dans certaines conditions, l'algorithme passe plus de temps pour placer les nœuds à cause de cette contrainte qui impose des transformations de graphe et un grand nombre de routages, alors qu'en augmentant la contrainte, il peut placer tous les nœuds ordonnançables (ou une très grande partie) ce qui contraint très fortement le placement des nœuds suivants, limitant ainsi le nombre de variantes et donc le temps global d'exécution.

Impact sur la qualité

L'autre point à analyser est l'impact sur la diversité des *mappings* de l'utilisation de cet élagage aléatoire. En effet, il ne faut pas que le fait de passer à l'échelle détériore trop fortement la qualité des résultats. La première version du flot ne permettant pas d'obtenir des résultats sur un trop grand nombre de tuiles, nous limiterons donc l'analyse de la qualité aux cas où cela est possible de comparer. Nous reprenons donc le même set d'applications que dans la section 3.2.3.

Les comparaisons se feront entre le flot avec un élagage exhaustif (ou exact), le flot possédant un élagage inverse saturé mais aussi avec une version possédant les deux élagage à la fois. Pour cette dernière version, c'est l'élagage aléatoire qui est effectué en premier permettant à l'élagage exact de ne travailler que sur environ $NbReference$ *mappings* partiels. Effectuer les élagages dans l'autre ordre permettrait certainement d'augmenter le nombre de solutions différentes au final, mais en contrepartie risquerait de considérablement augmenter le temps de compilation. Supposons par exemple que $NbReference$ vaille 10 000. Sur une architecture 8×8, s'il y avait 10 000 solutions partielles avant le dernier nœud ordonnançable parmi une quinzaine de nœuds alors il est possible qu'il y ait $10\,000 \times 50 = 500\,000$ solutions partielles. L'élagage exact étant d'une complexité en $O(n^2)$ (ou dans le meilleur des cas en $O(n \log(n))$), le temps d'exécution explosera. Les figures 3.25, 3.26 et 3.27 donnent les tendances pour les différents codes pour

Tableau 3.2 – Exemples de temps de compilation (en secondes) pour un code de 8 nœuds sur différentes tailles de CGRA et avec diverses contraintes de nombre maximal de tuiles (8 registres dans la RF).

Taille CGRA	Contrainte nombre de tuiles maximal										
	1	2	3	4	5	6	7	8	10	12	14
3×3	5,0	3,2	3,2	3,3	3,3	3,3	3,4	3,4			
4×4	4,9	3,2	3,2	3,3	3,5	3,5	3,6	3,6	3,6	3,6	3,6
8×8	4,8	3,2	3,2	3,7	4,6	5,4	5,7	5,7	5,8	5,8	5,8
10×10	4,8	3,2	3,3	4,1	6,1	7,6	8,1	8,2	8,1	8,2	8,2
16×16	4,8	3,3	3,7	7,1	14,6	20,9	22,8	23,0	22,6	22,5	23,0

...

Taille CGRA	Contrainte nombre de tuiles maximal										
	15	18	20	25	32	55	60	80	90	120	Toutes
3×3											3,3
4×4	3,6										3,6
8×8	5,8	5,8	5,8	5,8	5,8	5,8	5,8				5,8
10×10	8,2	8,3	8,1	8,2	8,2	8,3	8,3	8,1	8,2		8,2
16×16	22,4	23,1	22,5	22,5	22,5	22,9	22,6	22,6	22,9	22,5	22,9

Tableau 3.3 – Exemples de temps de compilation (en secondes) pour des DFGs possédant plus de nœuds (issus d'un code de calcul de Blowfish).

Nombre de nœuds	Taille CGRA	Contrainte nombre de tuiles maximal								
		2	3	4	5	6	8	12	16	Toutes
10	3×3	3,7	3,8	4,2	4,5	4,8	4,9			4,9
	4×4	3,7	3,8	4,3	4,8	5,2	5,6	5,6	5,6	5,6
	5×5	3,7	3,8	4,3	4,9	5,4	5,9	6,0	6,0	6,0
12	3×3	3,9	4,3	4,1	4,5	5,0	5,3			5,4
	4×4	4,2	5,0	5,2	6,3	8,6	9,8	10,1	10,2	10,2
	5×5	4,2	5,1	5,6	6,3	8,1	10,2	15,4	15,1	15,3
14	3×3	4,2	4,0	4,8	5,4	5,9	6,2			6,2
	4×4	4,2	3,8	4,9	5,6	6,3	7,2	7,6	7,6	7,6
	5×5	4,2	3,8	4,8	5,7	6,7	8,1	8,8	9,0	8,9
18	3×3	5,3	5,5	6,6	7,5	8,2	8,9			9,0
	4×4	5,3	5,1	6,8	7,8	8,9	10,6	11,5	11,5	11,4
	5×5	5,3	5,1	6,7	7,9	9,3	12,0	16,1	15,9	15,9
21	3×3	6,4	6,6	7,8	8,1	9,0	10,2			10,4
	4×4	6,4	6,3	8,2	9,3	11,1	14,1	14,6	14,7	14,6
	5×5	6,4	6,2	8,2	9,3	12,7	18,9	30,4	29,0	30,7
30	3×3	10,3	9,4	9,4	10,9	12,2	14,1			14,7
	4×4	10,2	8,0	9,5	11,1	12,7	16,0	19,3	19,2	19,1
	5×5	10,2	7,9	9,5	11,3	13,3	17,8	29,2	34,0	31,8
38	3×3	13,6	15,4	12,4	12,7	13,4	15,7			16,8
	4×4	13,6	13,4	12,5	12,6	13,5	17,2	22,5	22,4	22,5
	5×5	13,6	13,2	12,3	12,5	14,1	19,1	30,5	40,5	38,0
53	3×3	22,7	25,8	26,2	30,1	35,6	46,0			50,7
	4×4	22,7	22,4	26,3	30,0	35,9	48,3	68,1	80,8	80,7
	5×5	22,7	22,4	26,5	31,1	38,6	54,2	81,0	104,4	128,3

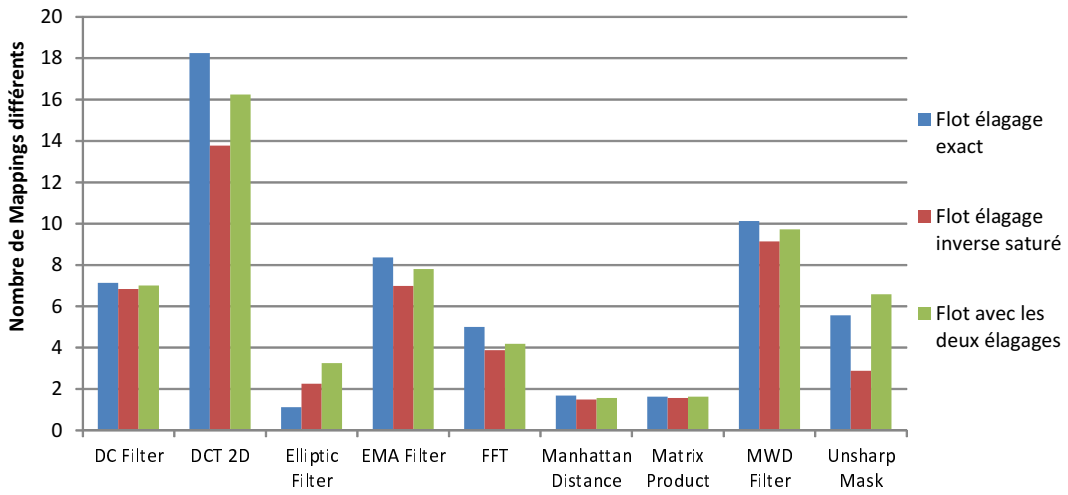


Figure 3.25 – Diversité des *mappings* pour les différents élagages.

respectivement le nombre de *mappings* différents obtenus (la diversité), le nombre de *mappings* différents générés par secondes (le débit) et le taux d'obtention de la meilleure latence parmi les trois versions. La figure 3.25 montre qu'en général le nombre de *mappings* différents obtenus par la version avec un élagage aléatoire est plus faible que celui avec un élagage exact – ce qui est attendu – mais de seulement quelques pour-cents en moyenne. La version possédant les deux élagages obtient quant à elle plus de *mappings* que la version avec seulement un élagage aléatoire tout en étant généralement inférieure à la version exacte, ce qui est là encore assez logique.

On peut cependant noter dans cet exemple deux points particulièrement intéressants :

— le premier concerne les résultats pour le filtre elliptique. Pour ce code, les méthodes possédant une partie aléatoire sont meilleures que celles possédant un élagage exact. Cela s'explique par le fait que lors des projections, certaines solutions ont été supprimées or ce sont ces solutions qui permettaient de placer un des nœuds à un cycle donné. Ce nœud a donc été transformé ce qui entraîne une plus grande diversité. La version possédant les deux élagages conserve plus souvent les solutions provenant de cette branche, favorisant la diversité ;

— le second concerne le masque flou (*Unsharp Mask*). Là encore, c'est l'aléatoire qui joue un rôle prédominant. Dans ce cas précis, l'aléatoire permet, par transformation d'un nœud d'obtenir une nouvelle branche de *mappings*. Mais cette branche n'est pas forcément conservée par la suite si l'élagage utilisé est l'inverse saturé alors qu'elle l'est beaucoup plus souvent dans le cas d'utilisation du double élagage. C'est pour cette raison que le flot avec les deux élagages est meilleur que celui avec l'élagage exact.

La deuxième figure (3.26) ajoute à la première l'aspect temporel. Le flot n'effectuant pas de comparaison est généralement celui qui produit le plus de *mappings* différents par unité de temps, c'est donc le plus efficace. Il est d'ailleurs nettement meilleur quand le nombre de *mappings* différents augmente (*e.g.* DCT 2D et MWD Filter). Les résultats pour le filtre DC peuvent sembler assez curieux, mais il s'agit en fait du cas particulier où le nombre de *mappings* et de *mappings* différents sont assez faibles et pour lequel il est plus rentable de passer un peu de temps à supprimer les doublons qu'à en conserver au moins $NbReference\ mappings$.

La troisième figure (3.27) donne le taux d'obtention de la meilleure latence parmi les trois méthodes, c'est-à-dire le pourcentage de fois que la méthode obtient la latence la plus faible entre les trois méthodes. Il y a plusieurs cas de figure :

— pas d'impact de l'élagage : c'est le cas pour le filtre DC, le filtre à moyenne exponentielle glissante ou EMA et le masque flou (*Unsharp Mask*) ;

— un impact négatif de l'élagage aléatoire : c'est la cas pour la DCT le le filtre MWD.

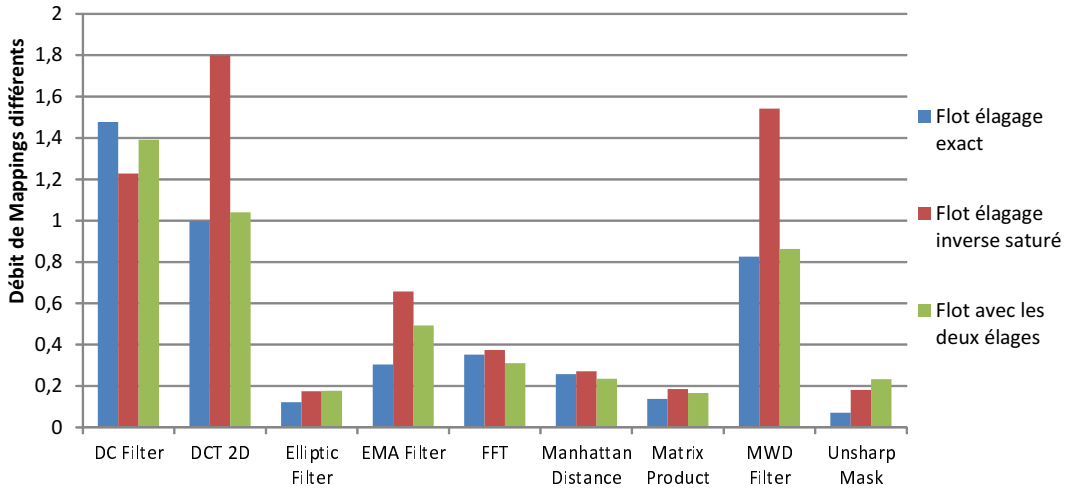


FIGURE 3.26 – Débit de *mappings* différents pour les différents élagages.

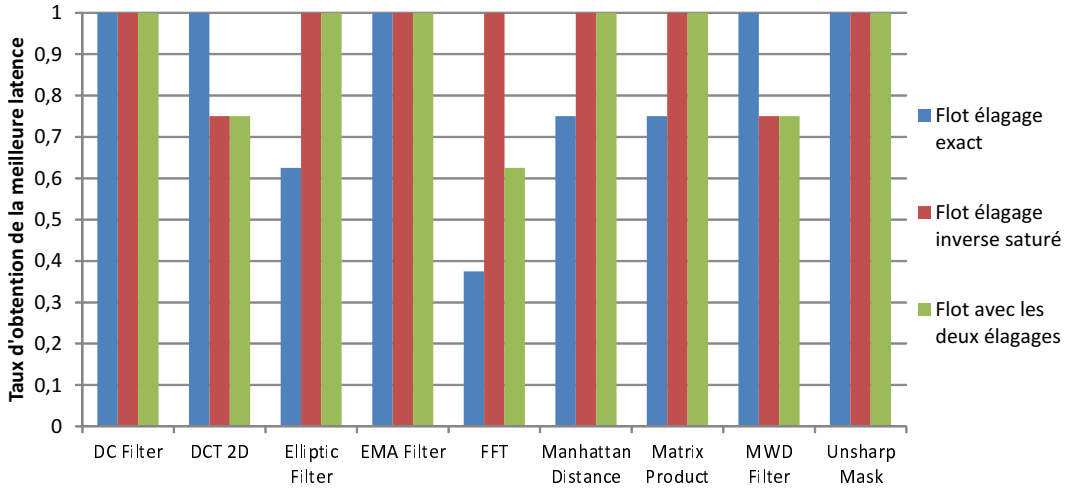


FIGURE 3.27 – Taux d'obtention de la meilleure latence entre les différents élagages.

Dans les deux cas, la différence provient du fait que pour la contrainte d'une seule tuile utilisable, les méthodes utilisant des élagages aléatoires n'ont pas réussi à trouver une solution (échecs de la projection). Ce résultat négatif peut cependant être relativisé par le fait que pour une contrainte aussi forte, l'utilisation de l'aléatoire ne se justifie pas.

— un impact positif de l'élagage aléatoire : c'est le cas pour le filtre elliptique, le calcul de la distance de Manhattan et le produit de matrice. Pour ces codes, l'aléatoire a entraîné une transformation de graphe qui au final a permis de diminuer la latence. De le cas de la FFT, ne pas utiliser l'élagage exact en plus de l'élagage aléatoire a permis d'obtenir encore plus souvent une meilleure latence. Après investigation, la raison est qu'il y a une seule branche de l'arbre des *mappings* permettant d'obtenir cette latence parmi toutes celles trouvées. Cette branche apparaît après une transformation due à l'élagage aléatoire. Elle possède un grand nombre de variantes équivalentes. Le fait d'exécuter l'élagage exact va supprimer les différentes variantes ce qui augmente la probabilité que cette branche disparaisse lors des élagages aléatoires suivants.

Ces différentes analyses peuvent laisser penser que le flot avec les deux élagages est la meilleure. Cependant, comme le montre la figure 3.28, cette méthode est certes bien meilleure en temps de compilation que la version exacte, mais son évolution par rapport au nombre de

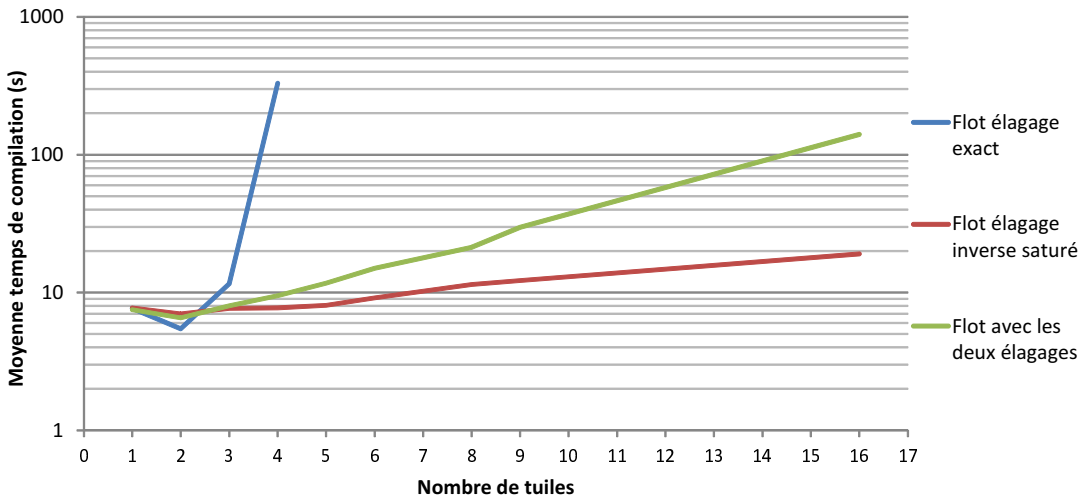


Figure 3.28 – Temps moyen de compilation pour les différents élagages.

tuiles est bien plus importante que celle du flot avec simplement un élagage aléatoire (dans une échelle logarithmique). Il y a donc un compromis à faire entre les deux en fonction de la taille du CGRA : si il est « plutôt petit », alors la version possédant les deux élagages permettra d'obtenir plus de *mappings* différents ; si au contraire il est plus important, il faudra dégrader légèrement la qualité du résultat pour avoir un temps raisonnable.

3.4 Conclusion

Dans ce chapitre, nous avons détaillé deux versions d'un flot permettant de projeter une application représentée par un DFG sur un CGRA en obtenant directement plusieurs solutions de même latence utilisant de manière différente l'architecture. Ce flot constitue la première étape permettant d'effectuer de la tolérance aux fautes par reconfiguration dynamique sur CGRA. Les résultats concernant la méthode semi-exhaustive ont été présentés par un poster à la conférence internationale GLSVLSI'14 [Peyret et al., 2014a], publiés dans la conférence francophone ComPAS'14 [Peyret et al., 2014c] et dans la conférence internationale ASAP'14 [Peyret et al., 2014b]. Les résultats concernant la méthode stochastique sont en cours de publication.

Ces travaux forment une preuve de concept et possèdent de nombreuses pistes d'améliorations. Parmi ces perspectives, les plus importantes sont les suivantes :

- tester l'ajout d'aléatoire dans l'ordonnancement d'un nœud pour augmenter la diversité ;
- ajuster dynamiquement la politique d'élagage en fonction des contraintes et éventuellement de paramètres extérieurs à la méthode comme la capacité mémoire et de calcul de l'ordinateur qui réalise le processus de projection ;
- poursuivre les développements de l'élagage aléatoire dans un arbre ;
- observer l'impact des choix de la fonction de priorité des nœuds ;
- évaluer l'intérêt d'une transformation de graphe qui, lorsque l'algorithme de projection ne trouve pas de solution par manque de registre, ajouterait un ou plusieurs couples de nœuds *load/store* pour mettre en mémoire une ou plusieurs variables stockées en registre et ainsi libérer de la place pour les opérations courantes ;
- ré-écrire le code de la méthode dans un langage autre que Java (*e.g.* C ou C++) pour augmenter sensiblement les performances, maintenant que la preuve de concept est faite.

Chapitre 4

Multi-projection de CDFG sur CGRA

4.1 Introduction

Le chapitre 3 a défini une méthode permettant d'obtenir un grand nombre de *mappings* tous de même latence pour une application représentée par un DFG sur un CGRA. Dans ce chapitre, un moyen d'étendre ce flot pour des applications possédant un flot de contrôle est présenté. Puis la méthode générale permettant d'obtenir, à partir des ce flot, dans un contexte de tolérance aux fautes, une variété de *mappings* encore plus importante est décrite. Enfin, les mécanismes de gestion des *mappings* lors du fonctionnement en ligne sont détaillés.

4.2 Flot pour les CDFGs

Cette section présente les modifications qu'il faut apporter au flot précédent pour qu'il soit capable de gérer des applications possédant un flot de contrôle. Puis le flot retenu est évalué.

4.2.1 Problématiques spécifiques

Comme présenté dans le chapitre 2, un CDFG diffère d'un DFG par trois aspects principaux :
- le fait d'avoir plusieurs BBs et de passer des uns aux autres grâce à des sauts ;
- le fait d'avoir des variables partagées entre les BBs ;
- et le fait d'avoir une instruction particulière PHI permettant de choisir une valeur pour une variable affectée conditionnellement en fonction du BB précédemment exécuté.

Ces aspects doivent être pris en compte par la méthode de projection sous peine de générer des *mappings* non fonctionnels. Dans cette section, pour chacun de ces aspects, un ensemble de solution est présenté. L'exemple de CDFG qui avait été présenté dans le 2 est repris dans ce chapitre pour plus de clarté. Les choix retenus parmi ces différentes solutions sont discutés en section 4.2.2.

4.2.1.1 Gestion des sauts

Nous avons fait le choix d'expliciter dans le graphe les opérations de sauts. Ainsi, une opération est ajoutée aux autres opérations du BB. Si ce saut est un saut inconditionnel, il n'a pas de dépendance spécifique par rapport aux autres nœuds. Par contre, si ce saut est un saut conditionnel, il a besoin de connaître le résultat d'un test. Il s'agit le plus souvent d'un test d'égalité ou d'inégalité. Par exemple, pour savoir si il faut sortir d'une boucle *for*, il faut tester la valeur de l'indice de boucle par rapport à la valeur de sortie. Cette opération de test, par le processus de projection, sera exécuté à un instant donné. Cet instant ne sera pas forcément le dernier car cette instruction n'est généralement pas sur le chemin critique. Le fait d'ajouter artificiellement une instruction de saut avec un lien la reliant avec le test permet de s'assurer que le lien de dépendance de donnée sera respecté. Il autorise une plus grande flexibilité de placement

et peut permettre de diminuer la latence globale d'un cycle d'exécution. En effet, dans le cas où il n'y a pas d'instruction de saut explicite dans le graphe du BB, la variable contenant le résultat du test sera située dans un registre particulier d'une tuile (registre de sortie ou dans la RF) après un certain nombre d'opérations de mémorisation permettant sa conservation jusqu'au dernier cycle. Pour générer le programme/code assembleur qui sera effectivement présent dans le CGRA, il faudra ajouter au contrôleur de la tuile exécutant l'opération de test (ou la dernière mémorisation du résultat) l'instruction de saut conditionnel. Cette instruction s'exécutera un cycle après toutes les autres, augmentant la latence du BB de un cycle.

L'ajout explicite de l'instruction/opération de saut dans le BB et de son lien de dépendance avec l'instruction de test, permet, par le processus de projection, de faire qu'elle s'exécute en parallèle des dernières instructions des autres tuiles et éventuellement dans une tuile voisine de celle qui a effectué le test (atteignable par le réseau d'interconnexion).

4.2.1.2 Gestion des variables partagées

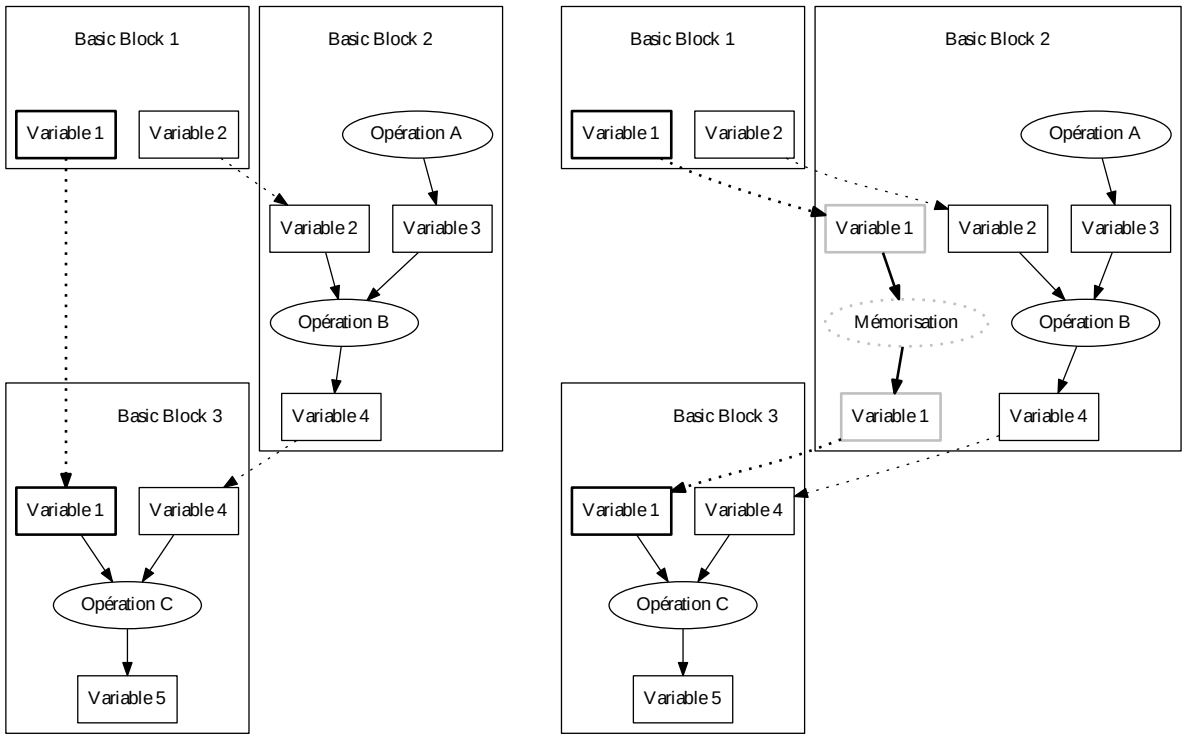
Le flot présenté dans le chapitre précédent ne prend en compte qu'un seul DFG. Dans le contexte d'un CDFG, cela revient à ne prendre en compte qu'un seul BB et ne permet pas de connaître la localité d'une variable en registre lorsque le flot passe de la projection d'un BB à un autre. C'est pourquoi, les variables partagées entre BBs nécessitent un traitement particulier dans le flot de projection.

Une première idée pourrait être de simplement récupérer les assignations de la variable (c'est-à-dire les différents registres où elle peut être en fonction des *mappings* obtenus) et de prendre ces positions comme base pour continuer la projection de l'application. Cependant, cette solution ne fonctionne pas dans le cas où une variable est produite dans un BB et n'est pas utilisée directement dans le BB suivant. À titre d'exemple, dans la figure 4.1a les BB 1, 2 et 3 s'enchaînent. La variable 1 est initialisée dans le BB 1 ; elle n'est pas utilisée dans le BB 2 mais seulement dans le BB 3. Il serait possible de forcer la localisation de la variable 1 dans le BB 3 à être la même que celle dans le BB 1 (par exemple le registre 1). Cependant, si dans le BB 2, une autre variable (variable 3 dans la figure) prend la place *a priori* libre du registre 1, alors quand l'exécution arrive au BB 3, ce ne sera plus la valeur de la variable 1 qui sera dans le registre 1 mais celle de la variable 3.

Plusieurs solutions existent pour palier ce problème. Les paragraphes suivants présentent trois politiques différentes pour permettre au flot de prendre en compte ces variables partagées sans le remettre en cause en intégralité et en expliquant leurs avantages et inconvénients.

Politique 1 : Réservation de registres

Pour empêcher d'écraser des données en registre, la première idée consiste à réserver ces registres de manière globale et donc de bloquer pour l'intégralité des BBs traversés les registres contenant des variables partagées. Ainsi, pour chaque variante de *mappings* d'un BB produisant des variables utilisées dans d'autre BBs, la méthode de projection connaîtra les positions des registres à ne pas écraser. Cette solution est assez simple à mettre en œuvre, mais pose le problème de complètement figer la position des registres contenant des variables partagées. Il s'agit d'un véritable problème car, si une tuile, lors de la projection du $i^{ème}$ BB, n'a plus de registre libre car chacun d'eux a été bloqué, ce sera comme si elle ne permettait pas d'effectuer de calcul. Ceci entraînera soit une augmentation de la latence résultante, soit l'échec de la projection par manque de ressource disponible. Cette première politique peut être grandement améliorée en considérant les durées de vie des variables, ce qui permettrait, dans chaque BB, de ne réserver que le nombre minimal de registres. Cependant, l'allocation reste fixe et peut rendre inutilisable certaines tuiles par manque de registres disponibles dans la RF. Cette solution n'est donc envisageable que si l'architecture possède un grand nombre de registres.



(a) Le CDFG partiel initial.

(b) Le résultat de la transformation.

Figure 4.1 – Exemple d'explicitation de dépendence pour la gestion des variables partagées par ajout de nœuds de mémorisation.

Politique 2 : Explicitation des liens de dépendances

Une deuxième méthode pour gérer ces variables partagées est d'expliciter la conservation de la donnée en registre dans tous les BBs traversés. Elle consiste à ajouter des variables fictives dans les différents BBs où la donnée a besoin d'être conservée et d'expliciter la conservation pour tous les cycles d'ordonnancement grâce à des opérations de mémorisation. Une illustration du résultat de cette méthode pour le CDFG partiel de la figure 4.1a est donnée en figure 4.1b.

Contrairement à la politique précédente, elle ne fige pas la position d'une variable dans un registre car la méthode de projection est capable de faire se déplacer des variables de la RF d'une tuile vers d'autres tuiles. Cependant, pour que le CDFG s'exécute correctement, il faut s'assurer que les registres utilisés d'un BB à un autre contiennent les bonnes valeurs. Dans l'exemple de la figure 4.1b, il il faut que les registres contenant les variables 1 et 2 du BB 1 soient les même que ceux du BB 2 et de même pour le passage du BB 2 au 3.

Cette politique présente l'avantage de ne pas figer le placement des variables. Dans le cas où, pour un BB donné, la projection échouerait, il pourrait être envisagé de reprendre les solutions du BB précédemment projeté et de forcer les variables partagées à changer de place, puis de relancer la projection pour le BB suivant. Ce serait une manière supplémentaire d'explorer l'espace des solutions, mais augmentant la complexité du processus de projection.

Politique 3 : Store/Load systématique

La troisième politique, qui peut sembler un peu naïve *a priori*, est celle dite du « *Store/Load* systématique ». Elle consiste à casser les dépendances explicites de données en insérant des *stores* et des *loads* là où il y des variables partagées. Plus précisément, un *store* est ajouté après une variable à chaque fois qu'elle est utilisée dans un ou plusieurs autres BBs et des *loads* sont ajoutés dans chaque BB où elle est utilisée.

Cette solution présente l'avantage de rendre les BBs indépendants et de ne pas « bloquer »

de registre avec des données qui ne seront pas utilisées dans le BB courant. Cependant, elle augmente le nombre d'accès à la mémoire centrale, ce qui peut s'avérer pénalisant si la bande passante est limitée (ce qui est souvent le cas dans la pratique). Cet aspect pénalisant peut néanmoins être réduit s'il existe un *scratchpad* commun à toutes les tuiles et que les adresses des *stores/loads* pointent vers ce *scratchpad*. De cette manière, la bande passante avec la mémoire centrale n'est pas affectée.

Concrètement, dans le flot présenté au chapitre précédent, une passe intermédiaire doit être ajoutée entre la récupération du CDFG depuis le *frontend* GCC et la partie effectuant les projections. Cette passe identifie les variables partagées et ajoute les *stores/loads* nécessaires en gérant les adresses de manière à ne pas entrer en collision avec d'autres accès mémoire.

4.2.1.3 Gestion des opérations *PHI*

Les nœuds *PHI* sont des opérations particulières introduites dans le CDFG de manière à pouvoir choisir une valeur en fonction du BB qui a été précédemment exécuté. On peut distinguer deux cas de figure.

Le premier cas est celui d'une variable qui peut prendre deux valeurs différentes en fonction du BB précédent : soit deux valeurs provenant de deux BBs différents soit d'un choix entre une valeur calculée précédemment et une valeur par défaut (comme dans la figure 4.2(c)). Le nœud ressemblera alors à une opération classique sauf qu'elle doit en plus connaître le BB précédent pour décider de son résultat. Ce cas ne pose pas de problème particulier car ces variables d'entrées sont des variables partagées et rentrent dans le cadre de la section précédente (4.2.1.2).

Le second cas de figure est le cas d'une variable locale au BB, c'est-à-dire qui n'est utilisée que dans ce BB, et qui est incrémentée au fur et à mesure des exécutions du BB. C'est typiquement le cas de la variable d'indice de boucle *for*. Seul ce second cas de figure est problématique. La figure 4.2(b) possède deux exemples de ce genre de nœud. On observe que la sémantique d'un DFG n'est pas respecté ici car il y a un cycle de dépendance de données. Ce cycle est par exemple fermé entre le nœud de variable dn_46 et l'opération an_36. Cette opération est un *PHI*, c'est-à-dire qu'au moment de s'exécuter, elle va regarder quelle est la bonne variable à choisir entre dn_46 et 0 pour les opérations suivantes qui en dépendent. Ce choix se fait en fonction de ce qui s'est passé en amont de ce BB. Dans le cas de la figure 4.2(b), le nœud *PHI* prendra la valeur de dn_46 si le BB précédent était le même (en cas de rebouclage) et la valeur 0 si ce n'est pas le cas (en provenance du bb_35).

L'apparition de cycles fermés dans un BB impose de les traiter de façon particulière et de modifier la méthode de projection en conséquence. En effet, si on applique le flot du chapitre précédent tel quel pour le BB bb_93, seule l'instruction de saut et l'opération an_40 qui détermine quel sera le futur BB à être exécuté seront ordonnancées et mappées. La méthode de projection ne trouvera pas de solution pour l'intégralité des nœuds car ils ne respectent pas le critère d'ordonnançabilité arrière qui est d'avoir toutes ces opérations filles d'ordonnancées (cf. chapitre 3 section 3.2.2.1). Il existe plusieurs solutions pour gérer ces boucles fermées dans un BB. Nous présentons deux politiques différentes dans les paragraphes suivants.

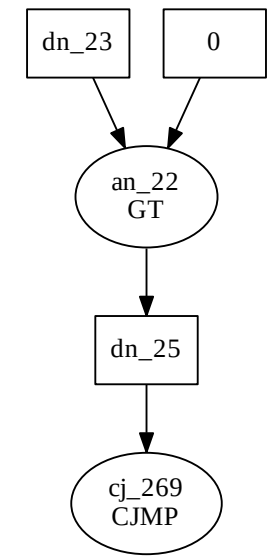
Politique 1 : Ouverture de boucle fermée par variable pseudo-partagée

Cette première politique ressemble à celle de gestion des variables partagées. Elle consiste à dire que la variable de sortie de l'opération d'incrémentation et la variable d'entrée de l'opération *PHI* (la variable représentée par le nœud dn_46 dans la figure 4.2(b)) n'est pas un seul nœud, mais deux nœuds d'une même variable partagée. Comme le partage se fait au sein du même BB, il ne s'agit pas à proprement parler de variable partagée, mais de variable pseudo-partagée ; bien que, durant l'exécution, le partage se fera entre deux exécutions successives (et donc distinctes) du BB.

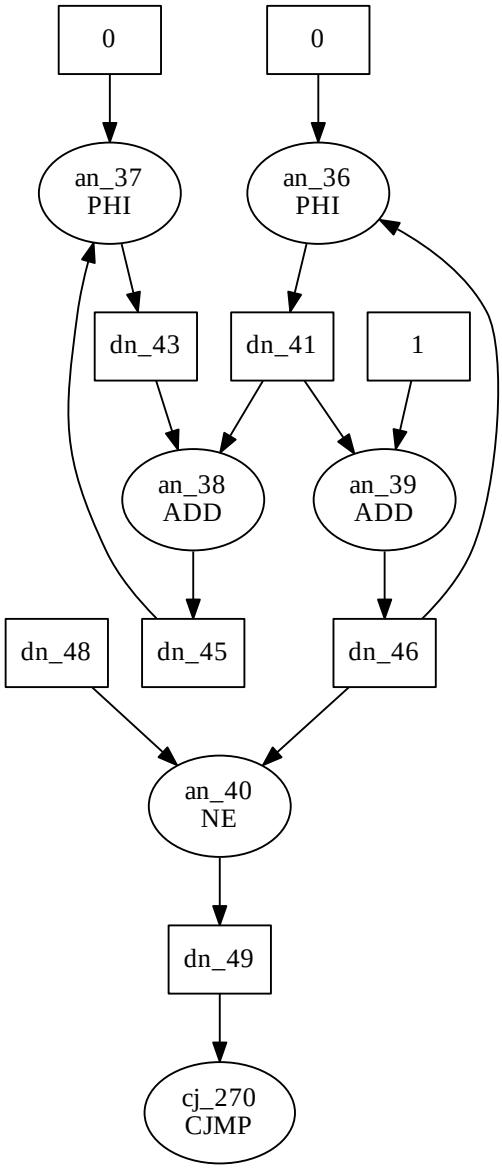
L'idée est alors d'appliquer à cette variable pseudo-partagée la politique de gestion des variables partagées. La figure 4.3 illustre le résultat de l'application de la troisième politique de

Algorithme 4 Exemple de code avec une boucle *for* simple

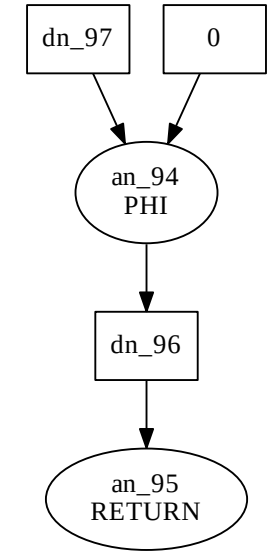
```
1  int boucleForSimple (int n)
2  {
3    int i;
4    int res = 0;              // Variable de resultat
5    for (i = 0; i < n; ++i)
6    {
7      res += i;               // Coeur de boucle
8    }
9    return(res);
10 }
```



(a) Basic Block bb_35.



(b) Basic Block bb_93.



(c) Basic Block bb_116.

Figure 4.2 – CDFG de l'algorithme 4.

gestion des variables partagées (*store/load* systématiques). Cette solution présente l'avantage d'ouvrir le cycle fermé dans le graphe du BB. Ainsi, le critère d'ordonnançabilité n'a pas à être modifié et le nœud store sera ordonnançable dès le départ ; permettant ainsi à l'ensemble du BB d'être ordonnancé et assigné.

Politique 2 : Restriction de placement de l'opération *PHI*

La seconde politique consiste à modifier le critère d'ordonnançabilité pour ignorer les nœuds *PHI* et ensuite à restreindre leur placement de sorte que le placement du nœud de donnée (dn_46 dans la figure 4.2(b)) contraigne le placement du nœud *PHI*. Ce mécanisme n'est pas celui de base de notre méthode de projection car, effectuant un ordonnancement arrière, les seules dépendances à considérer sont celles dues aux nœuds enfants. Mais ne considérer que le placement de nœud de donnée n'est pas suffisant. Il faut s'assurer que le registre qui contient cette valeur n'est pas utilisé entre la création de la variable et la fin du BB d'une part, et entre le début du BB et le nœud *PHI* d'autre part.

Pour ne pas écraser la donnée entre sa création et la fin du BB, nous avons déterminé deux approches :

- la première consiste, lors de la détermination des opérateurs compatibles pour l'opération créant la donnée à conserver, à ne choisir que les opérateurs qui ont au moins un registre atteignable qui n'a pas été utilisé dans les cycles déjà ordonnancés. Cette méthode présente l'avantage d'être très simple à mettre en œuvre mais a pour inconvénient majeur de pouvoir échouer sans possibilité de succès si l'architecture n'a plus de registres libres ;
- la seconde manière consiste à ajouter des mémorisations pour chacun des cycles précédents la génération de la donnée. Concrètement, il faut ajouter une recherche pour ce type de nœud de donnée dans l'étape qui ajoute des mémorisations pour les nœuds non ordonnançables mais ayant leur résultat déjà utilisé. Cette manière nécessite de modifier la méthode de projection, mais doit avoir un meilleur taux de succès car elle ne nécessite pas qu'un registre soit libre durant tous les cycles entre l'actuel et le dernier car la donnée peut se déplacer dans l'architecture.

Pour que la donnée soit accessible par le nœud *PHI*, les possibilités suivantes sont disponibles quel que soit le premier choix :

- le nœud *PHI* contraint à être placé de manière à pouvoir accéder au registre où a été fixé sa valeur d'entrée. Si elle est disponible, l'algorithme d'assignation empêche l'utilisation de ce registre pour le reste du BB. Si ce n'est pas possible, alors la seconde possibilité est utilisée ;
- le placement du nœud *PHI* n'est pas contraint, mais il faut mémoriser la valeur d'entrée (*i.e.* ajouter des nœuds de mémorisation) tant qu'elle n'est pas placée au même endroit que sa « version » déjà assignée. L'idée est de profiter de la mobilité des opérations de mémorisation et de la recherche en largeur des *mappings* pour trouver rapidement une solution. Mais il n'est pas garanti que ce moyen converge rapidement et cela aura pour conséquence d'augmenter la latence finale.

4.2.1.4 Évolution des priorités en fonction des politiques retenues

Les différentes politiques de gestion des spécificités des CDFGs ont un impact sur les fonctions de priorités des nœuds.

Les instructions de sauts

L'instruction de saut (si elle est représentée dans le BB) doit être exécutée en dernier. Selon les capacités du CGRA matériel, cette instruction doit être exécutée seule dans un cycle ou bien peut être réalisée en parallèle d'autres opérations (qui seront généralement des *stores* ou des calculs sur des variables qui seront utilisées par des nœuds *PHI*). Dans le cas d'une instruction de saut conditionnel, il faut que l'opération qui produit le résultat du test soit située à proximité

de l'instruction de saut pour qu'elle puisse communiquer son résultat. Mais selon la réalisation matérielle, ce passage de résultat peut utiliser le registre d'état de l'ALU et donc imposer un placement sur la même tuile pour l'opération de comparaison et le saut. De plus, cela interdit l'ajout de cycles intermédiaires (*i.e.* de mémorisations). Il faut alors adapter les priorités de certains nœuds en conséquence. En ordonnancement arrière, l'instruction de saut doit donc être la plus prioritaire. L'opération qui produit le résultat utilisé par un saut conditionnel doit pouvoir elle aussi avoir une plus grande priorité, malgré son éventuelle mobilité.

Les variables partagées

Les trois politiques de gestion des variables partagées n'ont pas les mêmes conséquences sur la priorité des nœuds. La politique de réservation de registres nécessite que les nœuds générant des variables partagées soient prioritaires de manière à trouver plus facilement des registres libres pour ces variables. De plus elle impose de traiter en priorité les BBs qui produisent des variables partagées de manière à connaître leurs emplacements lors de la projection des autres BBs.

La deuxième politique de gestion par explicitation des liens de dépendances impose un traitement particulier au premier cycle d'exécution ainsi qu'au dernier de chaque BB. Pour le dernier cycle d'exécution, c'est-à-dire le premier cycle d'ordonnancement (en ordonnancement arrière), il faut effectuer une étape de « pré-allocation » des registres contenant des variables partagées et traiter en priorité les nœuds des opérations se situant au dessus. Pour le premier cycle, c'est-à-dire le dernier lors de la projection, il faut aussi privilégier les mémorisations des variables partagées et éviter ainsi d'augmenter la latence du BB pour placer ces variables dans les bons registres.

La troisième politique de gestion par *store/load* systématique ne fait qu'ajouter des accès à la mémoire et n'engendre donc pas de changement de priorité des nœuds.

Les nœuds *PHI*

Appliquer la politique de gestion des nœuds *PHI* par variable pseudo-partagée revient au cas des variables partagées décrit précédemment. Si la politique de restriction de placement est utilisée, alors il faut augmenter la priorité des nœuds produisant la variable qui sera utilisée par le nœud *PHI* lors de la prochaine exécution du BB. En effet, plus ces nœuds sont traités rapidement et moins il y a de chance que la projection échoue par impossibilité de trouver un registre disponible. *A contrario*, les nœuds *PHI* doivent avoir une priorité moindre pour deux raisons :

- la première est que plus le processus de projection avance, moins il y a de nœuds à placer et de ce fait plus il y a de chance de pouvoir placer le nœud *PHI* de sorte qu'il utilise directement le bon registre ;
- la seconde est que si le BB utilise rapidement la donnée par l'opération *PHI*, le registre la contenant sera disponible pour le reste du code, facilitant le placement.

4.2.1.5 Influence de l'ordre de traitement des Basic Blocks

Le flot que nous proposons pour gérer les CDFGs se base sur celui décrit au chapitre précédent qui ne considère qu'un seul ensemble de nœuds. L'étendre pour traiter l'équivalent de plusieurs DFGs indépendants ne pose pas de problème du fait de leur indépendance. Mais les BBs ne sont pas indépendants *a priori*. L'ordre dans lequel les BBs sont traités peut donc avoir de l'importance en particulier sur le nombre de solutions partielles que la méthode va générer. Si les BBs sont réellement indépendants, alors à la fin de chacun d'eux, il est possible de repartir à zéro.

Les BBs peuvent être considérés comme indépendants uniquement si la politique de *store/-load* systématique est utilisée pour les variables partagées et pour les nœuds *PHI*. Dans ce cas, l'ordre de traitement des BBs n'a pas d'impact sur les performances de la méthode.

Dans le cas d'ajout de nœuds de mémorisation, il n'est pas possible de remettre à zéro le nombre de solutions au début d'un BB. Le CDFG doit être traité comme un tout pour pouvoir faire les liens entre chaque variante de placement trouvée pour les variables. Comme à l'intérieur d'un BB, les premiers nœuds placés sont ceux qui seront exécutés en dernier (ordonnancement en arrière), il faudra aussi traiter les BBs en ordre inverse. De ce fait, il sera possible de construire l'arbre des solutions en entier, mais avec potentiellement des problèmes de passage à l'échelle en termes de nombre de solutions.

Dans le cas de réservation de registres fixes, le nombre de variantes est réduit car il n'y a que les variantes possibles au moment du placement initial de la variable qui existent dans l'arbre de solution, mais cela constitue malgré tout un ensemble de départ à partir duquel il sera possible de trouver toutes les solutions possibles. Contrairement au cas précédent, il n'est pas nécessaire de partir de l'ensemble des solutions précédemment trouvées, mais simplement de connaître l'ensemble des registres réservés par un ensemble de solutions (une sorte de trace de réservation).

4.2.2 Flot retenu

Pour intégrer les CDFGs dans notre flot, nous avons implémenté une passe qui explicite et ajoute l'opération de saut de chaque BB et modifie les fonctions de priorité de manière à correctement les gérer. Concernant les variables partagées, le choix parmi les trois politiques s'est porté sur celle du *store/load* systématique et ce pour deux raisons principales :

- la première raison est la simplicité de sa mise en œuvre. En effet elle ne nécessite pas de modifier en profondeur la méthode mais simplement d'ajouter une passe indépendante qui peut être validée seule sans remettre en cause les résultats du reste de la méthode ;
- la seconde raison vient du contexte d'utilisation de la méthode : la tolérance aux fautes. L'idée est de véritablement pouvoir considérer que les différents BBs sont indépendants. Cette indépendance permet en cas d'apparition d'une faute permanente dans une tuile utilisée par un seul BB de ne chercher une nouvelle configuration que pour ce BB. Plus généralement, mettre les variables partagées dans la mémoire centrale, malgré le coût en bande passante, permet de reprendre une exécution interrompue par une reconfiguration à l'endroit où elle s'était arrêté (entre deux BBs) et non pas au début à cause de variables perdues dans l'architecture. Autrement dit, c'est un moyen de sauvegarder le contexte de l'exécution, ce qui peut s'avérer très utile pour faire de la tolérance aux fautes.

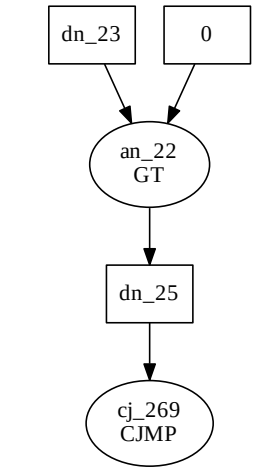
Enfin, concernant les nœuds *PHI*, notre choix s'est aussi porté sur les *stores/loads* systématiques pour les mêmes raisons que pour les variables partagées. De ce fait, l'ordre de traitement des BBs n'a pas d'importance car ils sont rendus indépendants. La figure 4.3 illustre le résultat pour l'algorithme 4 de l'application des différentes politiques de gestions des spécificités du CDFG.

Le pseudo-code de l'algorithme 5 donne les différentes étapes du flot permettant de générer les solutions de *mappings* pour une application représentée par un CDFG. Il se compose de quatre passes principales :

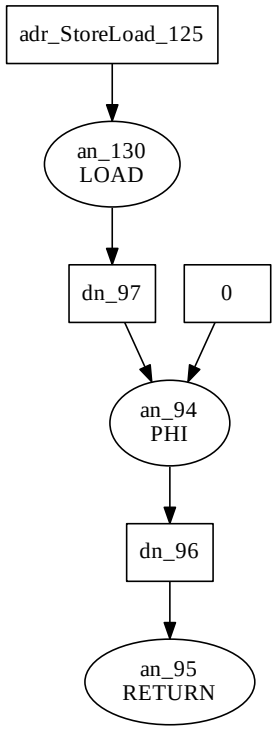
1. génération du CDFG à partir du code C de l'application ;
2. ajout des instructions de saut ;
3. ajout des *loads* et des *stores* pour les variables partagées et pour les nœuds *PHI* ;
4. pour chaque BB : génération de l'ensemble des *mappings*.

Algorithme 5 Flot de projection multiple pour les CDFGs

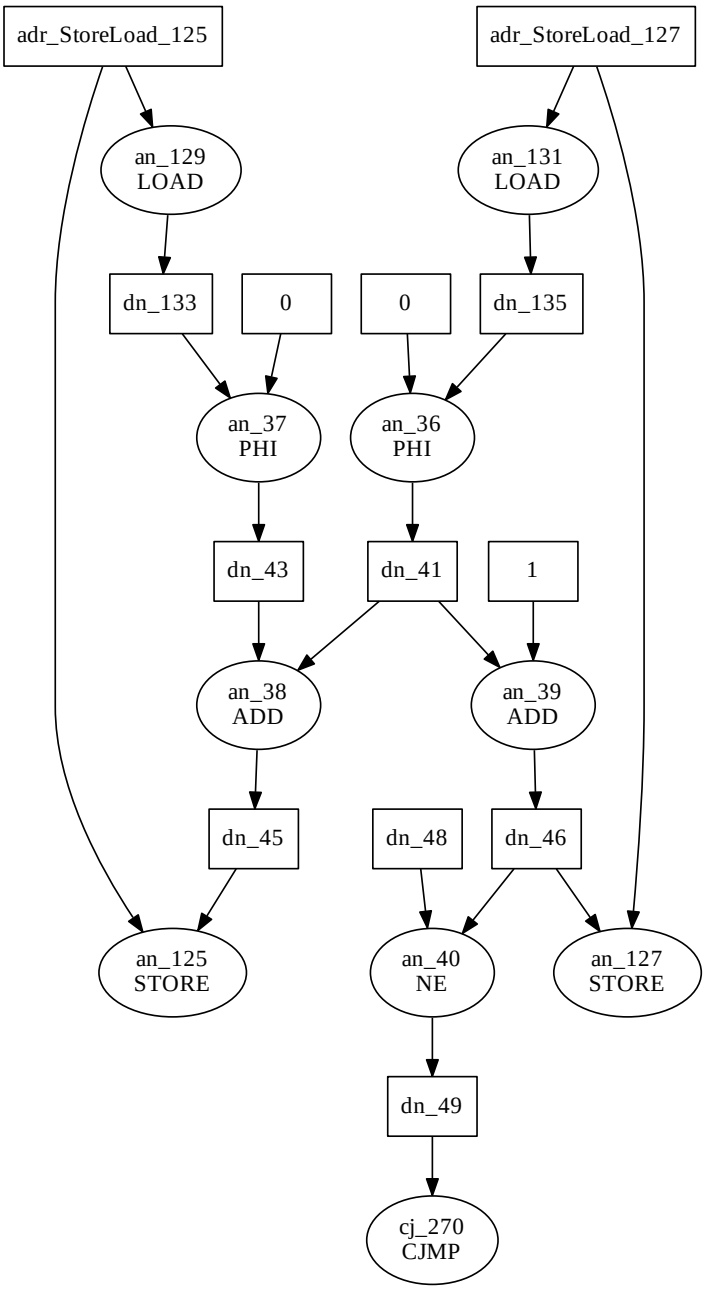
1: $cdfg$ = Compilation($CodeC$, $OptionsCompilation$)
2: $cdfg$ = PasseAjoutInstructionsSautExplicites($cdfg$)
3: $cdfg$ = PasseAjoutStoreLoad($cdfg$)
4: **pour** $bb \in cdfg$ **faire**
5: $listeMappings$ += MultiProjection(bb, $Archi$, $contraintes$)
6: **fin pour**



(a) Basic Block bb_35.



(c) Basic Block bb_116.



(b) Basic Block bb_93.

Figure 4.3 – CDFG de l'algorithme 1 après applications des différentes politiques de gestion retenues pour le flot.

4.2.3 Résultats

Le flot décrit précédemment a été implémenté en Java en se basant sur le flot pour les DFGs. Ces performances pour les différentes métriques sont quasi identiques à celles obtenues par le flot DFG pour des DFGs possédant le même nombre de nœuds. Comme il n'existe pas dans l'état de l'art de CGRA exécutant en interne des CDFGs, il n'est pas possible de comparer les performances de ce flot avec l'existant. Les passes supplémentaires ajoutent des nœuds, en particulier des opérations d'accès à la mémoire, mais cela ne pose pas de problème particulier à l'algorithme car dans un CDFG, le nombre de nœuds par BB est petit comparé à celui que l'on trouve dans un code intégralement déroulé. En effet, une FFT sur 64 points intégralement déroulée sous forme de DFG comporte plus de 3 000 nœuds d'opération. Alors qu'une FFT sur 1024 points représentée sous forme de CDFG ne comporte qu'une dizaine de BBs possédant en moyenne seulement 7 nœuds chacun.

Le nombre de BBs ainsi que le nombre de nœuds par BB sont étroitement liés aux options de compilation du *front-end* GCC. Le tableau 4.1 présente pour différents codes le nombre de BBs ainsi que le nombre de nœuds par BB pour les options de compilation O2 et O3 qui correspondent au niveau intermédiaire et au plus haut niveau d'optimisation. Les résultats pour les options inférieures (O0 et O1) ne présentent pas beaucoup d'intérêt étant donné le nombre de BBs ne comportant qu'une seule opération. On observe que généralement, le nombre de nœuds par BB est plus grand quand l'option O3 est activée. En effet, comme expliqué dans la documentation de GCC, cette option va avoir tendance à dérouler entièrement ou partiellement une partie des boucles pour accélérer l'exécution. Cependant, le nombre de BBs ne diminue pas forcément et peut même augmenter avec l'option O3.

Le nombre de nœuds des BBs des CDFGs pourrait faire penser que la méthode stochastique du flot n'est plus nécessaire, cependant, il existe des codes C qui, même avec des options de compilation favorisant un grand nombre de petits BBs, engendrent des BBs possédant parfois plusieurs centaines de nœuds comme par exemple le code de la Transformée en Cosinus Discrète Inverse (IDCT) utilisé. Compilé en O2, il génère 12 BBs dont un à 98 nœuds et un autre à 117 nœuds. Compilé en O3, il génère 5 BBs dont un à 153 nœuds et un autre à 171 nœuds. Il est alors nécessaire de recourir à l'élagage aléatoire pour que la méthode trouve des solutions.

Le tableau 4.2 présente les résultats pour un code de FFT sur 1024 points pour différentes tailles d'architecture et différentes contraintes. Pour ce code, l'élagage aléatoire avec un nombre de référence de 10 000 a été utilisé. Pour les « petits BBs » (2, 3, 4, 6, 13, 14 et 19) le nombre de *mappings* différents n'est pas affecté par cet aspect aléatoire et reste constant. Il en est de même pour les BBs 7, 11, 15, 16, 17 et 18. En revanche, pour les autres BBs, une variabilité due à l'aléatoire allant jusqu'à quelques centaines de solutions différentes a été constatée. Ce nombre de solutions différentes reste cependant conséquent (jusqu'à 205 650 dans cet exemple), ce qui est l'objectif recherché, et nous verrons dans la partie 4.3.2 comment ces solutions différentes sont gérées et combinées pour permettre de reconfigurer dynamiquement et rapidement une architecture présentant des fautes permanentes.

Tableau 4.1 – Influence des options de compilation sur le nombre et la taille des BBs.

Nombre de basic blocks

	ADPCM	Convolution	FFT	IDCT	Masque flou	Sobel	Moyenne 14 codes
O2	27	5	13	12	7	29	12,8
O3	24	1	17	5	1	33	8,8

Nombre moyen de nœuds par basic block

	ADPCM	Convolution	FFT	IDCT	Masque flou	Sobel	Moyenne 14 codes
O2	11,1	9	7,1	22	9,3	5,3	12,8
O3	16,8	62	14,8	65,8	50	9,3	22,9

Tableau 4.2 – Nombre de *mappings* différents pour les BBs d'une FFT 1024 (le nombre de nœuds du BB est noté entre parathèses).

Taille CGRA	Contrainte nb tuiles	bb1 (14)	bb2 (2)	bb3 (2)	bb4 (1)	bb5 (14)	bb6 (2)	bb7 (3)	bb8 (4)	bb9 (6)	bb10 (25)	bb11 (5)	bb12 (11)	bb13 (1)	bb14 (2)	bb15 (2)	bb16 (3)	bb17 (2)	bb18 (4)	bb19 (1)
2×2	2	4	4	4	4	4	4	6	6	4	4	4	4	4	4	6	6	6	6	4
	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	6	6	6	4	4
	4	1	4	4	4	1	4	4	1	4	1	4	1	4	4	6	6	6	4	4
3×3	2	18	9	9	9	18	9	36	36	18	18	18	18	9	9	36	36	36	36	9
	3	42	9	9	9	42	9	84	84	78	42	18	42	9	9	36	36	36	84	9
	4	81	9	9	9	81	9	84	126	186	81	90	81	9	9	36	36	36	84	9
	5	162	9	9	9	162	9	84	126	258	117	126	117	9	9	36	36	36	84	9
	6	243	9	9	9	198	9	84	126	258	84	126	117	9	9	36	36	36	84	9
	8	249	9	9	9	276	9	84	126	258	9	126	117	9	9	36	36	36	84	9
	9	276	9	9	9	249	9	84	126	258	9	126	117	9	9	36	36	36	84	9
4×4	2	32	16	16	16	32	16	120	120	32	32	32	32	16	16	120	120	120	120	16
	3	96	16	16	16	96	16	560	560	352	96	64	96	16	16	120	120	120	560	16
	4	280	16	16	16	280	16	560	1 820	928	280	256	280	16	16	120	120	120	560	16
	5	2 168	16	16	16	2168	16	560	1 820	1 120	512	448	768	16	16	120	120	120	560	16
	6	5 688	16	16	16	5 704	16	560	1 820	2 016	384	448	1 536	16	16	120	120	120	560	16
	8	16 152	16	16	16	12 296	16	560	1 820	2 016	2 440	448	5 984	16	16	120	120	120	560	16
	12	18 832	16	16	16	15 800	16	560	1 820	2 016	4 584	448	7 648	16	16	120	120	120	560	16
	16	18 084	16	16	16	15 152	16	560	1 820	2 016	3 504	448	6 704	16	16	120	120	120	560	16
5×5	2	50	25	25	25	50	25	300	300	50	50	50	50	25	25	300	300	300	300	25
	3	150	25	25	25	150	25	2 300	2 300	1 000	150	50	150	25	25	300	300	300	2 300	25
	4	475	25	25	25	475	25	2 300	12 650	2 650	475	400	475	25	25	300	300	300	2 300	25
	5	7 800	25	25	25	7 800	25	2 300	12 650	3050	410	800	1 510	25	25	300	300	300	2 300	25
	6	25 950	25	25	25	22 100	25	2 300	12 650	4 250	250	800	3 010	25	25	300	300	300	2 300	25
	8	106 785	25	25	25	79 960	25	2 300	12 650	4 250	5 350	800	12 060	25	25	300	300	300	2 300	25
	12	193 650	25	25	25	134 585	25	2 300	12 650	4 250	2 700	800	16 560	25	25	300	300	300	2 300	25
	16	205 650	25	25	25	134 860	25	2 300	12 650	4 250	4 800	800	14 460	25	25	300	300	300	2 300	25
	25	203 460	25	25	25	136 810	25	2 300	12 650	4 250	5 600	800	19 310	25	25	300	300	300	2 300	25

4.2.4 Limitations et perspectives du flot actuel

Dans cette section, nous avons vu comment étendre le flot défini au chapitre précédent pour l'utiliser avec des applications possédant un flot de contrôle et modélisées par un CDFG. Pour chacune de ses spécificités, nous avons proposé plusieurs politiques de gestion. Notre choix s'est arrêté sur celles qui présentent *a priori* le meilleur comportement dans un contexte de tolérance aux fautes. L'impact sur les performances de ces différentes politiques devra faire l'objet de travaux futurs.

Notre flot permet donc d'effectuer de la projection multiple de CDFG sur CGRA. Il n'a pas pour objectif d'accélérer des boucles et n'est pas tel quel en mesure d'effectuer la projection d'un cœur de boucle pipeliné. Son fonctionnement interne n'est cependant pas incompatible. Pour pouvoir gérer les boucles pipelinées, il faudrait modifier le flot de la manière suivante :

- ajouter un signalement (généralement sous forme de « pragma » dans le code) permettant de connaître quelle(s) partie(s) du code doit (doivent) être accélérée(s) ;
- utiliser des transformations de code pour que les parties de code à accélérer ne soient constituées que d'un seul BB. Ceci peut être fait de diverses manière comme dérouler des boucles intermédiaires ou exécuter les deux parties d'un « *if ... then ... else* » et de choisir la bonne valeur avec une opération *PHI* etc. ;
- pour chacun des BBs à accélérer, calculer la valeur minimale de l'II ;
- générer les modulo-DFGs [Rau, 1994] correspondant aux IIs.
- Déterminer les *mappings* possibles pour ces modulo-DFGs de la même manière que dans le chapitre précédent (3.2.4) en incrémentant éventuellement l'II si il n'y a pas de solution comme cela est fait dans [Hamzeh et al., 2012, Hamzeh et al., 2013] ;
- pour le reste des BBs, le flot précédemment décrit est directement applicable.

L'obtention des *mappings* pour des modulo-DFGs n'est cependant pas si simple et pourrait être effectuée de deux manières différentes qu'il faudrait comparer. La première manière consisterait à vraiment effectuer la projection du modulo-DFG avec notre flot de projection. Pour les nœuds qui ne sont pas sur le chemin critique, il serait possible de leur appliquer toutes les transformations qui ont été détaillées dans notre flot à condition de bien respecter la condition de rebouclage (placement à l'instant initial identique à celui détaillé de l'II pour les nœuds identiques). On obtiendrait alors un modulo-DFG qui n'effectue pas forcément exactement les mêmes opérations pour des occurrences partageant l'exécution, mais simplement une exécution similaire d'une récurrence à une autre. Dans le cas où un nœud sur le chemin critique n'a pas de solution de placement, la seule transformation possible est le recalcul. En effet, mémoriser un nœud sur le chemin critique décale l'intégralité des exécutions et nécessite de recommencer le processus en entier. Si aucune solution n'est trouvée, il faut augmenter l'II.

La seconde manière de faire serait de réaliser la projection des BBs à accélérer avec une contrainte en termes de nombre de tuiles utilisables dépendantes de la valeur de l'II et du nombre de ressources pour obtenir des BBs précontraints, déjà en partie transformés. Ce premier résultat permettrait d'affiner la valeur de l'II si besoin. Ensuite les modulo-DFGs seraient créés à partir de ces BBs et le flot serait relancé. L'intérêt de supprimer les nœuds de mémorisations simples pour permettre une relative malléabilité du graphe (les recalculs et les mémorisations permettant de diminuer le nombre d'arcs sortants devraient cependant être conservés pour éviter au maximum de devoir relancer le processus complet en cas d'échecs) serait à étudier. Cette manière présente l'avantage de pré-contraindre le graphe facilitant sa projection et d'éventuellement directement augmenter la valeur de l'II si elle est irréaliste. Mais elle nécessite de lancer au moins deux fois le processus de projection.

Ces changements, qui nécessitent d'être implémentés, testés et évalués, feront eux aussi l'objet de développements futurs.

4.3 Flot pour la tolérance aux fautes

Dans cette section, une méthode permettant d'augmenter le nombre de solutions différentes est présentée. Puis nous verrons comment gérer l'ensemble des *mappings* en ligne pour choisir la configuration courante du CGRA.

4.3.1 Amélioration de la diversité des *mappings*

Les deux flots présentés précédemment permettent d'obtenir un grand nombre de solutions de *mappings* pour un CGRA donné et une contrainte de nombre de tuiles utilisables. Toutes ces solutions ont la même latence et utilisent généralement le même nombre de tuiles (sauf si le DFG ou le BB est très petit ou si l'architecture possède un très grand nombre de ressources). Cependant, cela s'avère insuffisant pour pouvoir fonctionner le plus longtemps possible en changeant de configuration. Cette section commence par présenter les paramètres pertinents permettant d'influer sur la diversité des solutions, puis détaille deux algorithmes (un de diversification et un d'intensification) permettant d'utiliser ces paramètres pour augmenter la diversité des solutions.

4.3.1.1 Paramètres favorisants la diversification

Le premier paramètre dont il a déjà été question est le nombre maximal de tuiles que la projection est autorisé à utiliser. Lorsque ce paramètre n'est pas contraint, la projection peut s'étendre au maximum, en étant limité uniquement par la taille du CGRA, le réseau d'interconnexion, l'éventuelle contrainte de nombre simultané d'accès à la mémoire et ses dépendances de données.

Deux autres paramètres ont été retenus pour favoriser la diversité des *mappings* :

— le nombre maximal de voisins avec lesquels chaque tuile est autorisée à communiquer ;
— et le nombre de tuiles autorisées à utiliser ce nombre maximal.

Ils permettent tous deux de restreindre l'utilisation du CGRA en limitant l'utilisation du réseau d'interconnexion. Par exemple, sur un CGRA de largeur et hauteur supérieur à cinq tuiles possédant une interconnexion de type mesh-2D, la figure 4.4 présente les quatre réseaux existant possédant cinq tuiles. Ils ont les caractéristiques suivantes :

— celui de gauche possède une seule tuile avec quatre voisins ;
— ceux du milieu et du bas à droite possèdent une seule tuile à trois voisins ;
— celui d'en haut à droite possède trois tuiles à deux voisins.

Si le CGRA possède une largeur inférieure ou égale à cinq tuiles et qu'il possède un mesh torique alors d'autre réseaux sont atteignables comme une ligne bouclée sur elle-même où toutes tuiles ont deux voisins. De même, si le CGRA est un 3×3 torique alors il y aurait d'autres versions du réseau de gauche de la figure 4.4 : avec la tuile A connectée à la tuile E et/ou la tuile B connectée à la tuile D. L'annexe B présente les différents réseaux atteignables possédant huit tuiles dans un mesh-2D simple à quatre voisins.

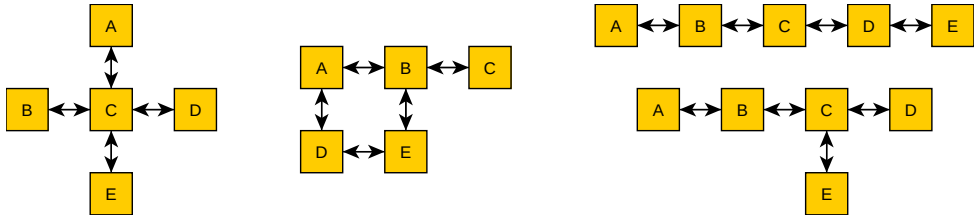


Figure 4.4 – Les quatre réseaux différents possibles pour 5 tuiles sur un CGRA dont les dimensions sont supérieures à 5 tuiles.

4.3.1.2 Les boucles de diversification

Les trois paramètres décrit précédemment sont utilisés dans l'algorithme 6 afin de favoriser au mieux la diversification des *mappings*.

Algorithme 6 Diversification des *mappings*

```
1: Initialisation(CDFG, Archi)
2: pour nbTuiles = NbTuilesArchi à 1 faire
3:     pour nbMaxVoisins = MaxVoisins à 2 faire
4:         pour nbMaxTuilesNbMaxVoisins = NbTuilesArchi à 1 faire
5:             listeMappings += MultiProjection(CDFG, Archi, contraintes)
6:             SuppressionRedondance(listeMappings)
7:         fin pour
8:     fin pour
9: fin pour
```

La première itération de l'algorithme correspond à celle sans contrainte particulière. La variable $NbTuilesArchi$ vaut le nombre totale de tuiles disponibles dans le CGRA et $MaxVoisins$ dépend du réseau d'interconnexion (*e.g.* 4 pour un mesh-2D ; 8 pour un mesh-X).

La fonction FindMappings() est celle qui réalise le processus de projection comme détaillé dans les chapitres précédents. La fonction SuppressionRedondance() n'est pas à la fonction d'élagage utilisée lors de la projection et doit s'exécuter en plus de cette dernière. Elle répond aux critères suivants :

- si un *mapping* utilise un réseau qui contient celui d'un autre *mapping* et possède la même latence, alors il n'est pas conservé ;
- si un *mapping* utilise un réseau qui est plus connecté que celui d'un autre *mapping* (avec les mêmes tuiles) et possède tout deux la même latence, alors il n'est pas conservé ;
- si un *mapping* est plus connecté qu'un autre *mapping* et/ou le contient et possède une latence plus faible, alors ils sont tous deux conservés ;

Dans tous les autres cas, le *mapping* est conservé. En effet, comme expliqué dans le chapitre 2, l'objectif de la méthode est triple : surface/latence/réseau. C'est pour cette raison qu'il faut conserver les *mappings* plus lents mais moins connectés et/ou plus petits.

Pour réduire le nombre d'itérations, sans perdre de généralité, il est possible d'agir de deux manières :

- connaissant la durée maximale acceptable pour l'exécution d'un DFG ou d'un BB, il est possible de sortir des boucles quand cette contrainte n'est plus respectée ;
- la connaissance des caractéristiques du réseau d'interconnexion permet d'affiner la condition d'arrêt de la boucle la plus interne.

En effet, certaines combinaisons sont redondantes. Par exemple si il y est possible d'utiliser cinq tuiles, mais que la contrainte du nombre de voisins maximal est à deux et que la limite de nombre maximum de tuiles pouvant avoir deux voisins est en dessous de trois, il ne sera pas possible d'obtenir une solution à 5 tuiles. Cette combinaison de contraintes donnera exactement les même résultats que celle avec seulement quatre tuiles autorisées et est donc redondante.

Par exemple, avec un mesh-2D simple, les formules empiriques des équations 4.3.1, 4.3.2 et 4.3.3 donnent la valeur de la borne supérieure du nombre maximal de tuiles pouvant communiquer avec le nombre de voisins maximal en fonction des deux autres variables :

– Si 4 voisins au maximum : $$borne = \begin{cases} 0 & \text{si } nbTuiles \leq 4 \\ 1 & \text{si } 5 \leq nbTuiles \leq 7 \\ \left(2 + \left\lfloor \frac{nbTuiles-8}{2} \right\rfloor\right) & \text{si } nbTuiles \geq 8 \end{cases} \quad (4.3.1)$$

$$- \text{ Si 3 voisins au maximum } : borne = \begin{cases} 0 & \text{si } nbTuiles \leq 3 \\ 1 & \text{si } nbTuiles = 4 \\ (nbTuiles - 4) & \text{si } nbTuiles \geq 5 \end{cases} \quad (4.3.2)$$

$$- \text{ Si 2 voisins au maximum } : borne = \begin{cases} 0 & \text{si } nbTuiles \leq 1 \\ (nbTuiles - 2) & \text{si } nbTuiles \geq 2 \end{cases} \quad (4.3.3)$$

Si la borne est zéro, cela signifie qu'il n'y a pas besoin d'effectuer ce tour de boucle car il est redondant.

Ces formules sont bien évidemment très dépendantes du réseau d'interconnexion et ne sont valables que dans le cas d'un réseau d'interconnexion possédant quatre voisins au maximum. En cas d'un réseau d'interconnexion torique, la borne doit être augmentée si le nombre de tuiles autorisée est supérieure à une des dimensions. Par exemple, dans le cas de deux voisins maximum, la borne ne sera plus $(nbTuiles - 2)$ mais $nbTuiles$ (qui correspond à un anneau).

Remarque

Dans le chapitre 2, il a été exclu *a priori* d'effectuer la projection d'un code par une méthode permettant de trouver un *mapping* pour chaque réseau qu'il est possible d'obtenir à partir de l'architecture initiale. Il faut maintenant vérifier que l'approche de diversification est bien meilleure que celle exclue.

L'approche proposée par diversification effectuera dans le pire des cas et sans optimisation :

— n tours de la boucle de nombre de tuiles utilisables avec n représentant le nombre de tuiles de l'architecture ;

— un nombre fixe de tours de la boucle intermédiaire dépendant du nombre de voisins disponibles dans le réseau d'interconnexion ;

— n tours de la boucle de nombre de ressources capable d'utiliser la capacité de communication définie dans la boucle de niveau supérieur.

Soit au total une complexité en $O(n^2)$. L'approche par contrainte de réseaux a une complexité en $O(2^n)$. Nous confirmons *a posteriori* que la démarche « objectif de réseaux » est bien pertinente.

4.3.1.3 Les boucles d'intensification

Après avoir effectué les boucles de diversification, il est possible d'effectuer une intensification de la recherche de *mappings* différents. Cette étape s'appuie sur les *mappings* trouvés précédemment pour chaque BB pour en obtenir d'autres.

Cette recherche se passe en deux étapes comme illustrées dans le pseudo-code 7 :

1. obtenir de nouvelles « signatures » de réseaux ;
2. réaliser la projection pour chacune d'elles.

Algorithme 7 Intensification des *mappings*

Nécessite : $\langle listeMappings_{bb} \rangle$: liste de *mappings* précédemment trouvée pour le BB considéré

1: $nouvellesSignaturesReseaux$ = ObtentionNouveauxReseaux($listeMappings_{bb}$)
2: **pour** $signatureReseau \in nouvellesSignaturesReseaux$ **faire**
3: $\langle listeMappings_{bb} \rangle$ += MultiProjection(bb, $Archi$, $signatureReseau$)
4: SuppressionRedondance($\langle listeMappings_{bb} \rangle$)
5: **fin pour**

Le principe général de cette intensification est le suivant : obtenir tous les réseaux possibles au delà d'une taille très petite de CGRA n'étant pas possible, une heuristique de détermination

de réseaux est mise en place. Elle consiste, à partir d'une solution de *mapping* déjà trouvée, à supprimer une tuile utilisée de la solution et à conserver le ou les réseaux obtenus de cette manière pour les utiliser par la suite. En d'autres termes cela revient à prédire des scénarios d'apparition de fautes permanentes sur des solutions de *mapping* déjà existantes. Par exemple, supposons que la diversification ait permis de trouver les réseaux de cinq tuiles de la figure 4.4. La première partie de l'intensification va alors pour chacun d'eux supprimer une tuile. Dans le cas du réseau de gauche, différentes variantes de réseau en « T » seront trouvées (avec des orientations différentes). Avec le réseau central, il sera possible d'obtenir un réseau carré, un « T » ou alors une ligne coudée. Les deux derniers réseaux permettront eux aussi d'obtenir un « T » ou des lignes.

Comme il n'est pas réaliste d'effectuer une synthèse pour chacun de ces réseaux particuliers, ce ne sera pas exactement le réseau qui sera donné comme paramètre à l'algorithme de projection, mais une « signature ». Cette signature est une représentation réduite et incomplète d'un réseau. Elle est composée du nombre de tuiles utilisant un certain nombre de voisins pour chaque nombre de voisins connectables possibles. Par exemple le réseau de gauche de la figure 4.4 a pour signature « 1-0-0-4 » car il possède une tuile avec quatre voisins, zéro tuile à trois et deux voisins, et quatre tuiles à un voisin. La seconde partie de l'intensification consiste à utiliser ces signatures explicites pour relancer la projection. Pour chaque signature différente, le flot de projection va obtenir l'ensemble des *mappings*. La signature suffit ici car durant le processus, si une solution en ligne est trouvée, les solutions en ligne coudée le seront aussi. De même l'orientation du « T » n'a pas d'importance dans la contrainte car toutes les orientations seront trouvées. Après ce processus, si le nombre de *mappings* différents a augmenté, il pourrait être intéressant de relancer l'intensification avec les solutions nouvelles.

Au final, après avoir exécuté les heuristiques de diversification et d'intensification, on dispose d'un grand nombre de solutions de *mappings* pour tous les BBs utilisant des réseaux différents. La section suivante présente la méthode développée pour déterminer en ligne la configuration du CGRA à partir de cet ensemble généré hors ligne.

4.3.2 Méthode de combinaison de DFGs et moteur de reconfiguration

La tolérance aux fautes permanentes par reconfiguration nécessite d'identifier la ressource fautive, puis de ne plus l'utiliser. Comme expliqué précédemment, le CDFG possède en son sein plusieurs BBs (équivalents de DFGs) qui s'exécutent de façon exclusive, c'est-à-dire un seul à fois sur l'architecture. Le processus de projection développé permet d'obtenir, pour chacun des DFGs/BBs, un ensemble de possibilités de placement ou configurations. Cet ensemble peut, selon les cas, être de taille conséquente (plusieurs dizaines voire centaines de milliers de configurations par BB) et la méthode de choix de la nouvelle configuration doit pouvoir gérer cet ensemble de configurations. Les travaux présentés ici sur le fonctionnement et l'évaluation de ce moteur de reconfiguration ont été en partie réalisés dans le cadre de la thèse de master de M. Sureshbabu Ramesh.

Le moteur de reconfiguration est un composant extérieur au CGRA qui possède les entrées suivantes : un ensemble de *mappings* pour chaque BB et la cartographie des fautes permanentes présentes sur le CGRA. Son but est de déterminer le plus rapidement possible, en cas d'apparition de faute permanente, une nouvelle configuration de l'application sur le CGRA qui n'utilise pas les ressources défectueuses et qui respecte la contrainte de latence de fonctionnement de l'application portée. Il prend en entrée l'ensemble des *mappings* générés par le flot décrit dans la section précédente, le détail de l'architecture et la latence maximale acceptable pour l'application. Au début du fonctionnement, il va configurer le CGRA avec un *mapping* valide. Puis pendant

le reste du fonctionnement, son comportement est décrit dans l'algorithme 8. À l'apparition d'une faute, il supprime les *mappings* qui utilisaient la tuile fautive et détermine, pour chaque BB utilisant cette tuile fautive, un nouveau *mapping* grâce à l'heuristique de reconfiguration. Il vérifie ensuite que la latence global est acceptable. Lorsque la méthode ne trouve pas une configuration valide, alors le fonctionnement s'arrête. Les paragraphes suivants décrivent les trois possibilités d'heuristique de sélection de *mapping* que nous avons retenus, les analysent et évaluent leurs performances.

Algorithme 8 Fonctionnement module de reconfiguration

Nécessite : $TuilesArchitectureCGRA$ La liste de tuiles disponibles
Nécessite : $\langle listeMappings_{BB} \rangle$ Ensemble de listes triées de *mappings* pour chaque BB
$tuilesValides = TuilesArchitectureCGRA$
$configurationCourante$ = ConfigurationInitial($listeMappings_{BB}$, $tuilesValides$, $LatenceMax$)
tant que $configurationCourante \neq \emptyset$ **faire**
  $idTuileDefectueuse$ = RechercheIdTuileFautive()
  si $idTuileDefectueuse \neq -1$ **alors**
    SuppressionMappingsUtilisantTuile($\langle listeMappings_{BB} \rangle$, $idTuileDefectueuse$)
    $BBaReconfigurer$ = BBUtilisantTuile($configurationCourante$, $idTuileDefectueuse$)
    pour tout $bb \in BBaReconfigurer$ **faire**
      $mapping_{BB}$ = ObtentionMappingCandidat($listeMappings_{bb}$)
      si $mapping_{BB} == \emptyset$ **alors**
        $configurationCourante \leftarrow \emptyset$
        rupture boucle // Plus de *mapping* pour ce BB
      sinon
        $configurationCourante$.fixer($bb, mapping_{bb}$)
      fin si
    fin pour
  fin si
  si Latence($configurationCourante$) $> LatenceMax$ **alors**
    $configurationCourante \leftarrow \emptyset$ // Dépassement de la latence max
  fin si
fin tant que
retourne 1

4.3.2.1 Les différentes politiques de reconfigurations

Le module de reconfiguration dispose d'un ensemble de *mappings* qui n'ont pas la même latence, qui n'utilisent pas le même nombre de tuiles et l'interconnexion de la même façon. L'objectif de ce module est de déterminer la combinaison de *mappings* qui sera choisie. Pour cela, chaque tuile du CGRA se voit attribuer un identifiant. Cet identifiant est un nombre entier affecté dans l'ordre naturel. Ainsi, pour un CGRA de taille $S = W \times H$, l'identifiant d'une tuile de coordonnées $tuile = (x, y)$ est $id_{tuile} = x + y * H$. Les tuiles disponibles sont représentées sous forme d'une liste avec pour chaque tuile un état associé défini par l'équation 4.3.4.

$$EtatTuile_{id} = \begin{cases} 0 & \text{si fautive} \\ 1 & \text{sinon} \end{cases} \tag{4.3.4}$$

Les *mappings* sont d'abord triés par latence croissante. Ensuite chaque politique définit l'algorithme de tri pour les *mappings* ayant la même latence.

Politique de recouvrement maximum

L'objectif de la première heuristique est de maximiser la réutilisation des tuiles. Ainsi le nombre de tuiles globalement utilisées par le CDFG est le plus petit possible. L'idée sous-jacente à cette politique est que plus le *mapping* du CDFG est petit et moins il a de risque d'utiliser une

tuile qui sera victime d'une faute. Ceci a un double impact. D'une part il sera moins souvent nécessaire de reconfigurer le CGRA. D'autre part, lorsque une faute affecte une tuile utilisée, il y a plus de BBs qui sont susceptibles de l'utiliser et donc il y aura davantage de BBs à re-*mapper*, ce qui augmentera le temps de reconfiguration.

Chaque *mapping* de BB est caractérisé par ses tuiles utilisées. Un *mapping* est représenté par une liste de tuiles classées par leurs identifiants. Pour utiliser au maximum les mêmes tuiles, le tri des *mappings* utilise la méthode de comparaison décrite dans l'algorithme 9 et donc cherche à placer en premier les *mappings* qui utilisent le moins de tuiles, puis les *mappings* qui utilisent les tuiles avec l'identifiant le plus petit (donc les tuiles « en haut à gauche »).

Politique de recouvrement minimum

L'objectif de la deuxième heuristique est de minimiser la réutilisation des tuiles. L'idée de cette politique est que lors de l'apparition d'une faute, il y aura peu de BBs à re-*mapper*, ce qui diminue le temps de reconfiguration. La contrepartie est que la probabilité d'avoir à re-*mapper* un BB est plus importante. Minimiser le recouvrement est idéalement ne pas avoir de recouvrement du tout. Or la résolution de ce problème est équivalente au problème du rangement du sac à dos qui est NP-complet. Pour réaliser cette politique, nous avons donc défini l'heuristique suivante : pour chaque BB du CDFG, un point d'ancrage préférentiel est défini. Les points d'ancrage sont répartis équitablement sur toute la surface du CGRA. Les DFG sont espacés les uns des autres par un *offset*, qui est la taille du CGRA divisé par le nombre de DFG, équation 4.3.5.

$$offset = \left\lceil \frac{S}{N_{BB}} \right\rceil \tag{4.3.5}$$

avec : S : la taille du CGRA (largeur x hauteur) et N_{BB} : le nombre de BB dans le CDFG. Chaque BB se voit affecter une tuile d'ancrage, équation 4.3.6, identifiée par son numéro. Si le nombre de BB est supérieur à la taille du CGRA, plusieurs BBs se voient donc affecter la même tuile d'ancrage.

$$\forall i \in \langle BB \rangle, Ancrage_{BB_i} = (i + offset) \bmod S \tag{4.3.6}$$

Politique aléatoire

La dernière politique n'a pas spécialement d'objectif particulier en termes d'utilisation du CGRA. Elle choisit la première combinaison de *mappings* qui respecte la latence et qui n'utilise pas les tuiles défectueuses.

4.3.2.2 Analyse

Pour illustrer la différence de résultat que peuvent avoir ces différentes heuristiques, considérons les *mappings* de cinq BBs de la figure 4.5. On considère que le processus de projection a généré ces *mappings* pour chaque origine possible et qu'il est donc possible de les déplacer sur l'architecture. La figure 4.6 donne, sur un CGRA 5×5, le résultat de l'application de la politique de recouvrement maximal (4.6a) et celui du recouvrement minimal (4.6b). Dans le premier cas, la réutilisation des tuiles est très forte (allant jusqu'à 4) et globalement très peu de tuiles sont utilisés (8 sur 25). Dans le second cas, la réutilisation des tuiles est nulle et le *mapping* du CDFG utilise 22 tuiles sur les 25 de l'architecture. En considérant que la probabilité d'apparition d'une faute est homogène sur le composant, la probabilité d'avoir une faute dans le cas de recouvrement maximal est 2,75 fois plus faible que celle du recouvrement minimal. La politique aléatoire devrait donner un résultat intermédiaire.

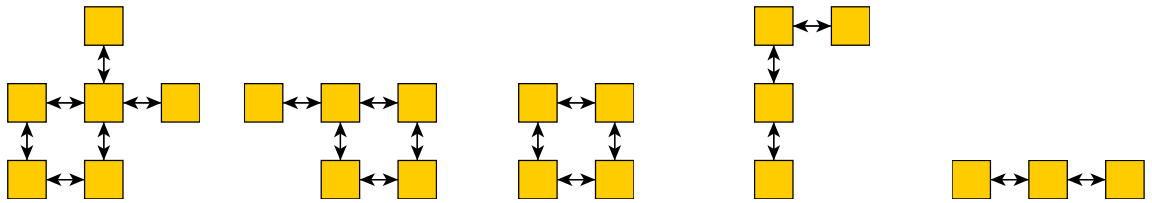
Le temps de reconfiguration $T_{reconfig}$ d'une heuristique peut être approximé par l'équation $T_{reconfig} = T_{heuristique} \times nbBB$, où $T_{heuristique}$ est le temps d'exécution de l'heuristique pour un BB et $nbBB$ est le nombre de BBs à reconfigurer en cas de fautes.

Algorithme 9 Comparaison entre deux mappings selon la politique de surface minimum.

Nécessite : m_1, m_2 Deux mappings à comparer
si nbTuiles(m_1) < nbTuiles(m_2) **alors**
    retourne m_1
fin si
si nbTuiles(m_1) > nbTuiles(m_2) **alors**
    retourne m_2
fin si
pour $i = 0$ **à** nbTuiles(m_1) **faire**
    $id_{m_1} = tuile(m_1, i)$
    $id_{m_2} = tuile(m_2, i)$
    si $id_{m_1} < id_{m_2}$ **alors**
        retourne m_1
    sinon
        retourne m_2
    fin si
fin pour

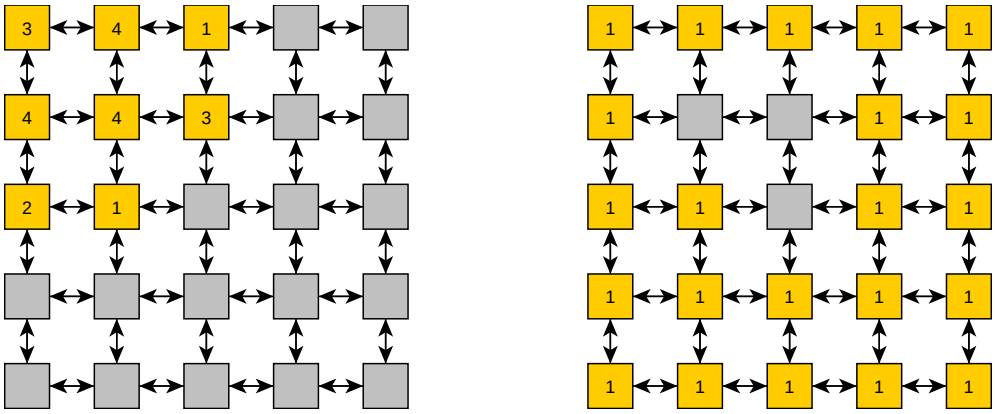
Algorithme 10 Comparaison entre deux mappings selon la politique de surface maximum

Nécessite : m_1, m_2 Deux *mappings* à comparer
Nécessite : a le point d'ancrage du BB
si $a \in$ tuiles(m_1) **alors**
    si $a \in$ tuiles(m_2) **alors**
        si nbTuiles(m_1) > nbTuiles(m_2) **alors**
            retourne m_1
        fin si
        si nbTuiles(m_1) < nbTuiles(m_2) **alors**
            retourne m_2
        fin si
    sinon
        retourne m_1
    fin si
sinon
    si $a \in$ tuiles(m_2) **alors**
        retourne m_2
    sinon
        si distance(m_1,a) < distance(m_2,a) **alors**
            retourne m_1
        sinon
            si distance(m_1,a) > distance(m_2,a) **alors**
                retourne m_2
            sinon
                retourne m_1 // L'algorithme garde l'ordre prédéfini
            fin si
        fin si
    fin si
fin si



(a) *Mapping* BB1 (b) *Mapping* BB2 (c) *Mapping* BB3 (d) *Mapping* BB4 (e) *Mapping* BB5

FIGURE 4.5 – Exemple de *mappings* pour cinq BBs d'une application.



(a) Utilisation globale avec la politique de surface munimum.

(b) Utilisation globale avec la politique de surface maximum.

FIGURE 4.6 – Illustration sur un CGRA 5×5 de l'application des politiques de recouvrement minimal et maximal pour les cinq *mappings* de la figure 4.5. Les chiffres dans les tuiles représentent le nombre de *mappings* de BBs qui utilisent la tuile.

Au niveau du temps d'exécution de chaque heuristique, celle qui devrait être la plus rapide est l'aléatoire. Ensuite, les deux autres heuristiques devraient avoir des temps d'exécution similaire car le problème est de même nature, seul l'objectif qui change. Au niveau du nombre de BBs à reconfigurer en cas d'apparition d'une faute, celle qui en possèdera le moins en moyenne est l'heuristique minimisant le recouvrement ; puis la méthode aléatoire ; enfin la méthode maximisant le recouvrement. Cependant, la probabilité d'apparition d'une faute dans le *mapping* est plus faible avec l'heuristique maximisant le recouvrement. Il est donc difficile de conclure quelle sera la politique la plus rapide.

Le fonctionnement décrit précédemment n'est pas optimal car il ne fait que réagir lorsqu'une faute apparait. De manière à être plus réactif, nous proposons que ce module pré-calcule de nouvelles configurations en fonction d'hypothèse de faute lorsqu'il n'y pas pas besoin de reconfigurer le CGRA, profitant ainsi du temps entre les fautes pour améliorer sa rapidité.

Dans l'exemple de la figure 4.6a, le module de reconfiguration commence par calculer des nouvelles configurations pour les tuiles utilisées quatre fois, puis trois, puis deux et enfin ayant une seule utilisation, tant qu'il n'y a pas de tuile fautive. Ce pré-calcul favorise plus l'heuristique minimisant la surface car le temps entre deux reconfigurations est statistiquement plus long. Si une tuile non utilisée par le *mapping* du CDFG est détectée comme étant fautive, le module ne change pas la configuration du composant, mais met à jour la liste des configurations utilisables et continue le pré-calcul. Le tableau 4.3 résume les tendances attendues de ces différentes heuristiques. Il n'y a pas de méthode qui ne présente pas de défaut et celle retenue sera celle qui donne le meilleur compromis.

Tableau 4.3 – Comparaison des heuristiques de reconfiguration

Critère	Recouvrement Max	Recouvrement Min	Aléatoire
Probabilité d'apparition d'une faute	Faible	Élevée	Moyenne
Temps total de Reconfiguration	Élevée	Modéré	Faible
Capacité à anticiper	Élevée	Faible	Modéré

Remarque : Plus le nombre de tuiles de l'architecture diminue et plus les trois heuristiques vont donner des résultats similaires qui s'approchent du résultat fourni par l'heuristique maximisant le recouvrement.

4.3.2.3 Évaluation

Nous avons prototypé les trois heuristiques de reconfiguration en Java et exécuté vingt séquences d'apparition de fautes permanentes. Pour chaque séquence, l'algorithme détermine aléatoirement une tuile victime d'une faute permanente, lance le module de reconfiguration et réitère tant qu'il y a au moins une solution de *mappings* pour le CDFG. Sur de petits CGRAs, aucune différence notable n'est visible entre ces trois politiques car après seulement quelques fautes permanentes elles fournissent le même résultat.

Pour un CGRA de taille plus importante (*e.g.* 8×8), les tendances ne sont pas les mêmes pour le début du fonctionnement. La figure 4.7 présente l'évolution du nombre moyen de tuiles utilisées par le CDFG complet sur un CGRA 8×8 en fonction d'un axe représentant le temps. Ce « temps » est mesuré ici avec le nombre d'apparitions aléatoires de fautes permanentes dans l'architecture. Nous avons décidé de laisser les tirages correspondants à des tuiles déjà défectueuses dans l'axe en considérant que l'apparition d'une défaillance pouvait être approximé comme équiprobable dans toutes les tuiles quel que soit leur état. Ne comptabiliser que ceux qui engendrent une faute sur une nouvelle tuile ne permet pas d'avoir une notion temporelle.

Cette figure montre que l'heuristique minimisant la surface possède deux phases de fonctionnement distinctes. La première est située avant le trentième tirage. Durant cette phase, le nombre de tuiles que le CDFG utilise est quasi-constant. Puis la surface utilisée par la CDFG diminue. L'heuristique maximisant la surface n'a pas de phase de plateau, mais décroit systématiquement. L'heuristique aléatoire possède un comportement intermédiaire.

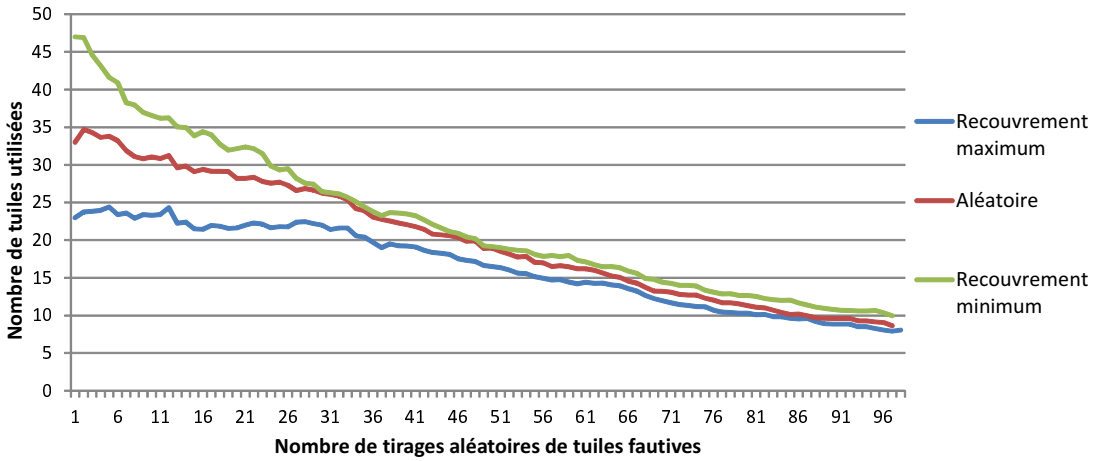


Figure 4.7 – Evolution du nombre de tuiles utilisées par les différentes politiques de reconfiguration pour une FFT 1024 sur un CGRA 8×8.

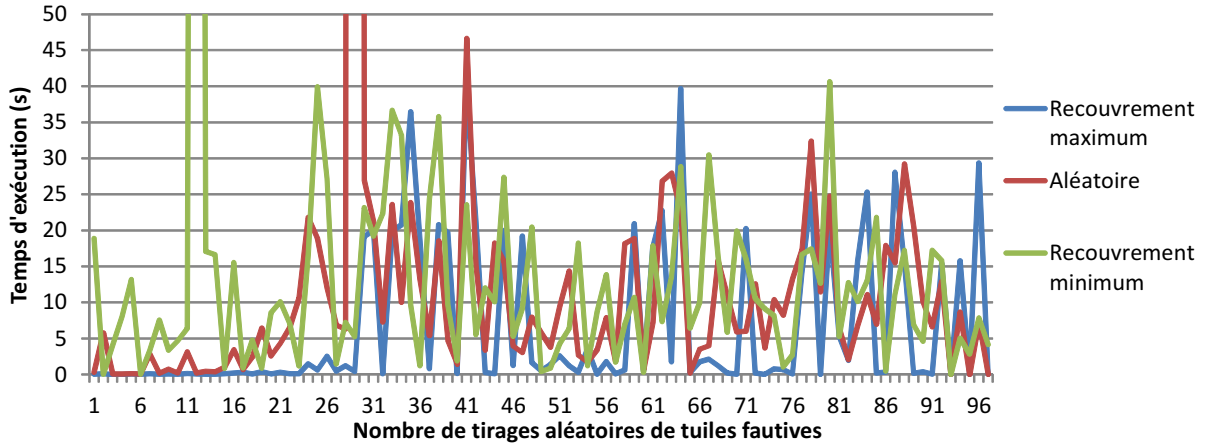


FIGURE 4.8 – Temps d'exécution des différentes politiques de reconfiguration pour une FFT 1024 sur un CGRA 8×8.

Concernant le nombre de BBs à re-*mapper*, la politique minimisant la surface est celle qui en moyenne pendant la première phase en obtient le moins. Cependant, ce résultat est trompé par le fait de considérer la moyenne car elle obtient un écart type supérieur de moitié par rapport aux autres heuristiques. En effet, son pire cas consiste à devoir changer les *mappings* de tous les DFGs alors que dans l'heuristique de maximisation de la surface par exemple, au moins au départ, seul un très petit nombre de *mappings* sont à changer.

Nous avons mesuré les temps d'exécution des heuristiques après chaque apparition d'une faute, qu'elle concerne une tuile utilisée par le *mapping* courant ou non, en utilisant la fonction Java System.nanoTime() [1]. Ces temps sont donnés dans la figure 4.8. Ils sont très variables et assez peu lisibles tels quels, c'est pourquoi nous avons lissé ces courbes en supprimant les deux valeurs aberrantes présentes (de plusieurs milliers de secondes). Le résultat de ce filtrage est donnée en figure 4.9. On constate que le temps moyen pour l'heuristique ciblant le recouvrement maximum est inférieur aux deux autres heuristiques. L'heuristique minimisant le recouvrement est celle qui obtient les moins bons résultats sur la première partie. Dans le seconde partie, à partir du trentième tirage, les trois heuristiques ont un comportement assez similaires, ce qui était prévisibles à partir de l'analyse du nombre de tuiles utilisées.

Contrairement à ce qui était attendu, l'heuristique aléatoire n'est pas spécialement plus rapide que les autres quand le nombre de tuiles valides diminue. De même, la meilleure heuristique semble être celle qui maximise le recouvrement. Les temps de reconfiguration obtenus ne sont pas révélateurs du temps réel de l'exécution car nous ne connaissons pas encore le type de processeur sur lequel s'exécutera l'heuristique.

1. Sa précision réelle fait débat, mais elle permet d'observer des tendances.

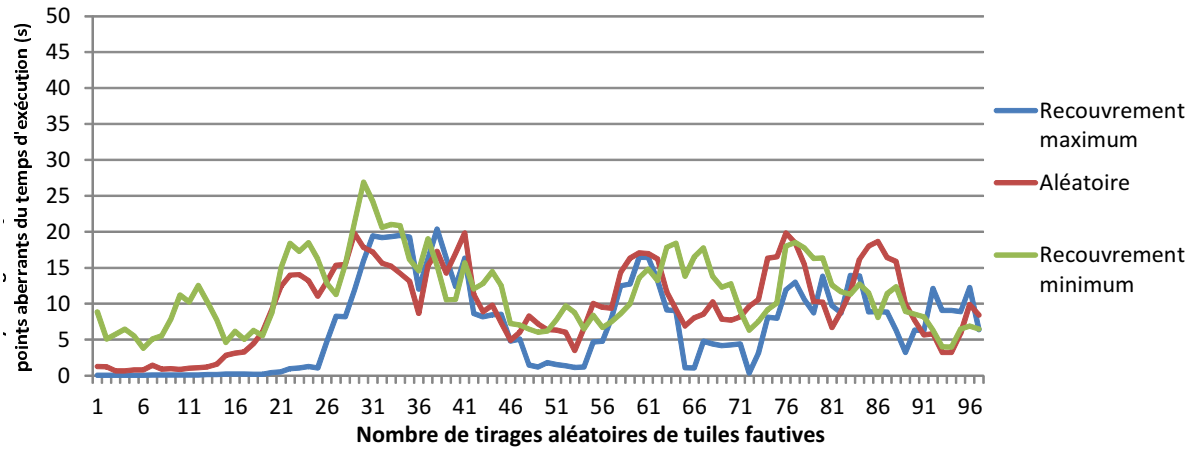


Figure 4.9 – Moyenne glissante sur cinq valeurs sans points aberrants du temps d'exécution des différentes politiques de reconfiguration pour une FFT 1024 sur un CGRA 8×8.

4.4 Conclusion

Dans ce chapitre, nous avons montré comment étendre le flot de projection multiple défini au chapitre 3 de manière à accepter des codes sources plus complexes car possédant un flot de contrôle. Les avantages et les inconvénients des différentes façons de faire ont été décrits. Les choix finaux ont été motivés par le contexte de tolérance aux fautes. Ils auraient été tout autres dans un contexte de haute performance par exemple. Des boucles de diversification et d'intensification ont été proposées pour permettre d'obtenir davantage de *mappings* différents en utilisant le flot proposé. Nous avons défini et proposé une méthode permettant de gérer la reconfiguration en ligne du CGRA lorsqu'une faute permanente est détectée en tenant compte de la grande quantité de configurations générées par les différentes boucles de contraintes. Le flot de diversification et le moteur de reconfiguration sont actuellement en cours de publication.

Chapitre 5

Un CGRA pour la tolérance aux fautes

5.1 Introduction

Les chapitres précédents ont défini un flot capable projeter un code complet sur CGRA. L'approche présentée dans le chapitre 4 suppose que l'architecture cible est capable d'exécuter une application intégrant du contrôle et intègre un ensemble de mécanismes permettant de tolérer les fautes transitoires et détecter les fautes permanentes. Dans ce chapitre, nous proposons une architecture CGRA innovante capable de gérer ce contrôle. Puis, nous analyserons les différentes moyens qui permettront de mettre en œuvre d'une part la tolérance aux fautes transitoires et d'autre part la détection de fautes permanentes dans cette architecture. Cette étude permettra de valider ou au contraire de rejeter certaines mises en œuvre. Enfin, notre proposition pour tolérer les fautes permanentes au niveau d'un système complet en se basant sur cette architecture et sans interruption de fonctionnement est détaillé.

5.2 Un CGRA pour CDFG : Gestion matérielle du contrôle

Cette section présente notre proposition d'architecture de CGRA et le modèle d'exécution associé capables de gérer le flot de contrôle en interne. Ces travaux font l'objet d'un dépôt de brevet [Peyret et al., 2014d].

5.2.1 Modèle d'exécution et architecture résultante

Dans cette sous-section, nous décrivons en détail comment le modèle d'exécution et l'architecture que nous proposons permettent de résoudre les problèmes de gestion du contrôle soulevés dans le chapitre 2 (section 2.6), c'est-à-dire le changement de BB et la gestion des instructions mémoire bloquantes. Plusieurs variantes des mécanismes peuvent exister. Elles seront alors détaillées et mises en perspective en fonction du contexte d'utilisation. L'architecture (représentée schématiquement en figure 5.1) se base sur un ensemble de tuiles de calcul comprenant :

- une mémoire programme (distribuée ou non dans chaque tuile) ;
- l'utilisation de mot de commande possédant des champs spécifiques pour le contrôle ;
- une unité de gestion du flot de contrôle dans chaque tuile permettant de contrôler l'exécution de l'instruction courante. Cette unité comprend un compteur de cycle, un pointeur d'instruction, une table de localisation de la première instruction de chaque BB dans la mémoire et plusieurs moyens de « garder » une instruction (c'est-à-dire d'effectuer un « NOP » à la place et ne pas incrémenter le pointeur d'instruction) ;
- un bus partagé entre les tuiles dédié à la gestion du contrôle ;
- une partie opérative classique : ALU + RF locale.

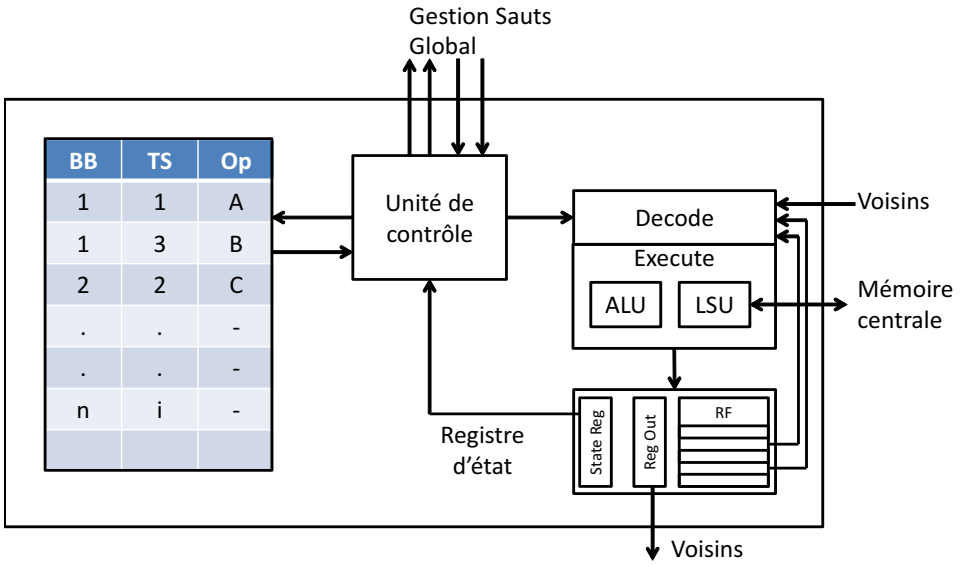


FIGURE 5.1 – Schéma d'une tuile gérant les sauts mais pas les *loads* bloquants. L'étage de decode reçoit ces données de la RF et/ou des voisins atteignables par le réseau d'interconnexion. Le registre de sortie permet de communiquer aux voisins une valeur.

À ces éléments nécessaires peut s'ajouter, pour une meilleure efficacité, un ensemble de mécanismes permettant de gérer des attentes bloquantes lors d'accès à la mémoire centrale par les tuiles. Ils seront détaillés dans la section 5.2.1.2. Cette sous-section suivante présente la gestion des changements de BB, puis détaille comment sont gérés les *loads*.

5.2.1.1 Gestion des changements de Basic Block

Dans le cas classique, le fonctionnement d'une unité de contrôle est le suivant :
- le pointeur d'instruction pointe vers l'instruction (ou l'ensemble des instructions dans le cas d'une mémoire centralisée) qui sera exécutée à ce cycle ;
- l'instruction est décodée pour piloter les différents éléments à contrôler ;
- le pointeur d'instruction est incrémenté de 1. Si sa valeur est supérieure à la dernière adresse de la mémoire, il passera à la première valeur (permettant de boucler l'exécution d'un programme).

Ce cas présente l'inconvénient de devoir explicitement écrire en mémoire le fait qu'une tuile ne fasse pas d'opération au cycle courant (le NOP), ce qui augmente grandement la taille de la mémoire programme. Pour remédier à ce problème, le concept de garde a été développé [Dijkstra, 1975]. Une garde permet de n'exécuter une instruction que si un prédicat est vérifié. Ce mécanisme est classiquement utilisé dans les processeurs spécialisés et dans certains processeurs généralistes (une bibliographie assez complète sur le sujet peut être trouvée dans [Pnevmatikatos, 1996]). Pour diminuer la taille de la mémoire programme, dans le mot mémoire contenant l'instruction, un champ contenant le cycle auquel l'instruction est censé être exécutée est ajouté. L'unité de contrôle doit alors posséder en plus un compteur qui permet de connaître le cycle courant. Le fonctionnement de l'unité de contrôle devient alors le suivant :
- le pointeur d'instruction pointe vers une instruction possédant un champ « cycle d'exécution » ;
- la valeur de ce champ est comparée au compteur de cycle de l'unité de contrôle ;
- si les valeurs sont identiques, alors l'instruction est effectivement exécutée, le pointeur d'instruction et le compteur sont incrémentés ;
- sinon, l'instruction n'est pas exécutée, le pointeur d'instruction ne change pas et le comp-

teur de cycle est incrémenté ;
— si ensuite la valeur du pointeur d'instruction est supérieure à la taille de la mémoire alors elle est remise à la première valeur (et pointe de ce fait vers la première instruction) ;
— si le compteur de cycle est plus grand que le nombre de cycles d'exécution de l'application, il est réinitialisé.

Sur ce principe, nous définissons le modèle d'exécution, et l'architecture d'une tuile de CGRA, capable de gérer des CDFGs. Le schéma de cette première version de tuile est donné en figure 5.1. Chaque tuile possède une mémoire programme locale. Cette mémoire est pilotée par un pointeur d'instruction qui permet d'avoir le mot mémoire courant et qui est situé dans l'unité de contrôle de la tuile. Les mots mémoires sont composés de trois champs :

1. un numéro de BB (un ID) ;
2. un cycle d'exécution ou *Timestamp* (TS sur les figures) ;
3. et l'instruction à proprement parler, qui pilote l'opérateur, les registres, etc.

L'unité de contrôle pilote le décodeur en lui fournissant l'instruction à exécuter. En plus du réseau d'interconnexion de données classique reliant les tuiles entre elles, un bus de gestion des sauts est ajouté. Toutes les unités de contrôles des tuiles sont reliées à ce bus. Ce bus possède une partie dédiée à l'ID du BB et un signal signifiant que l'ID du nouveau BB est sur le bus. En plus du compteur de cycle et du pointeur d'instruction, l'unité de contrôle possède une table permettant de connaître la position dans la mémoire de la première instruction de chaque BB ainsi qu'un registre permettant de connaître l'ID du BB courant.

Le fonctionnement pour une instruction classique est alors le suivant :
— le pointeur d'instruction pointe vers un mot mémoire ;
— si l'ID du BB contenu dans ce mot n'est pas le même que celui stocké dans l'unité de contrôle, alors l'instruction n'est pas exécuté et ce sera un NOP qui sera envoyé à la partie opérative ;
— sinon, si l'instant d'exécution est le même que la valeur du compteur de cycle, alors l'instruction est envoyé au décodeur, sinon, c'est là encore un NOP qui est envoyé ;
— dans le cas où l'instruction a été exécutée, le pointeur d'instruction est incrémenté ;
— le compteur de cycle est lui aussi incrémenté ;

Avec le contenu de la mémoire programme de la figure 5.1, au premier cycle l'instruction A sera exécutée. Puis au second cycle ce sera un NOP (TS ne correspond pas au cycle courant). Au troisième cycle, ce sera B et pour tous les cycles suivants jusqu'au changement de BB, des NOPs qui seront exécutés.

Le fonctionnement pour une instruction de changement de BB est le suivant :
— le pointeur d'instruction d'une des tuiles pointe vers l'instruction de changement ;
— quand le cycle correspond à celui du mot mémoire, cette instruction est exécutée ;
— si c'est une instruction conditionnelle, l'unité de contrôle va lire la valeur du registre d'état de le tuile pour savoir quel est le prochain BB à être exécuté (par exemple si la condition d'arrêt d'une boucle est vérifiée).
— si c'est une instruction inconditionnelle, l'ID du prochain BB est directement connu ;
— l'unité de contrôle de la tuile qui a l'instruction de changement de BB connaît donc l'ID du prochain BB. Elle va alors écrire sur le bus de gestion des sauts la valeur de cet ID et signaler qu'il y a un nouvel ID de BB sur le bus (il ne peut pas y avoir de conflit d'écriture sur ce bus car il n'y a qu'une seule instruction de saut par BB) ;
— toutes les tuiles étant reliées à ce bus, toutes vont enregistrer la nouvelle valeur de l'ID, remettre le compteur de cycles à zéro et la valeur du pointeur d'instruction sera initialisée grâce à la table des premières instructions de BB. Lorsque le signal de nouvel ID revient à son état initial, l'intégralité des tuiles va recommencer à incrémenter le compteur de cycles. Ainsi, toutes les tuiles auront le même cycle au même instant (reprise de fonctionnement synchrone).

Dans l'exemple de la figure, après avoir changé pour le BB 2, la tuile exécutera un NOP, puis C.

Dans le modèle CDFG, il n'y a qu'une seule instruction de contrôle par BB en dernière position. Dans notre modèle, cette instruction est écrite en mémoire programme et possède un ID de BB et un instant d'exécution. Dans le cas d'une instruction de saut inconditionnel, l'ID du BB suivant est donné en tant qu'« immédiat » unique. Dans le cas d'un saut conditionnel, il faut avoir dans la mémoire programme les deux IDs des BBs de destination. Deux mises en œuvre distinctes doivent être considérées :

— la première définit un saut conditionnel comme étant l'équivalent de deux instructions de saut inconditionnel gardées ;
— la seconde définit une instruction particulière pour le saut conditionnel (celle utilisée dans le fonctionnement précédent).

La première mise en œuvre présente l'avantage de ne pas avoir à ajouter une nouvelle instruction de saut. Elle utilise le mécanisme de garde sur la valeur d'un registre ou sur la valeur du registre d'état (en fonction des possibilités matérielles). Cependant, elle présente l'inconvénient soit d'être sur deux cycles d'exécution (le cas si le test est positif, puis au cycle suivant le cas si le test est négatif), soit d'utiliser deux tuiles (le cas du test positif sur une tuile, l'autre cas sur une autre tuile). La seconde possibilité de mise œuvre présente l'avantage de s'exécuter en un seul cycle. Elle doit alors avoir les IDs des deux BBs en immédiat. La section 5.2.2 donnera les conséquences de l'une et l'autre des implémentations.

5.2.1.2 Accès à la mémoire

Le chapitre 2, section 2.6 a illustré deux problématiques pour les accès à la mémoire centrale. La première est due au fait que le composant mémoire ne donne pas le résultat d'une lecture dans le cycle où la donnée est demandée mais après un nombre de cycles incompressible. La seconde provient du rafraîchissement de la mémoire qui bloque toutes les transactions le temps qu'il se termine et donc retarde l'arrivée de la donnée.

Cependant, la gestion des accès à la mémoire ne se limite pas simplement à la résolution de ces deux problématiques. La mise en œuvre et la manière dont les données sont transférées nécessitent aussi d'être traitées et seront détaillées dans un troisième paragraphe.

Gestion des instructions mémoire bloquantes

La gestion des instructions mémoire bloquantes (en particulier le *load* bloquant) peut se faire de deux façons différentes en fonction des besoins utilisateurs. La première façon consiste à ajouter un bus commun à toutes les tuiles permettant de signaler un arrêt global du fonctionnement : bus de « *freeze* » global. La figure 5.2 présente le schéma d'une tuile possédant ce mécanisme en plus de celui permettant de gérer les sauts. Le fonctionnement de la tuile est alors le suivant :

— à la fin d'une instruction d'accès à la mémoire, si la *Load-Store Unit* (LSU) signale qu'elle n'a pas terminé la demande (la donnée n'est pas encore disponible sur le bus ou la donnée n'est pas encore enregistrée si cela est vérifié), alors elle informe l'unité de contrôle ;
— l'unité de contrôle va alors mettre le signal de « *freeze* » global dans son état actif ;
— tant que la LSU n'aura pas donné son feu vert, l'ensemble des tuiles du CGRA exécutera l'instruction NOP (à cause du signal de « *freeze* ») ;
— quand la LSU aura terminé son opération, le signal de « *freeze* » repassera dans son état inactif et l'exécution reprendra son cours normal.

Cette façon de faire ne nécessite pas spécialement de modification pour le reste de la tuile en particulier au niveau des RFs et de la partie opérative (ALU).

La seconde variante de gestion des instructions mémoires bloquantes, illustré schématiquement dans la figure 5.3, permet d'éviter de stopper l'intégralité de l'exécution en cas de besoin, mais seulement certaines parties du composant. En effet, dans le cas d'un *load*, la donnée qui est chargée n'est utilisée que par une partie de la suite des instructions du BB. Il peut être intéressant de ne bloquer que cette partie de l'exécution. L'idée est la suivante : pour chacun des voisins, on a un registre dédié (dans une architecture CGRA classique, il n'y a qu'un seul

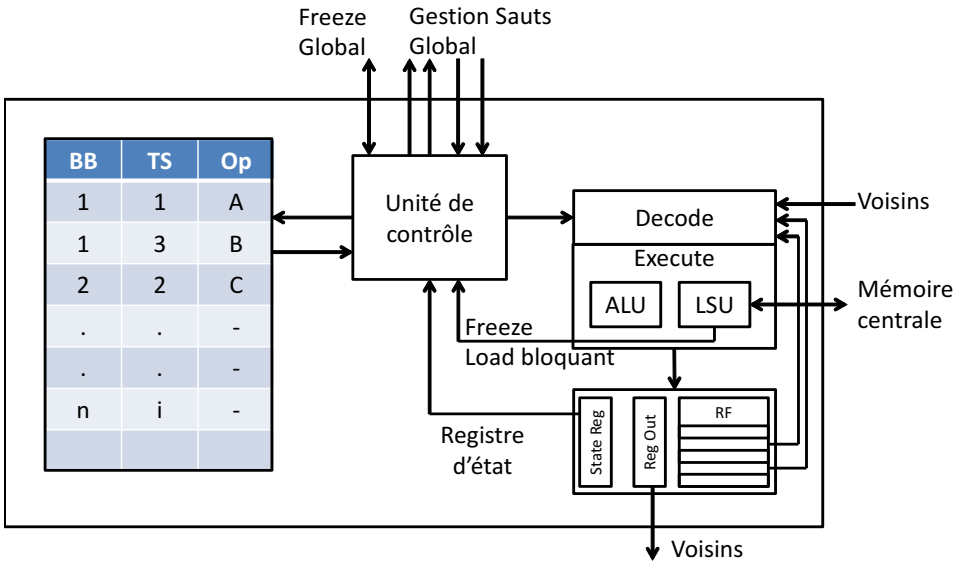


Figure 5.2 – Schéma d'une tuile gérant les sauts et les instructions mémoires bloquantes par « *freeze* » global

registre pour tous les voisins, le « registre de sortie »). Ces registres possèdent un champ supplémentaire permettant d'une part de signaler qu'il y a une valeur de disponible et d'autre part d'enregistrer si la valeur a été lue (consommée) par le voisin destinataire. Il s'agit en quelque sorte d'un *flag* d'« accusé de réception ». Au niveau du réseau d'interconnexion, ceci impose que les voisins soient capables de signaler la lecture de la donnée (au travers par exemple d'une remise à zéro du bit signalant qu'une donnée est disponible). C'est en contrôlant l'état de ces accusés de réception que l'exécution sera bloquée ou bien reprise. Aussi, comme l'exécution sera partiellement bloquée, il faut un signal global à toutes les tuiles qui permet de bloquer le changement de BB. En effet, il ne faut pas autoriser un saut si il reste des instructions du BB courant à exécuter dans n'importe quelle tuile de l'architecture. Pour cela, un signal « *Ready ChangeBB* » est ajouté à l'interface de gestion des sauts. Son fonctionnement est le suivant : si aucune tuile ne le force, il est à l'état OK, signalant qu'il est possible de changer de BB. Si une tuile (ou plus) n'est pas prête, alors il est forcé à l'état inverse. C'est l'unité de contrôle de chacune des tuiles qui lit et pilote ce signal. Chaque tuile peut donc prendre la main dessus. Dans le cas où le signal n'est pas dans son état OK, l'exécution d'un changement de BB est « gardée ». Les unités de contrôle des tuiles forcent le signal dans son état inverse (« *Not OK* ») tant qu'il reste des instructions du BB courant dans la mémoire. Ceci peut être fait simplement en vérifiant si l'ID du BB de l'instruction pointée par le pointeur d'instruction est celle du BB courant. En effet, lorsque la dernière instruction (pour une tuile donnée) du BB courant est exécutée, le pointeur d'instruction est incrémenté et il pointe donc vers une instruction d'un autre BB.

Le fonctionnement global devient alors le suivant :

— dans le cas normal, les résultats pour les voisins sont écrits dans les registres correspondants (il faut simplement que le mot mémoire contienne les destinations des données). Les *flags* sont changés pour signaler qu'une donnée est disponible uniquement dans les registres qui devront être lus le cycle suivant. Pour les autres registres de sortie, le *flag* est laissé dans son état « donnée non présente » ;

— si une tuile est bloquée (par un *load* par exemple) alors elle laisse les *flags* de ses registres de sortie dans l'état « donnée non présente », empêche l'incrémentation du compteur de cycle et du pointeur d'instruction et ne change pas les *flags* des données qu'elle a lues. Elle restera bloquée (à effectuer des NOPs) tant que la donnée ne sera pas disponible (ou enregistrée si elle est bloquée par un *store*). Lorsque celle-ci deviendra disponible elle

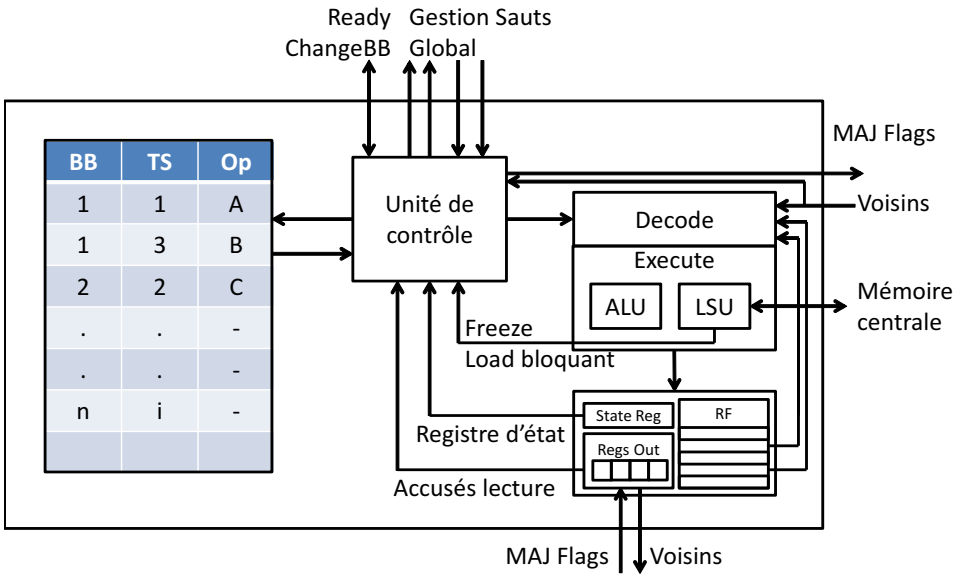


FIGURE 5.3 – Schéma d'une tuile gérant les sauts et les accès mémoire bloquants de manière locale

sera écrite dans les registres nécessaires (selon le besoin) et les *flags* de ces derniers seront mis à jour, signalant une donnée disponible. L'unité de contrôle signalera alors qu'elle a consommé les données d'entrée éventuelles ;

— lorsqu'une tuile utilise au moins une donnée provenant d'une autre tuile, elle vérifie que le *flag* signale bien qu'une donnée est disponible. Si c'est le cas pour toutes les données qu'elle doit lire, alors elle va remettre ce (ou ces) *flag(s)* à son (leur) état initial pour signaler à la tuile productrice que la (les) donnée(s) a (ont) bien été lue(s). Si ce n'est pas le cas, aucun *flag* n'est changé et la tuile se bloque (c'est-à-dire exécute un NOP sans incrémenter le compteur de cycles et le pointeur d'instruction) ;

— en début de cycle, l'unité de contrôle vérifie que l'intégralité des données qui devait être consommées l'a été. Si c'est le cas, la nouvelle instruction est exécutée si elle le peut comme définit précédemment. Si au moins un des voisins n'a pas consommé sa donnée, alors la tuile se bloque ;

— lorsque c'est une instruction de saut qui doit être exécutée, elle ne peut l'être que si le signal « *Ready ChangeBB* » est à l'état OK.

Ce fonctionnement ne risque pas de se bloquer avec toutes les tuiles s'attendant mutuellement (un interblocage ou *deadlock*) car l'ordonnancement est fixé à l'avance. Dans le pire des cas, l'ensemble des tuiles peut être bloqué par une seul lecture mémoire bloquante (comme avec le « *freeze* global »), mais quand celle-ci se terminera, le fonctionnement reprendra son cours.

Cette seconde solution peut être améliorée pour limiter les blocages en remplaçant les registres de sorties par des mémoires de type *First In First Out* (FIFOs) à plusieurs places (le cas présenté en détail précédemment est l'équivalent d'une FIFO à une seule entrée). Dans ce cas, une tuile dont les résultats n'ont pas été consommés peut ne pas être bloquée tant qu'il reste de la place dans ces FIFOs. L'introduction de ces FIFOs ne change pas les autres cas de blocage.

De manière à autoriser plusieurs codes à s'exécuter sur un même composant CGRA (par exemple sur un CGRA de 4×4 tuiles, faire s'exécuter un code sur les deux premières lignes et un second sur les deux dernières), un cloisonnement des signaux de contrôle peut être ajouté de manière à définir des zones hermétiques vis-à-vis du contrôle. Ainsi, une instruction bloquante sur un code ne bloquera pas l'autre.

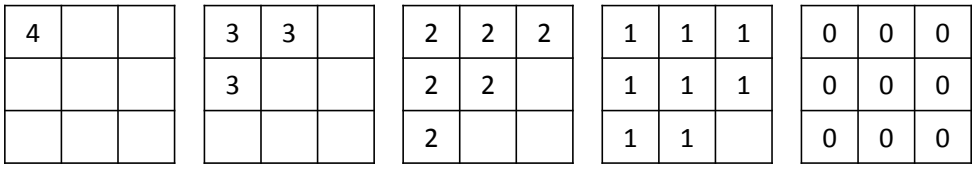


Figure 5.4 – Exemple de propagation de nombre de cycles restants avant changement de Basic Block sur un CGRA 3×3 possédant une interconnexion de type mesh 2D simple.

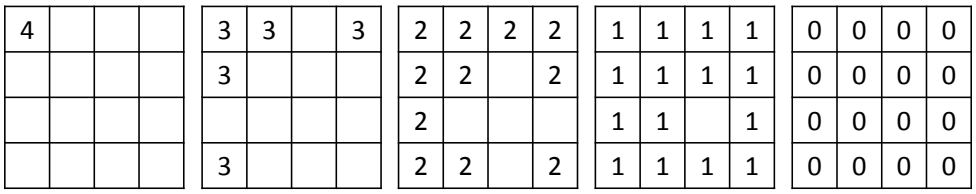


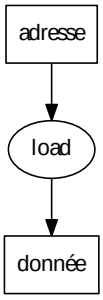
Figure 5.5 – Exemple de propagation de nombre de cycles restants avant changement de Basic Block sur un CGRA 4×4 possédant une interconnexion de type mesh 2D torique.

Il est aussi possible d'imaginer le cas où la gestion des sauts n'utilise pas un bus global mais une transmission de message de proche en proche. Un tel mécanisme permet de diminuer le chemin critique au niveau de l'unité de contrôle et donc d'augmenter la fréquence de fonctionnement. Cependant, elle nécessite aussi un plus grand nombre de cycles lors d'un changement de BB de manière à synchroniser les démarrages d'exécution des différentes tuiles. Une façon simple d'effectuer cette synchronisation est de propager en même temps que le nouvel ID de BB une valeur définissant le nombre de cycles restants avant de débuter l'exécution des instructions du nouveau BB. Cette valeur est décrémentée à chaque fois que l'information traverse une tuile et vaut initialement la plus grande distance entre deux tuiles. Ainsi, lorsque cette valeur passe à 0, la tuile la plus éloignée aura reçu le nouvel ID de BB et pourra donc bien démarrer de façon synchrone. La figure 5.4 présente un exemple de propagation dans le cas d'un CGRA 3×3 avec une communication restreinte aux plus proches voisins (haut, bas, gauche et droite). Dans cet exemple, c'est la tuile du haut à gauche qui initie le changement de BB. Elle a une distance de Manhattan de 4 avec la tuile du bas à droite et il est donc nécessaire de propager pendant 4 cycles le changement de BB. Au cycle suivant celui où toutes les tuiles sont à 0, la première instruction du nouveau BB sera exécutée. Dans le cas de réseau d'interconnexion plus complexe, la détermination de cette distance peut être plus difficile. Une illustration pour un CGRA 4×4 possédant un réseau mesh 2D torique est donnée en figure 5.5.

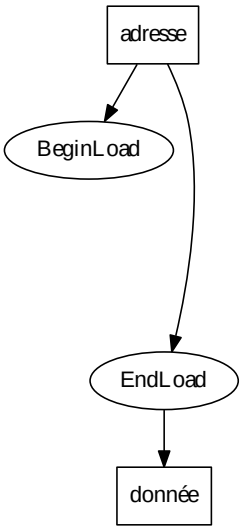
Gestion des accès mémoires avec une latence incompressible

Gérer les accès mémoires avec un latence incompressible est indispensable pour être capable d'exécuter un code efficacement. Il n'est en effet pas concevable qu'à chaque *load*, l'architecture soit bloquée pendant 8 cycles par exemple. Il faut être capable d'effectuer d'autres opérations (si possible) pendant ce temps. Matériellement, il est possible d'envisager une exécution « *out of order* » comme présentée dans [Parashar et al., 2013], mais celle-ci implique une approche complètement différente en termes de placement des opérations sur les opérateurs que celle proposée dans ce manuscrit.

Une approche permettant de gérer ces accès mémoires et possédant un impact très faible sur le flot est la suivante : les nœuds d'accès à la mémoire sont transformés en deux nœuds qui effectuent chacun une partie de l'opération. Le premier nœud effectue la requête auprès de la mémoire. Le second nœud vérifie que la requête a bien été effectuée. C'est seulement le second nœud qui devient bloquant. Si au moment où il s'exécute, l'opération mémoire n'est pas terminée, alors on se retrouve dans le cas d'une instruction mémoire bloquante classique. La figure 5.6 donne un exemple de transformation pour un *load*. On remarquera que l'adresse de



(a) Le nœud *load* original.



(b) Le résultat de la transformation.

FIGURE 5.6 – Transformation d'un *load* en deux nœuds : un BeginLoad et un EndLoad.

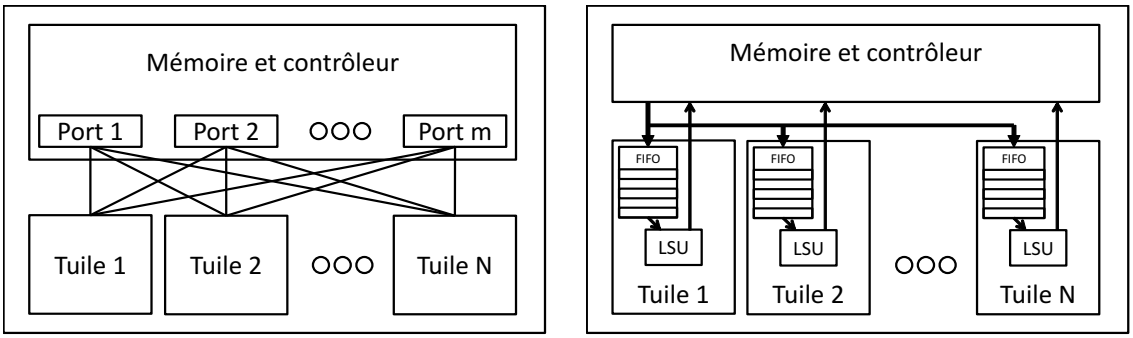
lecture du *load* est donnée en paramètre à la fois au premier nœud (le BeginLoad) et au second nœud (le EndLoad). La raison de cette redondance est double : il s'agit d'une part de pouvoir faciliter le contrôle du succès de l'opération auprès de l'interface mémoire car cette opération est la seule à ce moment de l'exécution à pouvoir/devoir accéder à cette adresse de la mémoire centrale ; d'autre part, cela permet de ne pas avoir à fixer le placement des deux nœuds sur la même tuile. En effet, si le EndLoad n'avait pas l'adresse de l'échange avec la mémoire, il ne pourrait pas savoir quelle valeur récupérer (en cas de plusieurs *loads* simultanées). L'adresse du EndLoad pourra être changée, le cas échéant, en un numéro de port de la mémoire dans lequel il faudra lire la donnée.

Par rapport au flot décrit précédemment, cette gestion ajoute une passe transformant les nœuds d'accès à la mémoire en des nœuds doubles avant la projection. Ces deux nœuds doivent être séparés lors de l'ordonnancement d'un nombre de cycles fixé dépendant de la technologie mémoire. Cet aspect modifie la priorité de ces nœuds d'accès à la mémoire. En effet, les nœuds de « Begin » doivent être prioritaires au cycle pendant lequel ils doivent être ordonnancés pour respecter cet écart et minimiser l'impact sur la latence globale. Par rapport au CGRA, cette modification impose simplement d'ajouter les instructions de Begin et de End au jeu d'instructions et de n'implémenter le blocage que pour les instructions End.

Méthode de transfert des données

Jusqu'à présent, dans le flot de projection de l'application sur l'architecture, nous avons implicitement fait l'hypothèse que toutes les tuiles pouvaient, au travers de leur *Load-Store Unit* (LSU) respective accéder à la mémoire centrale. Dans le flot, nous avons prévu de limiter le nombre d'accès mémoire parallèle de manière à être plus réaliste (cf. section 3.2.2.1).

Dans le cas d'un petit CGRA, il peut être envisagé que l'interface avec la mémoire se fasse au travers d'un petit nombre de ports indépendants (par exemple 2 ou 4). Chaque tuile est reliée aux différents ports, comme illustré en figure 5.7a. Lors d'un BeginLoad, l'adresse de la donnée est écrite dans un port. Puis le EndLoad lit dans le bon port la valeur demandée. Un fonctionnement de ce type impose de déterminer un numéro de port, de l'ajouter au BeginLoad et de remplacer l'adresse du EndLoad par ce numéro de port. Ce fonctionnement à base de ports permet une transformation qui peut s'avérer très avantageuse : la multiplication du EndLoad. Jusqu'alors, avec des nœuds *loads* en une seule étape, si la donnée était utilisée par beaucoup



(a) Accès à la mémoire par ports (b) Accès à la mémoire avec réception FIFO

Figure 5.7 – Deux variantes d'accès à la mémoire des tuiles

d'opérations différentes, le flot effectuait la transformation de routage suivi de la scission du nœud de mémorisation. En effet, il est en général préjudiciable de multiplier le nombre de *loads* dans le graphe car ils augmentent la pression sur la mémoire centrale. Avec des *loads* en deux parties, ce n'est pas le même problème. Deux tuiles différentes peuvent lire en même temps la valeur située dans un même port de sortie du contrôleur mémoire. Ainsi, il est possible de relâcher la pression sur le réseau d'interconnexion pour les variables provenant de la mémoire centrale sans pour autant augmenter le nombre de cycle du BB correspondant.

Cependant, malgré cette limitation, il n'est pas raisonnable d'imaginer que toutes les tuiles puissent communiquer directement avec la mémoire dans le cas de CGRAs de grande taille en raison du *fan-out* résultant sur les registres des ports. Une première solution peut être de rajouter des intermédiaires entre la mémoire et les tuiles de calcul. Ainsi, le nombre de tuiles connectées à chacun des intermédiaires sera réduit. Cependant une telle solution rajoute nécessairement des registres et donc des cycles d'attente incompressibles. Pour pallier cet inconvénient, il est envisageable de faire fonctionner la mémoire et les intermédiaires plus vite que le reste de l'architecture comme dans [Rakossy et al., 2014, Rakossy et al., 2012] où les différents « niveaux » de l'architecture (calcul, communication et mémoire) peuvent fonctionner à des fréquences différentes. À titre d'exemple, le ratio « (1 : 8 : 4) » fait s'exécuter la partie opérative à une fréquence f_c, le niveau intercommunication à $8 \times f_c$ et les accès mémoire à $4 \times f_c$, favorisant le déplacement des données dans l'architecture et permettant plusieurs accès mémoire par cycle de calcul tout en limitant la consommation énergétique.

Une seconde manière de pallier cette limitation, compatible avec la précédente, est de tirer partie de l'utilisation des accès à la mémoire par BeginLoad/EndLoad comme dans notre brevet [Peyret et al., 2014d] et illustré en figure 5.7b. La raison d'être des nœuds *loads* en deux parties est de pouvoir décorréler la demande d'une donnée en mémoire de son utilisation. Comme nous l'avons vu, il est intéressant de pouvoir faire en sorte que ce soit une autre tuile qui initie le *load*. Ceci permet à la tuile qui en a besoin d'effectuer une action de moins. Toujours dans l'idée de simplifier le travail de la tuile qui a besoin d'une donnée mémoire, nous proposons que ce soit le contrôleur mémoire qui donne la valeur directement à la tuile plutôt que ce ne soit la tuile qui aille chercher la donnée auprès du contrôleur. Dans ce mode de fonctionnement, les tuiles doivent avoir dans leur LSU une zone permettant de stocker au moins une valeur. Pouvoir réceptionner plusieurs données permet de pallier l'aspect indéterminé de la durée que mettra la mémoire pour retourner des données. Par exemple, supposons que la tuile ait besoin de deux données mémoire à deux cycles successifs. Un certain nombre de cycles avant la première utilisation (par exemple 10 cycles), un BeginLoad sera exécuté. Puis au cycle suivant, le second BeginLoad sera exécuté. Si la mémoire est disponible et bien positionnée, elle répondra aux requêtes en seulement 8 cycles. Le contrôleur mémoire va donc écrire dans la LSU le résultat du premier accès mémoire deux cycles avant qu'il ne soit utilisé. Puis le contrôleur va écrire le

résultat du second accès mémoire. Au cycle suivant, la tuile va consommer la première valeur, puis encore au cycle suivant, elle va consommer la seconde. Une FIFO remplirait parfaitement le rôle tampon. Il n'y a par ailleurs pas forcément besoin que cette FIFO soit de grande taille : une première analyse des *mappings* de codes ciblés montre qu'il y a rarement plus de quatre accès mémoire successifs sur une même tuile.

Dans ce cas de figure, il est encore possible de multiplier les EndLoads, mais comme c'est le contrôleur mémoire qui écrit dans les FIFOs des tuiles destinataires, il faut lui communiquer les IDs de ces tuiles. Cette communication implique que l'instruction qui sera écrite en mémoire programme puisse contenir un nombre *a priori* quelconque d'IDs de tuiles. Cette supposition n'est pas réaliste car, quelle que soit la réalisation physique, les instructions mémoire auront une taille maximale. Pour faire fonctionner ce mécanisme, il faut donc que pendant la compilation, les groupes de tuiles soient identifiés, numérotés et qu'ils soient chargés dans le contrôleur mémoire. Ainsi, quand le contrôleur mémoire recevra un BeginLoad avec une adresse mémoire et un identifiant pour le groupe de tuiles destinataires, il saura dans quelles tuiles écrire. Dans un contexte de tolérance aux fautes, cette solution à base de FIFOs présente un avantage non négligeable car elle diminue la vulnérabilité du contrôleur mémoire. Dans la première solution utilisant des ports, si l'un d'eux est victime d'une faute, alors toutes les tuiles utilisant ce port auront potentiellement des valeurs erronées. C'est donc la solution utilisant des FIFOs qui est implémentée et testée dans la section suivante.

Les deux solutions précédentes peuvent encore être améliorées en changeant la façon dont est considérée la mémoire centrale et son contrôleur. Il n'est en effet pas forcément très judicieux que les BeginLoads ne dépendant pas d'une valeur calculée (adresse constante/statique) soient placés sur des tuiles de calcul. Pour remédier à cela, nous proposons de déporter ces BeginLoads sur une des tuiles dédiées aux initialisations d'accès mémoire. Cette solution s'intègre parfaitement dans le flot en signalant simplement que l'opérateur de ces tuiles dédié n'est compatible qu'avec des BeginLoads. Si l'adresse du *load* n'est pas connue, ce sera aux tuiles de calcul de fournir l'adresse du *load* et donc d'exécuter le BeginLoad. Cette amélioration n'est véritablement envisageable que dans un contexte sans tolérance aux fautes, car spécialiser des tuiles diminue les possibilités de reconfiguration.

5.2.2 Implémentation

L'architecture de CGRA et le modèle d'exécution associé ont été implémentés en VHDL pour permettre de valider le fonctionnement par simulation. Cette section présente les différents choix de dimensionnement et d'implémentation qui ont été faits pour cette vérification.

Jeu d'instructions

De manière à pouvoir réaliser un prototype capable d'exécuter les codes applicatifs utilisés pour l'évaluation du flot, l'architecture a été pourvue d'opérateurs permettant d'exécuter les instructions classiques suivantes :

- addition, soustraction, multiplication signées et non signées ;
- décalage de bits à gauche et à droite pour des variables signées et non signées ;
- une opération MOV : « sortie = entrée » (un NOP ou une mémorisation) ;
- des opérations de lecture en mémoire en deux temps : *BeginLoad* et *EndLoad* ;
- une opération d'écriture en mémoire : *store* ;
- une opération de saut inconditionnel ;
- une opération de saut conditionnel ;
- l'opération *PHI* ;
- deux opération de test : égalité et supériorité ;
- diverses opérations logiques bits à bits classiques : « et », « ou », « ou exclusif », « non », « non et », « non ou exclusif ».

Format des instructions

Nous avons défini quatre formats pour représenter les instructions en mémoire programme :

— Le premier format, le plus courant, correspond à l'utilisation d'une opération prenant comme opérandes les valeurs contenues uniquement dans les registres de l'architecture. Ces registres peuvent provenir de la RF de la tuile ou des tuiles voisines.
— Le deuxième format correspond à l'utilisation d'une opération utilisant un opérande provenant d'un registre et une constante contenue directement dans l'instruction.
— Le troisième format permet de mettre en registre une constante plus grande et donc ne permet pas d'effectuer une opération simultanée.
— Le quatrième format permet d'avoir une opération de vote à trois entrées (nécessaire pour certaines mises en œuvre de la tolérance aux fautes).

Les quatre formats d'instruction commencent tous par les mêmes deux premiers champs comme définit précédemment : l'ID du BB et le cycle d'exécution TS. L'ID du BB codée sur 5 bits, ce qui permet de représenter jusqu'à 32 BBs, ce qui est suffisant pour une grande partie des applications (si ce n'est pas le cas pour un code, il est possible de changer les options de compilation pour augmenter la taille des BBs et diminuer leur nombre). Le TS codée sur 8 bits, soit 256 cycles disponibles, là encore cela est suffisant pour une grande partie des codes. Le champ suivant est celui déterminant le format, il est codé sur 2 bits. Ensuite, les champs diffèrent en fonction du format. Pour le premier format, l'instruction est composée des éléments suivants :

— le code de l'opération, sur 5 bits ;
— un champ, sur 2 bits, permettant éventuellement de « garder » l'opération en fonction du registre d'état (résultat précédent égal à 0 ou positif) ;
— un code définissant s'il faut écrire le résultat dans le registre de sortie, sur 1 bit ;
— un identifiant sur 6 bits pour le registre de la RF dans lequel le résultat doit être enregistré (le registre 0 est un registre fictif qui permet de ne pas enregistrer le résultat dans la RF) ;
— un champ sur 2 bits permettant de connaître la provenance du premier opérande (voisin, RF locale ou une éventuelle RF partagée) ;
— un identifiant sur 6 bits pour le premier opérande (cela correspond à Nord, Sud, Est, Ouest dans le cas à 4 voisins ou au numéro du registre dans la RF) ;
— les deux mêmes champs pour le second opérande.

Pour le deuxième format, la seule différence réside dans les deux champs du second opérande qui sont fusionnés pour avoir un immédiat de 8 bits.

Pour le troisième format, ce sont les quatre champs réservés aux opérandes qui sont fusionnés pour avoir un immédiat de 16 bits. Seule l'instruction MOV peut être exécuté dans ce format. Ce troisième format a été introduit pour faciliter le travail avec des constantes plus grandes que simplement celles représentées sur 8 bits car le reste de l'architecture est dimensionné pour des données représentées sur 32 bits.

Le quatrième correspond uniquement à un vote. Les bits d'opération et de garde sont récupérés pour permettre de définir une troisième provenance et ainsi effectuer le vote majoritaire.

Tableau 5.1 – Les quatres formats d'instruction.

2 bits	5 bits	8 bits	5 bits	2 bits	2 + 6 bits	2 bits	6 bits	2 bits	6 bits
Format1	BB	TS	OpCode	Garde	RegDest	TypeOpA	AdrOpA	TypeOpB	AdrOpB
Format2								Immédiat court	
Format3						Immédiat long			
Format4			AdrOpC			TypeOpA	AdrOpA	TypeOpB	AdrOpB

5.2.3 Limitations et perspectives

De nombreuses variantes des mécanismes pour gérer à la fois les accès mémoire et les changements de BB ont été présentées dans cette section. À chaque fois, une seule variante a été choisie pour être implémentée afin de répondre à l'objectif de prise en compte des fautes transitoires ainsi qu'à la détection, puis la correction de fautes permanentes. L'évaluation des autres variantes permettra de connaître les véritables performances de notre architecture dans un contexte sans tolérance aux fautes. Nous avons aussi identifié un certain nombre de points qui peuvent être vus comme des verrous à l'utilisation de notre architecture. Tout d'abord le chargement du *bitstream*, c'est-à-dire de la mémoire programme dans notre cas, n'a pas été abordé. La difficulté consiste à écrire dans un nombre de mémoires programmes égal au nombre de tuiles de l'architecture car elles sont distribuées. Aussi, l'architecture que nous avons décrite ne permet pas en l'état actuel de faire s'exécuter un système d'exploitation. En effet, dans le jeu d'instructions que nous avons implémenté, il n'y a pas de notion d'appel de fonction et il semble bien peu raisonnable de réécrire l'intégralité des codes applicatifs et du système de manière à former un seul et même CDFG.

Bien que le temps imparti de la thèse n'ait pas permis la description d'une architecture complètement mature, une première version a été décrite en *Very high speed integrated circuit High Definition Language* (VHDL) et synthétisée. Ceci a permis de vérifier par simulation le fonctionnement et d'obtenir une première évaluation des performances. La technologie de synthèse utilisée est la TSMC40 avec la chaine d'outils Synopsys©. La fréquence cible était 200 MHz et notre *design* nous a permis de respecter cette fréquence. La surface d'une tuile obtenue est d'environ 60 000 μm^2 avec la mémoire d'instruction dont une taille est d'environ 15 000μm^2. Ainsi la surface totale d'un CGRA 8×8 est d'environ $3mm^2$. La répartition de la surface est d'environ moitié pour l'unité de contrôle avec la mémoire programme et l'autre moitié pour la partie calculatoire, les registres et l'interconnexion. Ces résultats sont des résultats préliminaires qui illustrent simplement la faisabilité d'une telle architecture. Il faudrait passer beaucoup plus temps sur l'écriture du VHDL pour optimiser les chemins critiques et améliorer le placement des éléments pour véritablement connaître les performances de l'architecture CGRA proposée.

5.3 Mise en œuvre de la tolérance aux fautes transitoires

Nous avons précédemment décrit une architecture CGRA capable d'exécuter un code complet. Dans le cadre de la tolérance aux fautes, il faut que cette architecture soit capable de garantir que ses résultats sont correctes. Cette section analyse les différentes mise en œuvre des méthodes permettant de corriger les fautes transitoires. Elle commence par l'étude des moyens disponibles pour les registres de la tuile puis détaillera ceux permettant de fiabiliser les calculs. Étant donné la dépendance entre la mise en œuvre de tolérance aux fautes transitoires et celle de détection des fautes permanentes, cette section ne conclura pas sur l'utilisation d'une des mises en œuvre présentée, mais donnera leurs avantages et inconvénients respectifs.

5.3.1 Tolérance pour les registres

Comme présenté dans le chapitre 1, il existe deux grandes méthodes pour détecter et corriger des fautes au sein d'un registre ou plus largement d'un élément mémorisant une valeur. La première consiste à appliquer la triplication sur les registres et placer un retour permettant de corriger la valeur si elle a été changée [Carmichael, 2006]. La seconde consiste à utiliser des codes correcteurs d'erreur (ECCs). L'état de l'art ne permet pas directement de choisir l'une ou l'autre des solutions. La triplication n'augmente que peu la latence d'écriture et de lecture ($\approx 20\%$) mais entraîne un surcoût en surface d'environ 200% [Naseer et al., 2006]. Les codes ECCs n'ont pas un surcoût aussi important (entre 25 et 160% selon les implémentations, la taille de la RF à protéger et la longueur des mots, mais ils présentent un surcoût en latence d'écriture et/ou de lecture qui diminuera la fréquence de fonctionnement de l'architecture (entre 80 et 180% d'augmentation de latence) [Esmaeeli et al., 2011]. La triplication des registres permet néanmoins de corriger jusqu'à une faute par bit du registre, ce qui n'est pas le cas des codes correcteurs d'erreurs qui ont une capacité plus limitée. Il faudra tester les deux approches pour déterminer laquelle convient le mieux dans l'architecture de la tuile et correspond le plus au besoin de justesse de résultat. Avec la triplication, il est possible de corriger une erreur sur chacun des bits de la donnée, mais cette méthode expose trois fois plus les registres aux rayonnements et donc aux fautes.

5.3.2 Tolérance pour des calculs

Bien qu'il existe beaucoup d'autres méthodes moins coûteuses dans l'état de l'art (cf. chapitre 1), notre choix s'est porté sur la triplication pour les calculs car elle permet de réellement masquer les fautes transitoires et de facilement détecter les fautes permanentes comme cela sera illustré après.

Le concept de triplication impose seulement que chaque calcul soit effectué trois fois et qu'il y ait un vote majoritaire avant d'utiliser les résultats. La redondance peut être réalisée de trois façons différentes :

- uniquement architecturale, sans modifier le graphe de l'application : en faisant qu'une opération de l'application assignée sur l'opérateur d'une tuile soit tripliquée dans la tuile, de façon transparente ;
- logicielle, en ajoutant simplement un opérateur de vote de la tuile : en ordonnançant trois fois le même calcul potentiellement à des instant différents et en imposant que les ressources utilisées soient différentes ;
- architecturale et logicielle, en combinant les deux méthodes, par exemple en ordonnançant deux fois des calculs doublés matériellement et avec un vote majoritaire.

Ces trois grandes façons de mettre en œuvre la triplication n'ont pas les mêmes impacts sur l'architecture du CGRA, en particulier sur l'intérieur des tuiles, et sur le graphe de l'application à projeter. Trois variantes principales sont analysées dans la suite de ce chapitre et diffèrent essentiellement par la place de l'action du vote dans l'architecture.

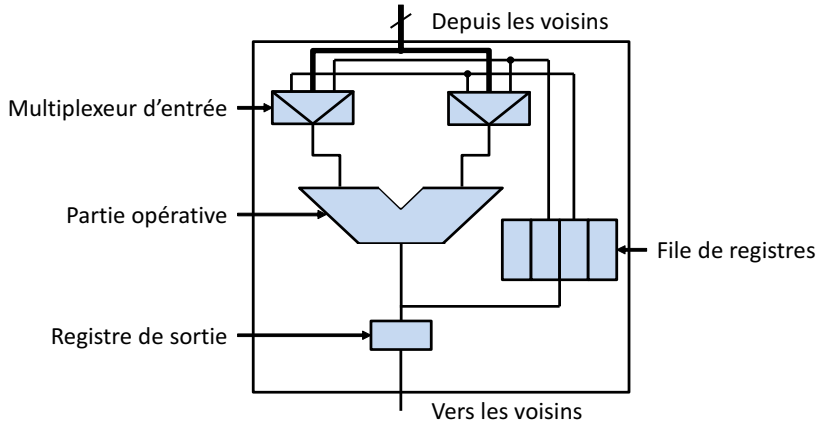


FIGURE 5.8 – Schéma simplifié d'une tuile.

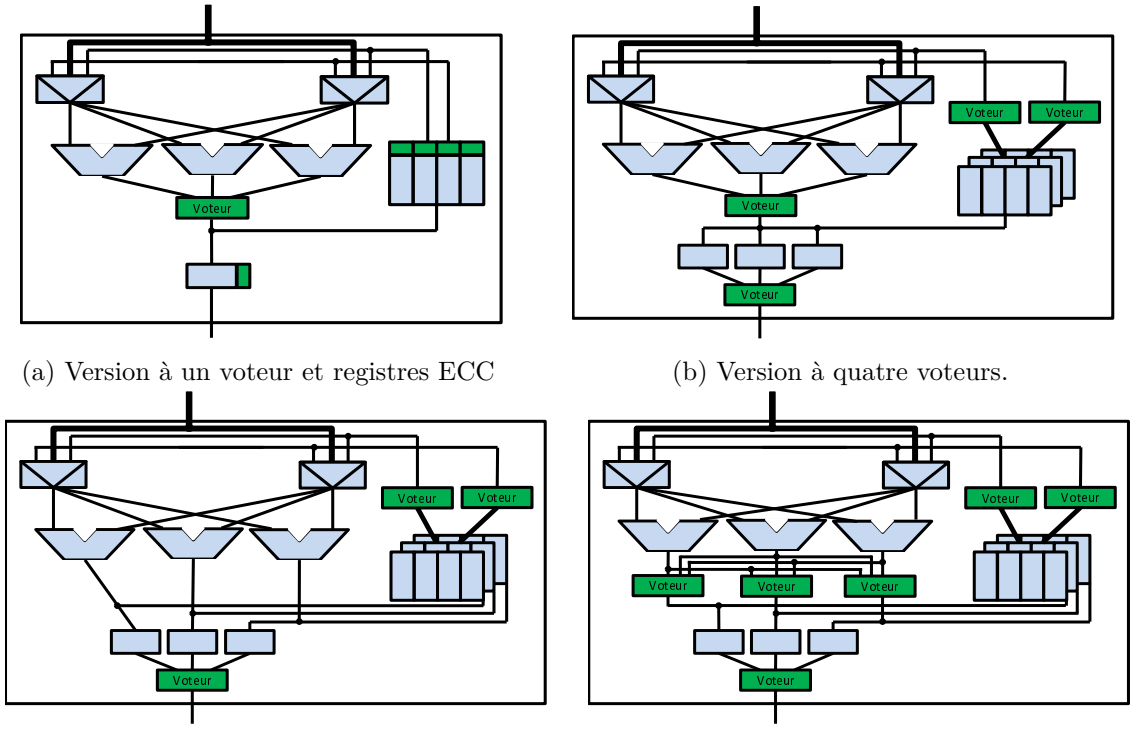
5.3.2.1 Triplication uniquement architecturale

La triplication uniquement architecturale modifie l'architecture interne de la tuile mais ne change pas le graphe de l'application. L'opérateur est triplé, le ou les voteurs sont ajoutés ainsi que toute la logique permettant de surveiller les occurrences des erreurs permettant de détecter les fautes permanentes. Le vote est situé après les opérateurs. D'un point de vue extérieur, la tuile est simplement plus « grosse » : elle contient d'avantage de ressources et produit des résultats corrects par construction. Fonctionnellement, elle possède un seul opérateur et la même interface que la version non tolérante aux fautes transitoires. Réaliser la triplication uniquement au niveau architectural permet donc d'utiliser la méthode de projection détaillée dans les chapitres précédents sans y apporter la moindre modification. Ce point est un avantage non négligeable, mais en contrepartie, si cette tuile présente une faute, l'intégralité de la tuile n'est plus utilisable (les ressources diminuent trois par trois).

Il existe cependant encore différentes manières pour la réaliser au niveau de l'architecture interne de la tuile. La suite de cette section présente ces différentes approches ainsi que leurs avantages et inconvénients. Pour illustrer ces différentes manières de réaliser une triplication uniquement architecturale, la tuile de la figure 5.8 sera utilisée comme point de départ pour les différentes variantes présentées en figure 5.9.

La figure 5.9a illustre une première manière de réaliser cette triplication. Elle consiste simplement à tripler les opérateurs et à ajouter un voteur qui donne son résultat aux registres dont la protection est effectuée par une implémentation de code correcteur d'erreurs (schématisée en vert au niveau des registres). Cette première implémentation présente comme avantages de n'avoir besoin que d'un seul voteur par tuile, d'être très simple à mettre en œuvre, de ne pas changer la pression sur l'interconnexion. Dans le cas de la figure 5.9b, il y a un voteur entre les opérateurs et les registres. Ainsi, les résultats mémorisés dans les registres seront corrects et les mêmes (sauf en cas de SEU entre le voteur et les registres). Dans la figure 5.9c, ce voteur est supprimé mettant un lien direct entre les sorties des opérateurs et les registres et diminuant ainsi le chemin critique. La figure 5.9d illustre l'implémentation conseillée par von Neumam [Neumann, 1956] : elle triple le vote et les résultats sont mémorisés dans des registres distincts. Les figures 5.9c et 5.9d illustrent deux autres implémentations n'utilisant pas non plus de code correcteur d'erreurs, mais tripliquants les registres. Elles diffèrent par la place des voteurs bien qu'elles imposent qu'il y ait un voteur pour la valeur sortant de la tuile et deux voteurs en sortie de la RF.

Le choix entre ces différentes versions n'est pas aisé *a priori* car l'impact sur la latence du mécanisme d'ECC et du vote ne sont pas connus. Les versions avec ECC et à trois voteurs semblent être les deux meilleurs candidats, les deux autres versions ajoutant des voteurs pour « seulement » mieux identifier l'origine d'une faute.



(a) Version à un voteur et registres ECC

(b) Version à quatre voteurs.

(c) Version à trois voteurs.

(d) Version à six voteurs.

Figure 5.9 – Exemples schématiques d'une tuile tripliquée de façon uniquement architecturale.

5.3.2.2 Triplication logicielle

La méthode logicielle consiste à tripler chaque opération dans le graphe et à y ajouter des opérations de vote. Les opérations de vote ont la particularité de ne pas avoir deux entrées, mais trois et sont suivies de variables. Ces variables sont ensuite connectées aux entrées des opérations suivantes (elles aussi tripliquées).

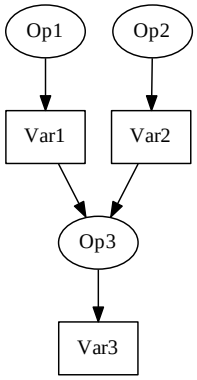
La figure 5.10 présente deux mises en œuvre différentes de cette triplication logicielle :

— en (a), un DFG (ou un BB) élémentaire de trois opérations est donné ;
— en (b), le résultat de la triplication logicielle avec une seule opération de vote est présentée ;
— en (c), le résultat de la triplication logicielle avec trois opérations de vote est illustrée.

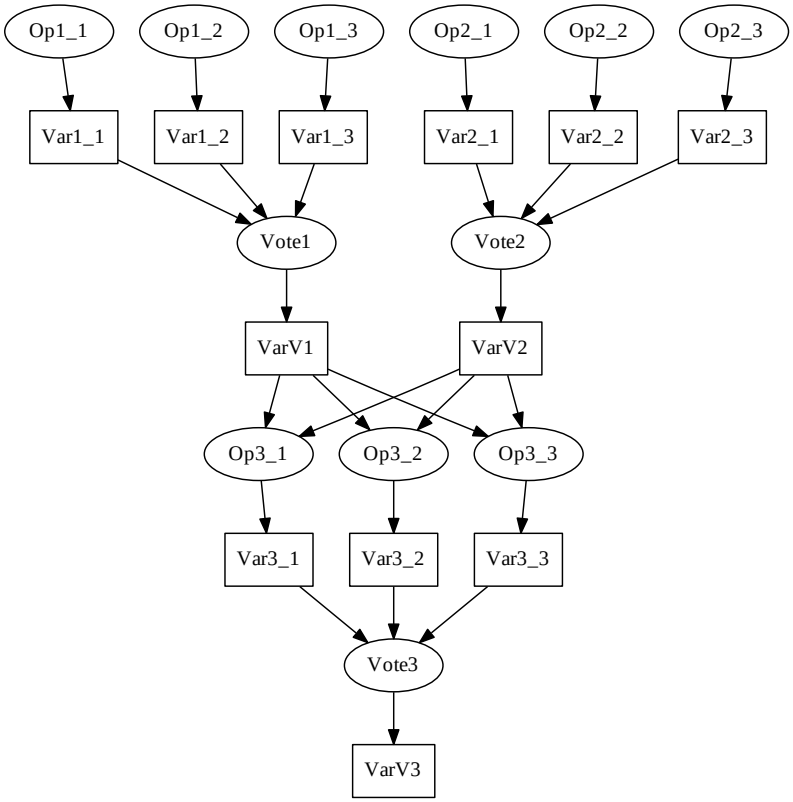
Ces deux mises en œuvre n'ont pas les mêmes conséquences sur le flot. Avec la première, ce sont les variables de sorties de votes qui vont être les plus contraignantes lors de la projection du fait qu'elle doivent toutes deux pouvoir accéder aux trois mêmes opérateurs. Avec la seconde, ce sont les opérations de votes qui vont contraindre le plus la méthode car les trois résultats des opérations triplées doivent accéder aux trois opérations de vote. Les conséquences dépendent du réseau d'interconnexion :

— si il s'agit d'un mesh-2D simple, alors il n'y a pas de solution de placement car deux tuiles différentes ne peuvent pas accéder à trois ressources identiques ;
— si le réseau d'interconnexion est plus riche (*e.g.* mesh-X, mesh-plus, etc.) alors il devient possible de les placer. Cependant, si les résultats sont consommés par plusieurs opérations, le problème de saturation de l'interconnexion peut se reproduire.

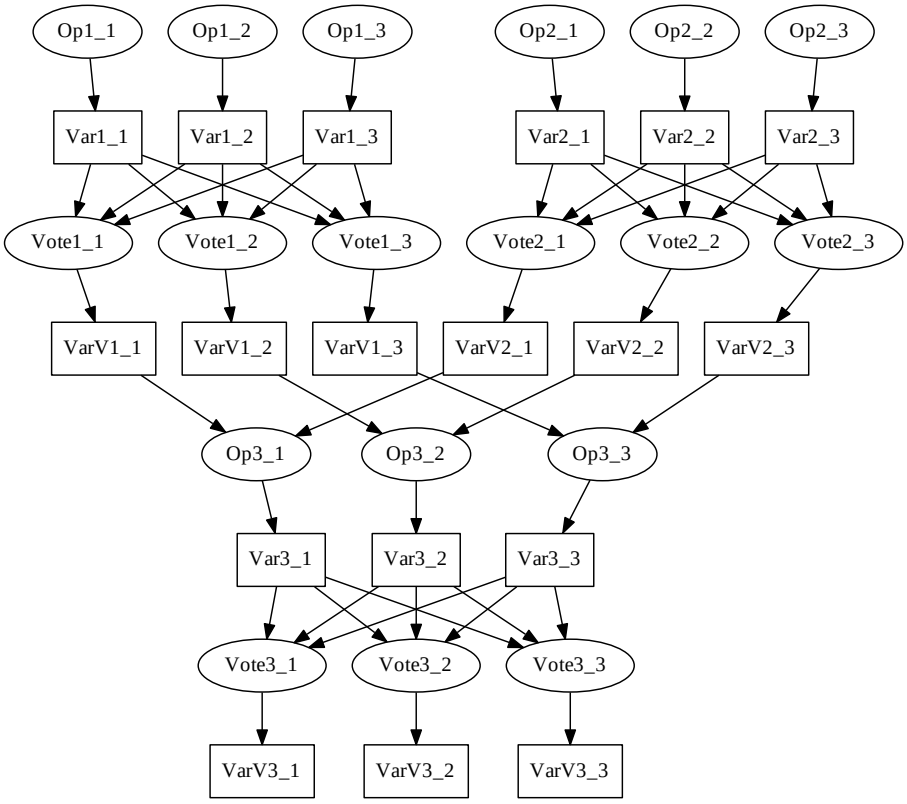
Matériellement, il faut un opérateur de vote qui ait le même rôle qu'un opérateur classique (c'est-à-dire qui soit « en parallèle » des autres opérateurs de la tuile, entre les multiplexeurs d'entrée et les registres). Comme l'opération de vote possède trois entrées, il faut que l'opérateur associé puisse accéder à trois opérandes. Pour cela, il y a deux solutions :



(a) Graphe original.



(b) Triplication logicielle avec une opération de vote.



(c) Triplication logicielle avec trois opérations de vote.

FIGURE 5.10 – Exemples de triplication logicielle.

— la première consiste à forcer le troisième opérande à provenir de la RF. Cette implémentation contraint la projection, car de ce fait, l'opération de vote doit se produire sur une des tuiles ayant effectué l'opération tripliquée. Ceci induit une faiblesse dans la sûreté du résultat dans le cas de la mise en œuvre de la figure 5.10(b) car si cette tuile est victime d'une faute permanente, alors le résultat du vote sera probablement erroné (dépendant de la faute). Or c'est ce résultat qui sera utilisé pour le reste des opérations, impliquant un résultat faux à la fin de l'exécution. Dans le cas de la figure 5.10(c), les votes seront placés aux mêmes endroits que les trois opérations, mais il n'y a pas de faiblesse supplémentaire qui apparait ;
— la seconde solution consiste à ajouter un troisième multiplexeur d'entrée pouvant obtenir des valeurs depuis les voisins et la RF comme pour les deux autres multiplexeurs d'entrée. Cette version est plus coûteuse en surface et pourrait entraîner une latence plus grande en augmentant le *fan-out* des registres, mais elle relâche une partie la contrainte du placement des opérations de vote en leur permettant d'être dans des tuiles voisines.

Vis-à-vis de la tolérance aux fautes transitoires, cette méthode permet une protection inférieure à la version uniquement architecturale car la « distance » entre deux votes est plus grande : deux registres, un opérateur et un multiplexeur, soit un registre de plus.

Au final, quel que soit le détail de la mise en œuvre, la triplication logicielle présente deux inconvénients majeurs :

— la pression sur le réseau d'interconnexion ;
— et la latence du graphe qui est par définition doublée car un étage d'opération de vote est ajouté après chaque opération « classique ».

Ce second inconvénient peut être réduit en diminuant la finesse de la triplication et en ne plaçant des opérations de vote que toutes les deux opérations (équivalent de la Selective TMR de [Samudrala et al., 2004]), mais cela empêche de pouvoir simplement monitorer la provenance des fautes et donc déterminer si une tuile est victime d'une faute permanente.

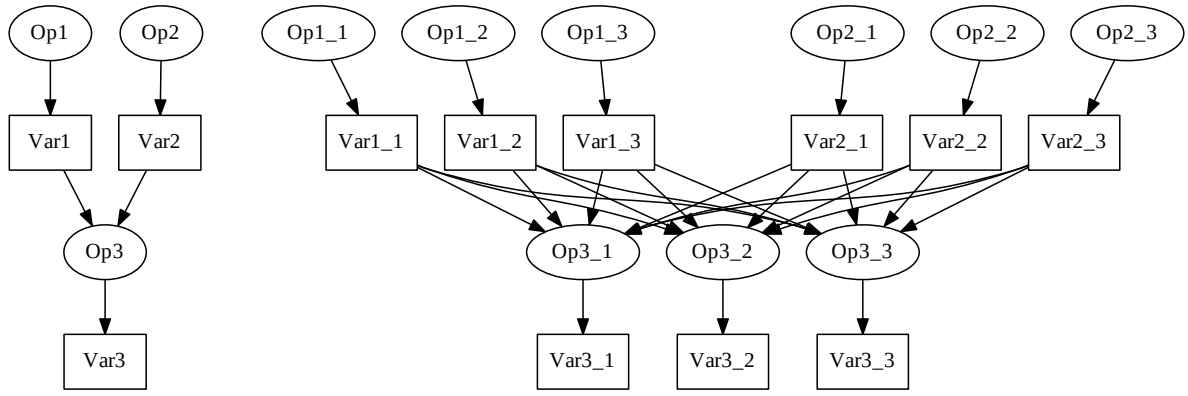
Mais d'un autre côté, bien que la détection de la provenance de la faute et donc le monitoring de l'état des tuiles soient plus complexes, elle permet d'utiliser toutes les ressources opérationnelles. En effet, dans la version uniquement architecturale, quand une faute permanente est détectée, c'est l'ensemble de la tuile, soit trois fois les ressources, qui n'est plus utilisée. De plus la méthode logicielle n'augmente pas le chemin critique au sein de la tuile car il y a toujours le même nombre d'éléments à traverser : un multiplexeur et un opérateur (de vote ou classique).

5.3.2.3 Triplications hybrides

Des méthodes intermédiaires à ces deux approches sont possibles. Elles impactent à la fois le graphe et l'architecture pour mettre en œuvre la tolérance aux fautes. Deux versions sont présentées dans les paragraphes suivants.

Opérateurs à six entrées

La première de ces deux versions intermédiaires consiste à déplacer les opérations (et donc les opérateurs) de vote. Au lieu de les placer en sortie de l'opérateur, elles sont placées directement en entrée de celui-ci juste après les multiplexeurs. Concrètement, il faut alors six multiplexeurs d'entrée qui permettent d'alimenter les deux voteurs qui fournissent les opérandes à l'opérateur. La figure 5.11b illustre les transformations qu'il faut effectuer dans le graphe de la figure 5.11a pour réaliser cette méthode. Pour la même raison que la triplication logicielle, elle ajoute une très forte contrainte sur le réseau d'interconnexion. Et même encore plus forte car un réseau d'interconnexion à six voisins ne permet pas de placer ne serait-ce que le graphe de la figure 5.11b. Cette méthode a besoin d'au moins un réseau d'interconnexion possédant huit voisins dans les cas sans trop de dépendances de données et plutôt un « mesh-X-plus » à douze voisins pour être moins contraint et accepter des graphes avec des nœuds possédant plusieurs successeurs.



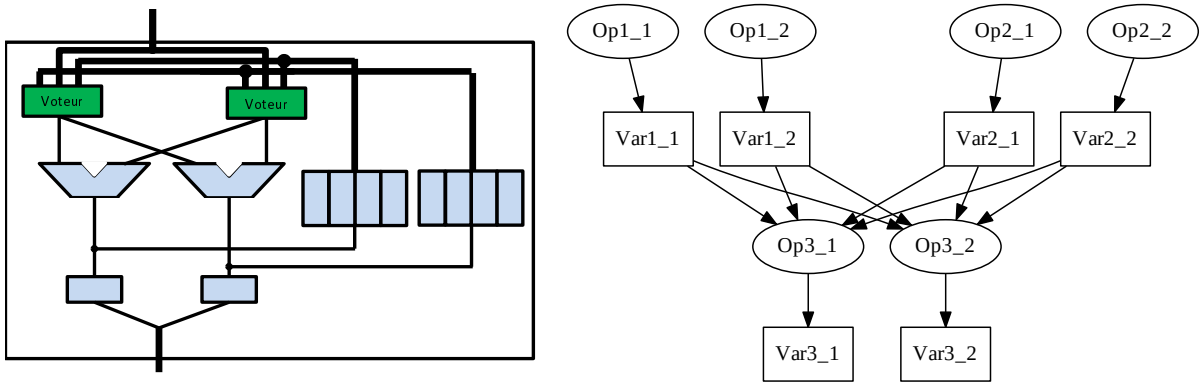
(a) Graphe original. (b) Résultat tripliqué.

FIGURE 5.11 – Résultat de la triplication hybride par opérateurs à six entrées.

Cet inconvénient limite l'utilisation de cette implémentation malgré sa meilleure utilisation des ressources valides (comme la triplication logicielle) sans doubler le nombre de cycles d'exécution. Elle aurait un chemin critique similaire à la triplication uniquement architecturale.

Double Duplication

La seconde version intermédiaire de triplication présentée dans cette section répartit la triplication entre le graphe et l'architecture. Pour cela, les opérations du graphe et l'intérieur des tuiles sont dupliqués. Les deux opérations du graphe sont assignées sur deux tuiles différentes et sur chaque tuile, le calcul est effectué deux fois. La figure 5.12 illustre l'architecture de la tuile en 5.12a et le graphe provenant de la figure 5.8 dupliqué en 5.12b, tous deux résultant de l'utilisation de cette méthode de tolérance. Les opérations sont ainsi effectuées quatre fois, mais avec seulement deux provenances réduisant la pression sur l'interconnexion, ce qui est visible dans le graphe. Les quatre occurrences sont ensuite transmises aux opérations suivantes qui effectueront le vote en entrée d'opérateur comme dans la version « opérateur à six entrées ». Seuls trois des quatre résultats sont nécessaires pour effectuer le vote majoritaire. Ceci permet de tolérer une ressource défectueuse parmi les quatre sans avoir à se reconfigurer : l'application continue d'utiliser les mêmes tuiles, c'est simplement le choix des entrées du vote qui doit changer.



(a) Schéma d'une tuile. (b) DFG dupliqué.

FIGURE 5.12 – Architecture et graphe résultat de l'utilisation de la double duplication pour effectuer la triplication.

L'inconvénient de cette méthode est qu'elle impose de transmettre le double de données avec par exemple un bus de largeur deux fois égale à celle d'un mot. De plus, il faut ajouter un *flag*, permettant de savoir si la donnée provient d'une ressource défectueuse, à chaque mot de donnée pour que le vote s'effectue avec les bonnes entrées. Les avantages principaux de l'utilisation de cette variante sont que les ressources sont « mises de côté » moins vite (uniquement deux par deux et quand il y a moins de trois parmi quatre données valides) et qu'elle n'augmente pas vraiment la pression sur l'interconnexion. En effet, il est tout à fait envisageable d'utiliser cette méthode avec un réseau de tuiles connectées par un mesh-2D simple.

5.3.2.4 Évolutions possibles

Nous avons cherché des moyens de diminuer l'inconvénient majeur de la triplication uniquement architecturale qui est qu'à chaque fois qu'une faute permanente apparaît, les deux autres ressources de la tuile ne sont plus utilisées bien que valides. Dans cette sous-section, deux améliorations sont proposées.

Fusion architecturale et double duplication

L'idée de cette évolution consiste à permettre à l'architecture le changement de mode de triplication de la version uniquement architecturale à la double duplication. Ainsi, tant qu'il est possible d'exécuter l'application avec la triplication uniquement architecturale, c'est elle qui est utilisée. Puis quand il n'y a plus assez de ressources, l'architecture changerait de fonctionnement pour effectuer de la double duplication permettant de rallonger la durée d'utilisation de l'architecture.

Pour cela, il faut que la tuile contienne à la fois les ressources permettant d'effectuer la triplication en interne et à l'aide d'une autre tuile. Il faut donc :

- que les liens de communication permettent le passage de deux mots de données ;
- qu'il existe des voteurs à la fois en sortie d'opérateur (un) et en entrée (deux) et qu'il soit possible de choisir l'utilisation de l'un puis de l'autre ;
- que la logique permettant de détecter les fautes permanentes soit fusionnée ;
- qu'il y ait la capacité de stocker deux fois un résultat, ce qui impose de ne pas utiliser les code correcteurs d'erreur au niveau des registres mais de la triplication « simple ».

Tous ces éléments induisent un surcoût non-négligeable en plus de l'ajout de ressources devant piloter l'ensemble et qui sont sensibles, elles aussi, aux rayonnements et susceptibles d'être affectées par des fautes. Il faudra une évaluation complète, en particulier sur l'augmentation de la durée de vie du composant global, pour savoir si ce coût est justifié ou non.

Méthode architecturale avec ressources supplémentaires

Cette seconde évolution est moins spéculative que la précédente. Elle consiste à adapter un mécanisme déjà existant dans l'état de l'art pour permettre de ne pas « mettre de côté » une tuile tripliquée en local dès qu'une faute permanente est détectée. Dans [Rakossy et al., 2013], l'idée d'ajouter des ressources supplémentaires qui ne sont pas utilisées tant qu'il n'y a pas de faute permanente est introduite (cf. 1.3.2.2). Cette idée peut être adaptée dans le cadre de la triplication au sein même d'une tuile. Par exemple dans la figure 5.13, il y a quatre ressources de calcul dans la tuile pour effectuer la triplication. Une ressource est de trop et ne sera pas utilisée. L'idée est alors de changer de manière périodique la ressource qui n'est pas utilisée pour pouvoir soit la laisser simplement éteinte soit effectuer de l'auto-test (proposition de [Rakossy et al., 2013]). De cette manière chaque tuile peut tolérer deux fautes permanentes avant de ne plus être utilisée passant ainsi d'un ratio de 2/3 à $2/4 = 1/2$. Autrement dit, au lieu de ne plus utiliser une tuile alors que 2/3 de ses ressources sont encore valides, une tuile ne sera plus utilisée que lorsqu'elle n'aura plus que la moitié de ces ressources valides. Le prix à payer est le nombre de tuiles par unité de surface qui sera plus faible car dans ce cas, les tuiles contiennent quatre fois plus de ressources que la version non protégée.

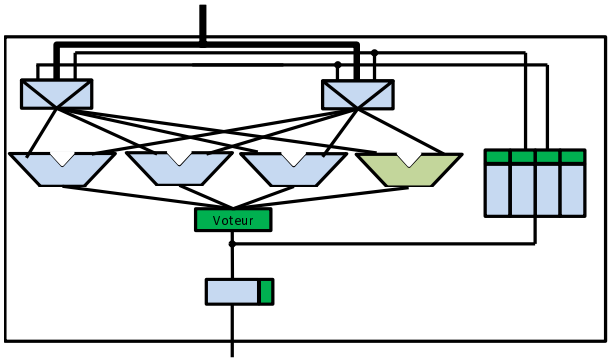


FIGURE 5.13 – Illustration de l'ajout de ressources suplémentaires pour la triplication uniquement architecturale avec registres protégés par ECC.

Pour déterminer le meilleur compromis, il faut évaluer le changement de taille de tuile que provoque l'ajout d'une ressource (et donc la diminution du nombre total de tuiles) par rapport au nombre de fautes tolérables. Un modèle simpliste ne considérant que les ressources de calcul permet de dire que le nombre total de tuiles est inversement proportionnel au nombre de fautes permanentes tolérées (au niveau des ressources) dans la tuile. Il tendrait à pousser la réalisation de tuiles possédant un très grand nombre de ressources avec comme contrainte de réaliser le nombre de tuiles permettant d'exploiter le parallélisme maximum de l'application à projeter. Cependant, les fautes permanentes ne se produisent pas que dans la partie calculatoire, mais aussi dans la partie mémorisation. De ce fait cette tendance est erronée car perdre trop de ressources de mémorisation empêche l'utilisation de la tuile même s'il reste des ressources valides.

5.3.3 Bilan

Le tableau 5.2 présente une comparaison qualitative des impacts des différentes méthodes sur quatre critères : la surface de la tuile, le temps d'exécution, la pression induite sur le réseau d'interconnexion et la possibilité d'évolutions. La « durée d'exécution » n'est pas relative qu'à la fréquence qui sera obtenue par réalisation de la tuile, mais prend aussi en compte l'ajout des opérations de vote dans la méthode logicielle qui double le nombre de nœuds à exécuter du graphe. À ce stade de l'analyse, aucune méthode n'est véritablement éliminée car elles représentent toutes un compromis différent mais les versions ECC, à trois voteurs et double duplication semblent être les meilleurs candidats. La version à six entrées n'est intéressante que si le réseaux d'interconnexion est très riche. Le doublement de latence des versions logicielles les rend relativement peu utilisables. La section suivante étudie l'impact de chacune d'elles sur la capacité à détecter précisément l'origine d'une faute.

Tableau 5.2 – Comparaison des impacts des différentes méthodes pour les fautes transitoires.

Méthode	Architecturale		Logicielle		Hybride	
	voteur/ECC	3, 4, 6 voteurs*	1 vote	3 votes	6 entrées	Double duplication
Impact surface tuile	Fort	Très fort	Faible		Faible	Modéré à fort
Impact temps d'exécution	Modéré	Faible	Fort		Faible à modéré	
Pression sur l'interconnexion	Aucune supplémentaire		Modérée	Forte	Très forte	Faible à modérée
Évolutions	2		0		0	1

* les variantes sans ECC sont regroupées car un voteur est petit par rapport à l'opérateur et aux registres et ajouter un voteur sur le chemin de données a moins d'impact que l'ECC [Esmaeeli et al., 2011] : .

5.4 Mise en œuvre de la détection de fautes permanentes

Comme proposé dans le chapitre 2, la tolérance aux fautes permanentes s'effectue par reconfiguration dynamique en arrêtant d'utiliser les ressources défectueuses. Ceci est fait en deux étapes. La première consiste à détecter qu'une faute permanente s'est produite dans une ressource. La seconde étape consiste à trouver une nouvelle configuration et à reconfigurer le composant. La première étape, détaillée dans cette section, présente l'impact de la technique de tolérance aux fautes transitoires sur la méthode de détection des fautes permanentes. En effet, la mise en œuvre de la tolérance aux fautes transitoires influe grandement l'ensemble des mécanismes à mettre en œuvre pour pouvoir suivre l'état des différentes ressources. Néanmoins, cette détection se base toujours sur le même principe qui est décrit au paragraphe suivant.

5.4.1 Méthode de suivi

Pour détecter qu'une faute permanente s'est produite, il faut pouvoir comparer un résultat attendu avec le résultat réellement obtenu. Pour ce faire, il existe deux grandes stratégies. La première consiste à effectuer périodiquement des tests du composant pendant le fonctionnement comme dans l'approche [Rakossy et al., 2013]. Ce n'est pas l'approche retenue dans notre cas de part sa réactivité limitée, mais elle n'est pas incompatible comme montré au paragraphe 5.3.2.4. La seconde consiste à se baser sur les mécanismes de correction de fautes transitoires existants dans le composant pour détecter les fautes permanentes.

Le vote majoritaire ou la correction ECC va permettre de déterminer les différents bits erronés. Cette information sera transmise à un compteur qui permettra d'analyser dans la durée le nombre de fautes détectées [Aliee and Zarandi, 2011]. La notion de temps est indispensable car, pour un cycle d'exécution, une faute transitoire aura les même effets qu'une faute permanente. Pour les différencier, il faut regarder sur plusieurs cycles si une ressource donne majoritairement un résultat faux. Une approche naïve consisterait à compter le nombre de fois d'affilée qu'une ressource donne un mauvais résultat. Cependant une telle approche n'est pas suffisante car une des fautes classiques auxquelles sera confronté le composant est le « collage » d'un bit à une valeur particulière (*e.g.* « *stuck at* 0 »). Si c'est le cas, alors le vote majoritaire ne pourra pas voir qu'il y a une faute si la valeur de collage est la même que celle attendue. Il y a donc un compromis à trouver permettant de discriminer des fautes transitoires apparaissant sur la même partie du composant et une faute permanente à visibilité « intermittente » (par exemple sept résultats faux sur les dix derniers) et ce indépendamment de l'implémentation retenue pour la tolérance aux fautes transitoires. Concernant la finesse du suivi des ressources, deux grandes politiques différentes s'opposent mais peuvent être combinées pour profiter de leurs avantages et réduire leurs inconvénients respectifs.

Suivi à grains fins

La première politique de suivi consiste à suivre les erreurs bit par bit. Cela permet d'éviter de rejeter une ressource qui par « malchance » a été victime plusieurs fois de suite de fautes transitoires mais sur des bits différents et ainsi d'éviter des faux-positifs. L'inconvénient majeur de cette approche est qu'elle nécessite un nombre de compteurs de suivi de nombre de fautes d'affilées égal à la largeur des mots de données. De ce fait, pour une architecture travaillant avec des mots de 32 bits, il faut 32 compteurs. En plus du coût en termes de surface, ces compteurs présentent l'inconvénient d'être eux aussi sensibles aux fautes. Donc multiplier leur nombre n'est pas profitable par rapport à la tolérance aux fautes.

Suivi à gros grains

La deuxième politique, à l'opposée de la première, consiste à regarder si une erreur (ou plus) s'est produite sur un mot de donnée, peu importe le positionnement de cette erreur. Une telle politique présente l'avantage de ne pas multiplier le nombre de compteur : un par ressource. Mais

par contre elle va augmenter le nombre de faux positif. Cette augmentation va entraîner une diminution plus rapide du nombre de ressources disponibles dans l'architecture et ainsi diminuer la durée de vie totale du composant, sans pour autant que les ressources ne soient véritablement défaillantes. Si cette politique est choisie, il faudra évaluer l'utilité pour la durée de vie d'ajouter du test permettant de confirmer *a posteriori* la défaillance de la ressource et si possible de la réutiliser par rapport au coût d'un tel ajout.

Suivi à granularité intermédiaire

La troisième politique de suivi est une version intermédiaire des deux précédentes. L'idée est de ne pas suivre finement chaque bit, mais de suivre des regroupements de bits. Il est même possible d'envisager un suivi hiérarchique des bits de données. Par exemple sur un mot de 32 bits, supposons que les bits soient regroupés par 4. Il y a donc 8 compteurs pour ces groupes. Il est possible de placer le seuil de faute permanente à 6 pour les 8 derniers résultats. Un autre compteur peut alors être ajouté avec un seuil plus élevé pour l'ensemble du mot de donnée. De cette manière, le surcoût en compteurs est limité tout en conservant une bonne précision de détection. L'exemple présenté ici peut être étendu à un nombre plus important de niveaux hiérarchiques.

Le suivi à grains fins n'est pas pertinent du fait de son coût en surface et de sa vulnérabilité aux fautes. Le suivi à gros grains peut certainement être un choix pour prototyper rapidement, mais le suivi à granularité intermédiaire semble être la méthode la plus pertinente. Mais le choix précis de la granularité, de la hiérarchisation et des seuil n'est pas simple et nécessiterait une étude approfondie qui permettra de trouver le meilleur compromis entre le nombre de faux positifs et le coût d'implémentation.

5.4.2 Détection avec la méthode architecturale

Dans cette sous-section, l'hypothèse faite est que la méthode uniquement architecturale (5.3.2.1) est utilisée pour effectuer la correction de fautes transitoires. Dans cette mise en œuvre, l'intégralité des informations nécessaires au suivi est présente localement dans la tuile : les ressources de calcul, les registres et les voteurs. Plus précisément, dans la première version (figure 5.9a), il y a un voteur pour les calculs et de la correction ECC pour les registres. Il est donc possible de surveiller indépendamment la partie calculatoire et la partie registres en ajoutant simplement des compteurs associés aux ressources. C'est également le cas des configurations présentées en figure 5.9b et 5.9d.

L'architecture de la figure 5.9c ne possède pas de voteur entre la partie calculatoire et la partie registre. Il n'est donc pas possible de savoir si la faute provient de la partie calculatoire ou de la mémorisation. Ceci rend impossible la mise en place d'une méthode comme celle de [Abella et al., 2010] qui permet de re-numéroter les registres pour ne plus utiliser ceux qui seraient défectueux et permet de ce fait de ne considérer que la tuile n'est plus utilisable uniquement quand il n'y a réellement plus assez de registres.

Cependant, dans le flot proposé, la granularité de la reconfiguration, c'est-à-dire la différenciation entre deux configurations, se fait au niveau de la tuile et il n'est donc pas nécessaire d'avoir une finesse supérieure pour le suivi de l'état des ressources. Cela permettra simplement une éventuelle amélioration de la tolérance. Avec l'implémentation uniquement architecturale, détecter et signaler une faute permanente est donc relativement simple et peu coûteux en ressources (simplement les compteurs).

5.4.3 Détection avec la méthode logicielle

Dans cette sous-section, l'hypothèse faite est que la méthode logicielle (5.3.2.2) est utilisée pour effectuer la correction de fautes transitoires. Avec cette méthode, les valeurs à comparer peuvent provenir de la tuile locale et de deux autres tuiles. Quand l'opération de vote s'effectue, elle va désigner le ou les bits qui sont erronés d'une ou plusieurs provenances. Plusieurs cas se présentent alors :

— si l'erreur provient de la tuile effectuant le vote, la tuile peut incrémenter son compteur de suivi de vote ;
— si l'erreur provient d'un voisin, il faut que la tuile votante signale à ce voisin que son résultat était erroné tout en transmettant les éventuelles données à transférer.

Il est donc nécessaire que le réseau d'interconnexion possède une liaison supplémentaire de la même largeur que la finesse du suivi des ressources. Avec cette mise en œuvre de la tolérance aux fautes, il serait donc préférable de choisir un suivi à gros grains ou éventuellement intermédiaire pour limiter le surcoût dans l'interconnexion.

Cependant, la triplication logicielle présente un autre inconvénient majeur et même rédhibitoire : la prise de décision. Contrairement à la méthode uniquement architecturale pour laquelle toute la triplication est réalisée dans une seule tuile et qui ne pose pas de soucis pour l'identification de la ressource fautive ; la méthode logicielle possède un ou plusieurs votes déportés qui nécessitent un retour. Le problème est alors de savoir qui doit considérer ce retour. La réponse dépend de la réalisation de la triplication avec un voteur ou avec trois voteurs. Avec un seul voteur, si à un cycle donné un voteur détermine qu'il y a une erreur, il y deux possibilités :

— les deux résultats des votes précédents (pour les deux opérandes) ont été utilisés dans des tuiles différentes de celles ayant effectué les votes, alors c'est la tuile où l'opération inter-vote a été effectuée qui est fautive ;
— dans tous les autres cas, le résultat d'au moins un des votes précédents a été stocké à la fois dans la RF et dans le registre de sortie. Il est donc possible qu'une faute transitoire se soit produite dans l'un et pas dans l'autre. De ce fait, il n'est pas possible de savoir si la faute provient de la tuile ayant effectué l'opération inter-vote ou d'un des registres qui contenaient un des opérandes. Cette ambiguïté n'est pas présente avec la première possibilité car, si une faute est présente dans le registre de sortie, alors les trois opérations suivantes utilisent l'opérande erronée et le voteur ne peut pas détecter d'erreur (bien que le résultat soit faux).

Cette incertitude permet de conclure qu'il est préférable de ne pas utiliser la version de la triplication logicielle à un seul voteur.

Dans le cas à trois voteurs, le problème de la détermination de la tuile fautive ne se pose pas de la même façon. Comme il y a trois votes pour chacun des résultats des opérations tripliquées, la tuile qui a fourni un résultat aux voteurs peut se trouver dans deux situations :

— la première situation est celle où les trois retours des voteurs ne sont pas identiques. Dans ce cas, la tuile sait qu'il s'agit d'une faute transitoire qui a eu lieu entre son registre et l'un des voteurs et ne doit pas prendre en compte ce retour pour la détection de fautes permanentes ;
— la seconde situation correspond à celle où les trois voteurs sont d'accord. Dans ce cas, le seul moyen de savoir si la faute provient de la tuile qui a effectué l'opération inter-vote ou bien du résultat du vote précédent est le même que pour le cas à un seul voteur, à savoir comparer ce que disent les voteurs pour d'autres utilisations du résultat vote. Si les différentes opérations inter-votes issues d'un même résultat de vote sont fautives alors il est raisonnable de penser que la faute provienne de la tuile du vote précédent. Dans le cas contraire, c'est la tuile ayant exécuté l'opération inter-vote qui est fautive.

Dans la pratique, l'implémentation de ce retour à deux niveaux est tout aussi peu réaliste et la conclusion est donc la même que pour la triplication à un seul voteur : il est préférable de ne pas utiliser cette méthode de tolérance aux fautes transitoires pour détecter les fautes permanentes.

5.4.4 Détection avec les méthodes hybrides

Dans cette sous-section, l'hypothèse faite est qu'une des méthodes hybrides (5.3.2.3) est utilisée pour effectuer la correction de fautes transitoires.

Avec les opérateurs à six entrées

L'utilisation d'opérateur à six entrées implique l'utilisation de deux voteurs en entrée d'opérateur. De ce fait, lorsqu'une erreur est détectée par un des voteurs, il doit communiquer à la tuile fautive qu'elle a transmis une donnée erronée. Il faut donc que le réseau d'interconnexion prévoit des chemins de retour dédiés au suivi des ressources en plus des chemins de données classiques, augmentant de manière non négligeable la largeur de l'interconnexion (en fonction de la granularité du suivi). Concernant la prise de décision, il n'y a pas le même problème qu'avec la tolérance logicielle. Une tuile reçoit *a minima* trois retours de voteurs. Trois cas de figures peuvent se présenter :

— les retours ne signalent pas d'erreur : il n'y a rien à faire ;
— tous les retours signalent une erreur : la tuile est fautive et il faut incrémenter le compteur de faute ;
— un ou plusieurs retours (mais pas tous) signalent une erreur : deux politiques de décisions peuvent être mises en place.

La première politique consiste à dire que si une majorité des voteurs signale une erreur, alors la tuile est fautive et le voteur qui ne signale pas d'erreur à lui aussi subit une faute, qui sera signalée au prochain vote. La seconde politique consiste à supposer que la probabilité qu'une erreur multiple MBU se produise est très faible. Dans ce cas, si au moins deux retours signalent une erreur, alors la tuile va considérer qu'elle est fautive. En revanche si simplement un retour signale une erreur, la tuile va considérer qu'il s'agit d'une erreur transitoire lors de la transmission de la donnée et donc ne va pas incrémenter son compteur de suivi. Il est possible d'appliquer cette même politique avec un seuil supérieur à deux, par exemple trois, ce qui revient alors à considérer que la probabilité d'apparition de fautes doubles ou DEUs n'est pas négligeable mais que celle de fautes triples ou TEUs l'est.

Dans le cas où le seuil est fixé à deux, si le résultat d'une opération n'est utilisé que par une seule autre opération, les deux politiques prendront la même décision. Si le résultat est utilisé plus d'une fois, alors le comportement sera différent. Il faudra tester les deux approches pour déterminer celle qui offre le meilleur compromis pour la durée de vie globale du composant.

Avec la double duplication

Le fait d'utiliser de la double duplication ne change pas beaucoup d'éléments du point de vue de la détection de fautes permanentes par rapport à l'utilisation d'opérateur à six entrées. Il faut un retour double (un pour chaque exemplaire de la donnée) par tuile voisine. Le suivi de l'état des ressources doit se faire de manière séparée entre les deux voies d'une même tuile permettant ainsi d'utiliser la voie fonctionnelle si l'autre est victime d'une faute permanente. La prise de décision se fait de la même façon que dans le cas des opérateurs à six entrées.

5.4.5 Bilan

Le bilan de cette analyse est différent du précédent : la méthode logicielle n'est plus envisageable car elle ne permet pas de détecter de façon précise quelle tuile est fautive lors de la détection d'une erreur. Avec les méthodes hybrides, pour la détection de fautes permanentes, il peut exister une incertitude qu'il faut lever en utilisant une des politiques sur le nombre de retours qui incrémentent le compteur. Il n'y a pas de souci de détection avec la triplication uniquement architecturale car tout se passe en local dans la tuile. Il sera donc nécessaire de tester la triplication architecturale, la double duplication et éventuellement celle dite des opérateurs à 6 entrées dans le cas où le réseau d'interconnexion est très riche.

5.5 Architecture système et fonctionnement

Dans les parties précédentes de ce chapitre, nous avons vu comment l'architecture de la tuile lui permet de tolérer les fautes transitoires et détecter les fautes permanentes. Dans le chapitre 4, nous avons vu comment générer et gérer un grand nombre de *mappings* pour le système. Cette section décrit l'architecture globale du système et son fonctionnement de manière à pouvoir assurer un traitement des données sans interruption et ce, même en cas d'apparition de fautes permanentes.

5.5.1 Système « Ping Pong »

L'architecture globale du système utilise comme brique de base un CGRA dont les tuiles ont été rendues tolérantes aux fautes transitoires et sont capables de dire si elles sont victimes d'une faute permanente. Associé au CGRA, le gestionnaire de reconfiguration permet de trouver la nouvelle configuration du composant de manière à poursuivre l'exécution le plus longtemps possible. Seulement il n'est pas raisonnable de penser que le temps de reconfiguration sera systématiquement suffisamment court pour ne pas interrompre le fonctionnement ou n'entraînera pas un retard inacceptable dans le traitement. Dans notre contexte, le temps imparti est de l'ordre de la centaine de millisecondes. Il n'est donc pas toujours envisageable d'avoir le temps de ré-écrire le contenu des mémoires de programmation de chacune des tuiles dans ce délais. Il faut donc avoir un autre moyen d'assurer la continuité du fonctionnement.

Nous avons choisi de le faire par redondance. Pour cela l'architecture du système global doit contenir plusieurs exemplaires du CGRA. Ces CGRAs ne fonctionnent pas simultanément dans le cas normal et permettent de basculer les traitements d'un CGRA vers un autre le temps de la reconfiguration. La figure 5.14 donne une illustration schématique de l'architecture « Ping Pong » avec deux voies. Dans cette figure, les CGRAs sont des 4×4 avec un réseau d'interconnexion de type mesh-2D torique. Le gestionnaire de reconfiguration est extérieur et pilote la reconfiguration des deux voies. Il s'agit d'un processeur qui doit tolérer les fautes et être capable d'exécuter l'algorithme de reconfiguration défini dans le chapitre précédent.

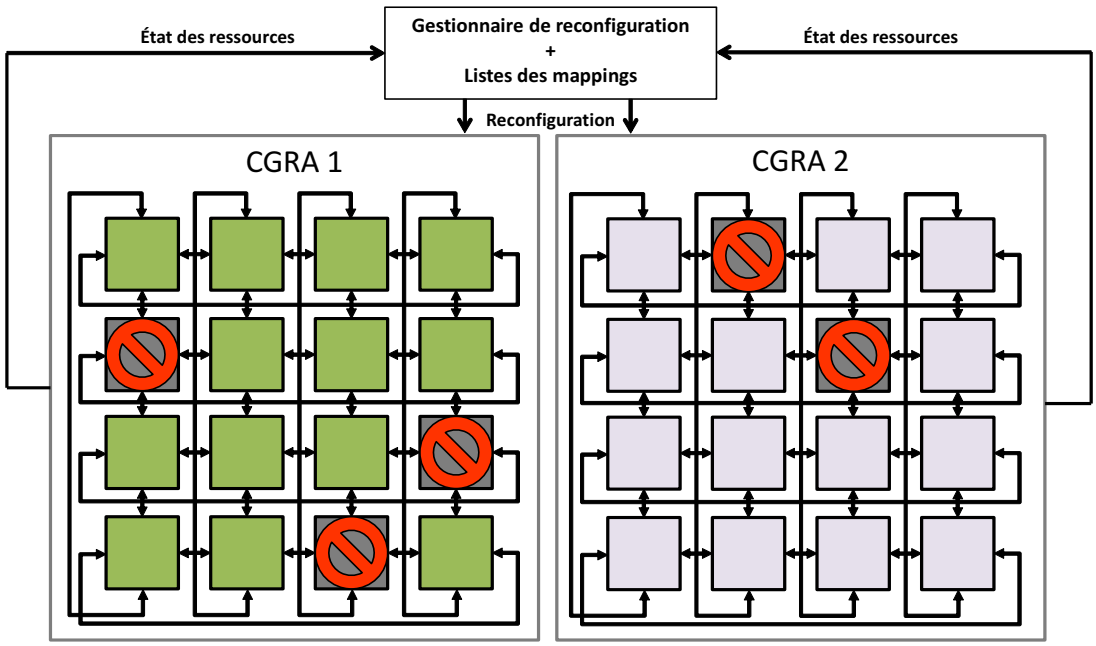


Figure 5.14 – Schéma du système utilisant le fonctionnement « Ping Pong ». Le CGRA de gauche est en fonctionnement et possède trois tuiles défectueuses, celui de droite a deux tuiles défaillantes.

Pendant la phase de fonctionnement le système se comporte comme suit :
— le premier *mapping* de l'application est porté sur la première voie de fonctionnement ;
— en cas de détection d'une faute permanente, une autre voie est activée avec le code correspondant à son état actuel, puis le fonctionnement bascule vers cette voie. La qualité des résultats fournis pendant le temps de basculement est assuré par le fait que la triplication donnera un résultat majoritairement correct même avec une faute permanente de détectée (sauf si une faute transitoire apparaît sur le même bit que la vote permanente sur un des deux autres exemplaires de la données) ;
— le gestionnaire de reconfiguration cherche alors quelle configuration est disponible pour la voie fautive parmi celle trouvée pendant la phase hors ligne et n'utilisant pas la ressource défectueuse. Il met ensuite à jour la liste de configurations restantes pour la voie. Il peut aussi éventuellement prévenir l'utilisateur de la dégradation de l'architecture (pour maintenance préventive par exemple) ;
— le fonctionnement ne basculera à nouveau sur la première voie que lorsque une faute permanente sera détectée sur la seconde (si deux voies de fonctionnement) ;
— et ainsi de suite tant qu'il existe des configurations permettant de fonctionner.

Comme précisé précédemment, le contrôleur peut pré-calculer, pendant le temps où l'architecture ne présente pas de faute permanente, les différents scénarios possibles pour permettre une reconfiguration plus rapide. Par exemple, si les tuiles 1, 2, 3, 6, 9 et 10 sont utilisées, déterminer au moins une configuration dans le cas où la tuile 1 est victime d'une faute, puis la tuile 2, etc.

5.5.2 Amélioration de l'architecture de la tuile

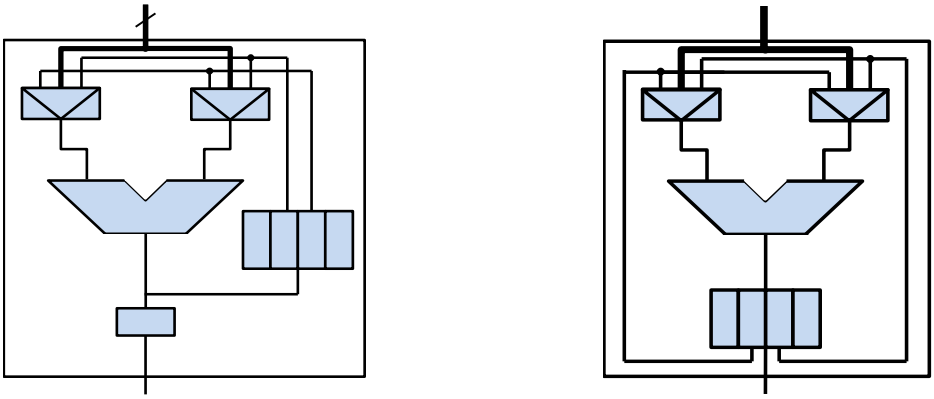
Dans cet chapitre, les réflexions sur l'implémentation de la tolérance aux fautes se sont basées sur une architecture « assez classique » d'une tuile, c'est-à-dire des multiplexeurs d'entrées, une partie opérative, une RF interne à la tuile et un registre de sortie. Dans le cadre de la tolérance aux fautes, cette tuile possède deux faiblesses qu'il faudrait supprimer pour permettre d'augmenter encore la durée de vie globale du système.

5.5.2.1 Architecture et contrôle de la RF

La première faiblesse vient de la méthode de reconfiguration qui fixe de manière statique le placement des données dans les registres. Or si un registre qui est utilisé régulièrement est victime d'une faute permanente, la tuile entière sera déclarée comme étant défectueuse. Elle ne sera plus utilisée alors qu'il peut rester suffisamment de registres valides dans la RF pour continuer à fonctionner s'ils étaient utilisés à la place des registres fautifs. L'idée est alors de décorréler l'affectation présente dans le *mapping* du placement physique de la valeur à la manière de ce qui est fait dans [Rakossy et al., 2013] pour les calculs. Pour cela il faudrait :
— un suivi des erreurs qui permet de savoir si une faute provient d'un registre ou bien était présente avant (*e.g.* non compatible avec l'implémentation à trois voteurs de la triplication architecturale de la figure 5.9c qui ne possède pas de voteur entre la partie opérative et la partie mémorisation) ;
— une intelligence réalisant la renumérotation dynamique des registres et capable de signaler qu'il n'y a plus assez de registres disponibles au processeur de reconfiguration et qu'il ne doit plus utiliser cette tuile.

Une évolution comme celle proposée dans [Abella et al., 2010] qui consiste à scinder les registres de la RF en plusieurs morceaux pour permettre de ne plus utiliser qu'une partie de mot est aussi envisageable.

L'inconvénient de ce genre de méthode est que l'intelligence de renumérotation et la table faisant la correspondance entre le numéro donné dans le *mappings* et le placement réel de la donnée sont susceptibles d'être affectées par des fautes transitoires et/ou permanentes. Lors de



(a) Architecture classique d'une tuile. (b) Tuile avec une RF de sortie.

Figure 5.15 – Modèle classique de tuile et amélioration par RF de sortie.

l'implémentation et des tests, il faudra vérifier si ce module supplémentaire censé supprimer une faiblesse n'en rajoute pas une plus importante.

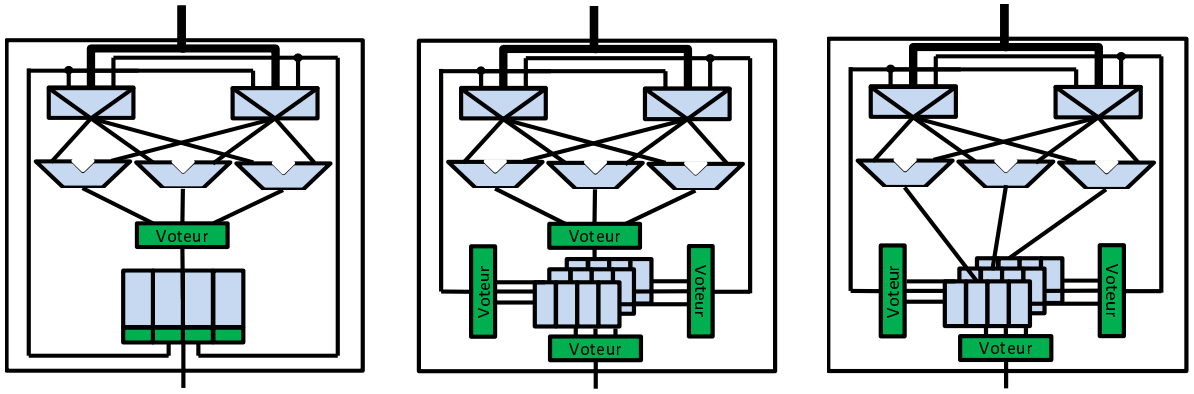
Une seconde idée pour limiter la faiblesse de la RF est de changer la différenciation entre deux configurations différentes au niveau du flot pour prendre en compte l'utilisation des registres. Ainsi, si deux *mappings* utilisent les mêmes tuiles et de la même façon le réseau d'interconnexion mais qu'ils n'utilisent pas les RFs de manières similaires alors les deux *mappings* seraient conservés. Cette solution semble cependant peu réaliste au vu du nombre de *mappings* déjà à stocker et à considérer par le module de reconfiguration.

5.5.2.2 Le registre de sortie

La seconde faiblesse, et certainement la plus critique, vient du registre de sortie. Considérons, le cas d'une tuile utilisant la triplication architecturale. Il y a donc en interne de cette tuile trois parties opératives, au moins un voteur, une RF protégée (par triplication ou par ECC) et un registre de sortie protégé. Si le registre de sortie est victime d'une faute permanente, alors la tuile ne sera plus capable de communiquer des résultats avec ses voisines. Elle ne sera donc plus utilisée malgré la quantité de ressources encore totalement fonctionnel en son sein. Pour ne plus être confronté à cette sensibilité, nous proposons une architecture n'utilisant pas de registre de sortie à proprement parler. Pour cela, au lieu d'avoir une RF interne et un registre de sortie séparé dans la logique de fonctionnement (comme dans la figure 5.15a), l'architecture de tuile proposée possède une RF de sortie. Cette RF de sortie, comme illustrée sur la figure 5.15b, est un RF possédant trois sorties différentes. Une seule de ces sorties permet de transmettre des données aux voisins via le réseau d'interconnexion. Les deux autres permettent uniquement d'alimenter l'opérateur en interne.

À partir de cette architecture, la figure 5.16 présente trois exemples d'implémentation de la triplication par méthode architecturale. En 5.16a, la tolérance est réalisée avec un voteur et la RF protégée par un code ECC. En 5.16b, ce sont quatre voteurs qui sont utilisés avec la RF tripliquée. En 5.16c, une version avec un voteur de moins est présentée.

La réalisation d'une RF possédant trois sorties est assez coûteuse en termes de surface. Le fait d'avoir un registre de moins dans la tuile ne sera pas nécessairement pénalisant car dans les codes analysés et utilisés avec le flot, il est très courant qu'une valeur aille à la fois dans le registre de sortie et dans la RF. Avec cette architecture, pour avoir la valeur à disposition en interne et la transmettre aux voisins, il suffit de l'avoir en un seul exemplaire dans la RF de sortie, réduisant aussi la probabilité d'apparition d'une faute. Ainsi, le registre qui communiquera des données aux voisins ne sera pas toujours le même. De plus, si le système décrit dans la sous-section précédente est mis en place, la faiblesse du registre de sortie disparaît complètement,



(a) Voteur et ECC. (b) 4 voteurs. (c) 3 voteurs.

FIGURE 5.16 – Exemples d'implémentation de la tolérance aux fautes transitoires pour l'architecture de tuile à RF de sortie.

remplacé par celle de ce système qui reste à évaluer. Comme précédemment, pour pouvoir mettre en place ce système, il faut pouvoir identifier que la faute se produise bien dans la RF et donc l'architecture proposé en figure 5.16c n'est pas envisageable.

5.6 Conclusion

Dans ce chapitre, plus prospectif que les précédents, nous avons proposé une nouvelle architecture CGRA ainsi qu'un modèle d'exécution associé permettant de gérer matériellement et de façon distribuée le flot de contrôle. Cette proposition, qui fait l'objet d'un dépôt de brevet, a été implémentée en VHDL, testée en simulation et synthétisée. Puis il a été montré comment rendre un CGRA basé sur ces tuiles tolérant aux fautes transitoires. En se basant sur cette tolérance, nous avons défini dans quel cas et comment il est possible de détecter les fautes permanentes, le tout en restant compatible avec l'architecture et le flot décrits précédemment.

Puis une première version d'un système complet intégrant des CGRAs tolérants aux fautes a été présentée. Il intègre plusieurs exemplaires de CGRAs tolérants aux fautes, avec un module de reconfiguration permettant de dynamiquement changer l'utilisation par l'application de l'architecture. L'architecture globale du système fait elle aussi l'objet d'un dépôt de brevet [Peyret et al., 2014e].

Il reste encore de nombreux points à explorer, tester et valider. Parmi ceux-ci, l'implémentation des différentes variantes des tolérances et des différents réseaux d'interconnexion (mesh-2D simple, mesh-2D torique, mesh-plus, mesh-X, autre...) permettra véritablement de choisir une méthode. De même, il faudra tester et valider l'architecture proposant une RF de sortie. De plus, il est possible d'imaginer un système possédant plus de deux voies de fonctionnement. Après basculement, la voie sera éteinte pour lui permettre de se régénérer, le tout augmentant ainsi la durée de vie globale de l'architecture.

Conclusion et perspectives

Conclusion des travaux

Les bénéfices qu'apporte l'électronique numérique sont jusqu'à présent entravés dans certains domaines par le fait qu'elle est sensible aux fautes, qui peuvent être par exemple provoquées par l'électromigration, la température ou les rayonnements neutronique et gamma. Ces fautes entraînent, par des processus électriques et physiques, des erreurs dans le fonctionnement et/ou dans les résultats fournis lorsque les composants ne sont pas protégés par des procédés de fabrication très coûteux. Ces erreurs peuvent avoir deux origines : les fautes transitoires et les fautes permanentes. L'étude de l'état de l'art des méthodes de tolérance de ces deux types de fautes sur les architectures électroniques nous a permis de choisir celles qu'il faut mettre en œuvre dans un contexte où l'architecture est susceptible d'être confrontée à ces fautes : la redondance multiple, éventuellement associée à des codes correcteurs d'erreurs, pour tolérer les fautes transitoires et la reconfiguration dynamique pour tolérer les fautes permanentes. Le type d'architecture le plus à même d'intégrer ces mécanismes est le CGRA, c'est-à-dire une architecture reconfigurable à gros grains. Cependant, ce type d'architecture n'a pas été initialement conçu pour exécuter un code applicatif complet. Or, pour limiter les points faibles de l'architecture par rapport à la tolérance aux fautes permanentes, il faut que tout le code applicatif s'exécute sur l'architecture.

Projeter une application sur une architecture CGRA est un problème NP-complet. Comme il n'existe pas dans la littérature de flot permettant de projeter une application en temps réel, il faut obtenir des solutions différentes en amont qui permettront la reconfiguration dynamique pendant le fonctionnement. Avoir un *mapping* pour chaque possibilité de fautes dans l'architecture n'étant pas réaliste, nous avons développé un flot dont l'objectif est d'obtenir un maximum de solutions différentes pour permettre de reconfigurer dynamiquement le CGRA rapidement en cas d'apparition de fautes. La première étape de la réalisation de ce flot a consisté à développer et mettre en œuvre une méthode permettant de projeter de façon multiple des codes sans flot de contrôle sur un modèle d'architecture CGRA. Ce flot, valorisé en conférences internationales, a été comparé à d'autres approches de l'état de l'art et permet d'obtenir des résultats similaires en termes de latence d'exécution de l'application et de taux de succès, tout en permettant une plus grande diversité de solutions. Ce flot de projection multiple a ensuite été étendu et adapté de manière à pouvoir gérer des codes complexes possédant un flot de contrôle. Pour pouvoir utiliser le très grand nombre de solutions générées par notre flot, nous avons décrit une méthode permettant en cas d'apparition d'une faute permanente de trouver une nouvelle configuration relativement rapidement. Les résultats de cette étude sont en cours de publication.

Les travaux de cette thèse se sont aussi focalisés sur la conception d'une architecture CGRA permettant d'exécuter des codes complets. Nous avons défini une architecture ainsi que le modèle d'exécution associé permettant de répondre à cette problématique. Nous avons commencé la réalisation d'un prototype en VHDL et validé son comportement en simulation. Les premières synthèses ont permis de vérifier qu'une telle approche était réaliste en termes de surface silicium et en fréquence de fonctionnement. Cette architecture et son modèle d'exécution font

actuellement l'objet d'un dépôt de brevet. Puis, nous avons analysé les différentes manières d'implémenter la tolérance aux fautes transitoires et la détection de fautes permanentes dans cette architecture en illustrant les avantages et les inconvénients de chacune d'elles. Enfin, nous avons proposé un système basé sur ce CGRA tolérant les fautes transitoires et permanentes en utilisant le module de reconfiguration défini avec le flot et un système de basculement de fonctionnement pour lequel un brevet est également en cours de dépôt.

Cette thèse formule donc une approche complète permettant de tolérer les fautes dans une architecture numérique. La solution proposée inclut l'architecture du système, son fonctionnement global et sa chaîne de compilation permettant de l'utiliser avec des codes applicatifs complexes décrits en langage de haut niveau.

Perspectives

Les travaux présentés dans cette thèse ont permis de lever une partie des verrous qui freinent l'utilisation d'architectures électroniques numériques dans des milieux susceptibles d'engendrer des fautes. Cette section présente les différentes voies ouvertes par les travaux de cette thèse. Elle commence par présenter les perspectives concernant le flot de conception permettant de projeter des applications sur des architectures CGRAs. Ensuite, les perspectives de travaux sur l'architecture de CGRA capable de gérer du contrôle sont décrites. Enfin, celles concernant le système global tolérant aux fautes sont détaillées.

Flot de conception

Le modèle d'architecture proposé dans cette thèse ne permet pas la représentation d'un bus de communication partagé à accès unique. Cette limitation actuelle peut être levée de deux manières : soit en complexifiant le modèle de l'architecture pour y intégrer des multiplexeurs d'accès, soit en modifiant le flot pour repérer et traiter différemment ce type de lien de communication. L'ajout de multiplexeur dans le modèle d'architecture impose de faire de même dans le modèle d'application. Le repérage dans le flot peut se faire de façon plus simple en annotant les arcs appartenant au même bus et en ne permettant l'utilisation que d'un seul arc de chaque annotation par cycle. Annoter les arcs présente l'avantage de ne pas modifier fondamentalement le modèle.

Nous avons fait plusieurs choix d'implémentation de fonction de décisions comme celles déterminant la priorité des nœuds ou le type de transformation de graphe à appliquer. Il faudrait tester les impacts de ces choix en implémentant d'autres fonctions et vérifier si les résultats ne sont pas meilleurs. En particulier tester l'influence de l'ordre de traitement entre les mémorisations et les opérations de calcul, entre les nœuds d'un même type mais possédant un nombre de successeurs différents et tester d'autres choix de transformation lorsque le choix est ouvert.

L'élagage possède une place importante dans le flot proposé. Quatre axes d'amélioration peuvent être explorés. Le premier est celui de l'implémentation d'une méthode d'élagage exacte optimale, c'est-à-dire ayant une complexité en $O(n \log_2(n))$ (contre $O(n^2)$ actuellement) et mesurer son impact sur le temps d'exécution. Le deuxième axe de développement concerne l'adaptation automatique des politiques d'élagage en fonction de l'avancement de la projection et du nombre de solutions partielles. Le troisième axe concerne l'élagage dans un arbre d'utilisation dont les tout premiers résultats sont prometteurs et devront faire l'objet d'études complémentaires. Le quatrième axe d'amélioration de l'élagage concerne l'élaboration d'une heuristique non aléatoire performante. Nous avons déjà identifié un ensemble de paramètres et de variables qui semblent pertinents à prendre en compte, mais dont l'assemblage reste à déterminer.

L'intégration d'un élagage aléatoire a parfois permis d'améliorer la latence du DFG correspondant. L'ordonnancement étant réalisé par une heuristique, il serait intéressant de tester

l'ajout d'aléatoire dans l'ordonnancement comme proposé dans [Trabelsi, 2008] pour de la synthèse de haut niveau (HLS). Une façon de faire serait par exemple de générer un nombre aléatoire pour chaque nœud ordonnançable et de n'ordonnancer le nœud à ce cycle que si le nombre dépasse un certain seuil.

L'extension du flot pour gérer les CDFGs ne s'est pas faite sans compromis. Nous avons décrit plusieurs variantes permettant de gérer les spécificités d'un CDFG, en particulier les variables partagées et les opérations PHI, mais nous n'avons testé qu'une variante pour chaque. Il faudrait valider expérimentalement ces choix en implémentant et testant les autres variantes.

Le dernier élément de la méthode concerne l'implémentation de la gestion des boucles pipelinées dans le flot. Cette gestion, comme précisé dans le chapitre 3, n'est pas incompatible avec le flot proposé, mais nécessite de nombreux aménagements et de re-développer une partie de la méthode.

Enfin, pour rendre utilisable le flot dans le cadre de projet, il faudrait ré-écrire le code dans un langage plus efficace que le Java qui permet de prototyper rapidement, mais ne permet pas de gestion fine de la mémoire et n'est pas connu pour ses performances calculatoires. Un langage tel que le C ou le C++ semble beaucoup plus à même de répondre aux besoins de performances de notre algorithme.

Dans le chapitre 5, nous avons défini des instructions de lecture en mémoire en deux parties (BeginLoad et EndLoad). Ceci permettrait de faire de la spéculation pour les accès mémoire un peu à la manière de la prédiction de branchement intégrée dans les processeurs généralistes. Autrement dit, de manière à diminuer l'attente incompressible entre le BeginLoad et le EndLoad, il serait intéressant d'investiguer la pertinence d'ordonnancer le BeginLoad avant même que l'on ne soit certain que l'on en ait vraiment besoin, c'est-à-dire dans le ou les BBs s'exécutant avant celui où la donnée est nécessaire.

Architecture CGRA pour CDFG

L'architecture de CGRA que nous avons proposée possède encore un certain nombre de points à affiner, tester et valider. Tout d'abord, il faudrait effectuer une exploration architecturale pour dimensionner l'architecture au mieux, en particulier le nombre de registres en RF, la présence de RFs partagées entre plusieurs tuiles et le type de réseau d'interconnexion. L'amélioration de l'architecture interne de la tuile en utilisant des RFs de sortie devra aussi être évaluée par rapport au flot car elle permet à une variable d'être stockée dans la tuile pendant un certain nombre de cycles avant d'être utilisée par les voisins sans avoir à bloquer l'opérateur pour cela.

Il faudra ensuite poursuivre les développements VHDL pour obtenir un prototype complet. Ceci implique d'avoir répondu à deux problématiques qui n'ont pu être abordées durant cette thèse. La première est la méthode de communication avec la mémoire centrale. La seconde est la méthode de chargement des mémoires programme de chaque tuile.

À plus long terme, grâce à l'approche complète développée dans cette thèse avec à la fois une architecture et un flot permettant d'y projeter des applications, ces travaux pourront être intégrés dans des plateformes type *manycore* hétérogènes où le parallélisme des applications pourra être utilisé pleinement sur le CGRA.

Système tolérant aux fautes

Les perspectives pour le système tolérant aux fautes se répartissent selon quatre axes principaux. Le premier axe concerne l'implémentation de la tolérance aux fautes dans l'architecture CGRA proposée. De très nombreuses variantes ont été analysées, mais malgré cette analyse, il n'a pas forcément été possible d'en choisir une à chaque fois. Concernant les registres, il est probable que la triplication corresponde mieux à nos besoins. Mais comme ni la surface réelle d'une tuile protégée, ni la surface silicium totale disponible pour le composant n'est connue, il ne faut pas exclure les codes correcteurs du fait de leur faible coût en surface. Il faudra aussi tester

et évaluer les différentes variantes de placement des voteurs dans l'architecture ainsi que le choix entre des tuiles tripliquées en interne ou bien la double duplication. La possibilité d'implémenter les évolutions décrites doit aussi rentrer en ligne de compte lors de ces tests. De même, il faudra étudier l'impact de la granularité du suivi des fautes.

Le deuxième axe concerne plus particulièrement le module de reconfiguration. Trois heuristiques ont été présentées et testées. Les expériences tendent à montrer que l'heuristique de surface minimale offre les meilleurs résultats. Mais il est possible d'implémenter cette politique différemment. En particulier, le choix qui a été fait est celui de ne changer que les *mappings* des BBs affectés par la faute. Il s'agit donc d'une heuristique cherchant un optimum local. Il faudrait tester la possibilité de chercher une nouvelle configuration qui minimise la surface de manière globale et observer ses performances. De même, il faudrait implémenter et évaluer la pertinence du précalcul en fonction des différentes heuristiques.

Le troisième axe concerne la complémentarité de l'approche avec d'autres méthodes de surveillance de fonctionnement. Nos travaux se sont surtout penchés sur vérification des chemins de données. Il faudrait pour obtenir un système complètement fiabilisé, que le flot de contrôle soit aussi vérifié en mettant en œuvre, par exemple, des moniteurs pour le changement de BB comme proposé dans [Ben Hammouda, 2014].

Le quatrième axe concerne le prototypage de ce système. Dans le système réel, la quantité de mémoire avec laquelle fonctionnera le module de reconfiguration sera limitée. Il faudra donc trouver un moyen efficace de représenter les *mappings* en mémoire. Nous envisageons de réaliser des tests réels sous irradiateur de manière à vérifier expérimentalement le système global. Pour cela, les tuiles du CGRA seront implémentées sur plusieurs petits FPGAs dont une partie sera sous le faisceau et donc subira des fautes transitoires. L'avantage de l'utilisation de FPGAs pour cette expérience est qu'une faute transitoire dans le *bitstream* est équivalente à une faute permanente. Ce sera aussi l'occasion de mettre en œuvre le mécanisme de basculement de fonctionnement dit « ping pong ». Après ces vérifications expérimentales, la réalisation d'un ASIC sera envisageable.

Ces perspectives doivent être explorées dans le cadre de deux thèses entre de CEA, le Lab-STICC et la DGA. La première thèse, traitera principalement des aspects architecturaux alors que la seconde se focalisera sur le flot de conception. Ces travaux seront complétés dans le cadre d'un projet prochainement soumis en réponse à l'appel ANR-ASTRID. Enfin l'intégration du CGRA dans une architecture *manycore* et les problématiques d'accès à la mémoire seront étudiées dans le cadre de la thèse de S. DAS sous la codirection de Ph. COUSSY et de L. BENINI.

Annexe A

Conventions

A.1 Glossaire

Pour lever toute ambiguïté, dans ce manuscrit, les termes suivants ont les sens définis ici :

— **Projection** (ou processus de projection) : Mise en correspondance du graphe de l'application avec celui de l'architecture. Ce processus génère des *mappings*. Pour ne pas alourdir le texte, le terme de « multi-projection » ne sera employé que si nécessaire.
Pour signifier un autre sens, un qualificatif est ajouté, par exemple « projection mathématique » ou « projection géométrique ».

— ***Mapping*** : Nom donné à une correspondance particulière entre le graphe d'une application et le graphe de l'architecture. Un *mapping* est une solution trouvée la projection.

— **Configuration** : Une configuration correspond à une certaine utilisation de l'architecture définie par un *mapping*.

— **Réseau** : Un réseau, tel qu'il est utilisé dans le chapitre 2 et dans l'annexe B, est un sous-ensemble de tuiles de l'architecture relié par une sous partie de l'interconnexion. Il est donc constitué d'un seul graphe. Il ne faut pas le confondre avec le terme « réseau d'interconnexion » qui désigne uniquement la manière dont sont reliées les tuiles.

— Le **temps de compilation** est utilisé pour définir le temps que met la méthode de projection pour s'exécuter.

A.2 Liste des acronymes

ALU Unité Arithmétique et Logique . . . 18
ALAP *As Late As Possible* . . . 46
ASAP *As Soon As Possible* . . . 46
BB *Basic Block* . . . 35
BIST *Built-In Self-Test* . . . 13
CAS *Column Address Strobe* . . . 47
CDFG *Control and Data Flow Graph* . . . 158
CFG *Control Flow Graph* . . . 35
CGRA Architecture Reconfigurable à Gros Grains . . . 158
CMOS *Complementary Metal Oxide Semiconductor* . . . 4
DCT *Discrete Cosine Transform* . . . 66
DEU *Double Event Upset* . . . 12
DFG *Data Flow Graph* . . . 20
DRAM *Dynamic Random Access Memory* . . . 47
DMA composant d'Accès Direct à la Mémoire . . . 24

DWC *Duplication With Comparison* . . . 9
ECC Code Correcteur d'Erreurs . . . 10
EDAC *Error Detection And Correction* . . . 15
EDM *Error Detection Module* . . . 29
EGRA *Expression-Grained Reconfigurable Architecture* . . . 18
EMA *Exponential Moving Average* . . . 66
EMF *Eclipse Modeling Framework* . . . 66
FEHM *Flexible Error Handling Module* . . . 29
FFT *Fast Fourier Transform* . . . 30
FIFO mémoire de type *First In First Out* . . . 114
FPGA Architecture Reconfigurable à Grain Fin . . . 4
IDCT Transformée en Cosinus Discrète Inverse . . . 94
ITAR *International Traffic in Arms Regulation* . . . 1
ID identifiant . . . 37
II Intervalle d'initiation . . . 20
ILP *Integer Linear Programming* . . . 23
LSU *Load-Store Unit* . . . 112
LUT *Look-Up Table* . . . 5
MCS *Maximum Common Subgraph* . . . 55
MBU *Multiple Bit Upset* . . . 12
MDFG *Modulo Data Flow Graph* . . . 21
MOS *Metal Oxide Semiconductor* . . . 6
MWD *Moving Window Deconvolution* . . . 66
NOC *Network On Chip* . . . 53
PFF Polyomino à Forme Fixée . . . 41
PFL Polyomino à Forme Libre . . . 41
RF File de Registres . . . 12
RISC *Reduced Instruction-Set Computer* . . . 24
RDP *Reconfigurable Data Path* . . . 29
RoHS *Restriction of the use of certain Hazardous Substances in electrical and electronic equipment* . . . 8
SEFI *Single Event Functionnal Interrupts* . . . 4
SEL *Single Event Latch-up* . . . 5
SET *Single Event Transient* . . . 4
SEU *Single Event Upset* . . . 4
SOI *Silicon On Insulator* . . . 7
SOS *Silicon On Sapphire* . . . 7
SSA *Static Single Assignement form* . . . 37
TDMA *Time Division Multiple Access bus* . . . 53
TEC *Time-Extended CGRA* . . . 21
TEU *Triple Event Upset* . . . 12
TID *Total Ionizing Dose* . . . 6
TMR *Triple Modular Redondancy* . . . 10
VHDL *Very high speed integrated circuit High Definition Language* . . . 120
VLIW *Very Long Instruction Word* . . . 25

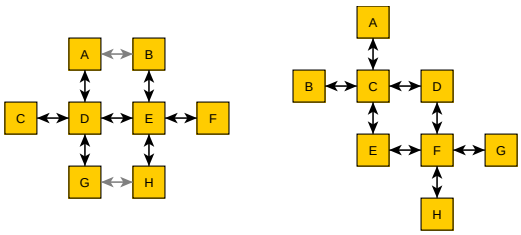
Annexe B

Illustration des réseaux à huit tuiles

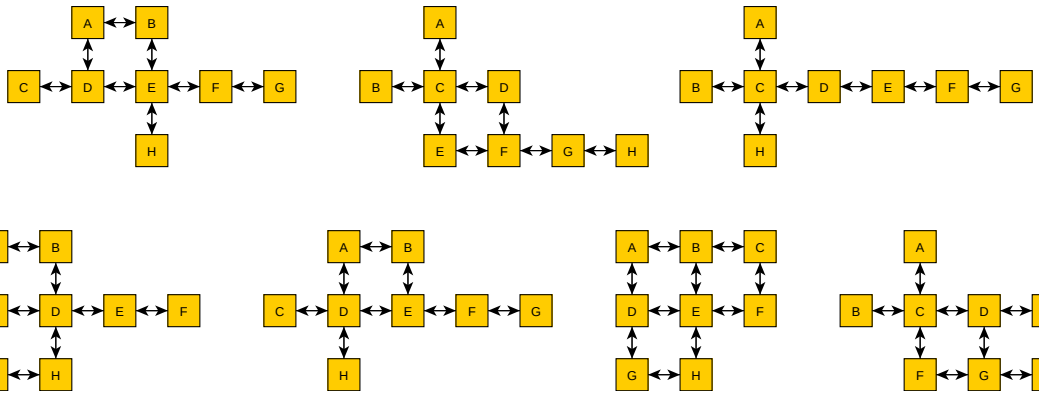
Dans cette annexe, les schémas des 36 réseaux différents à huit tuiles sur un CGRA possédant un mesh-2D simple sont présentés. Ils sont classés en fonction du nombre de voisins maximal puis en fonction du nombre de tuiles autorisées à utiliser ce nombre maximal, illustrant la pertinence de ces deux paramètres.

À 4 voisins maximum

2 tuiles à nombre de voisins maximum :

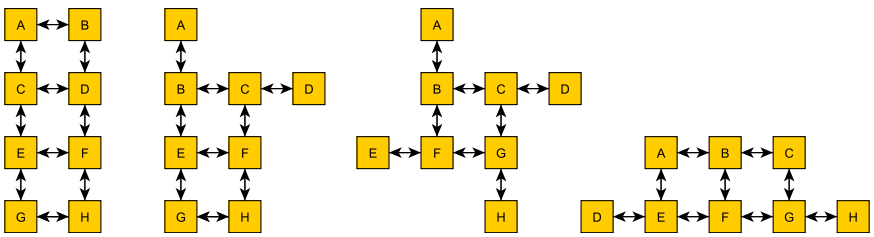


1 tuile à nombre de voisins maximum :

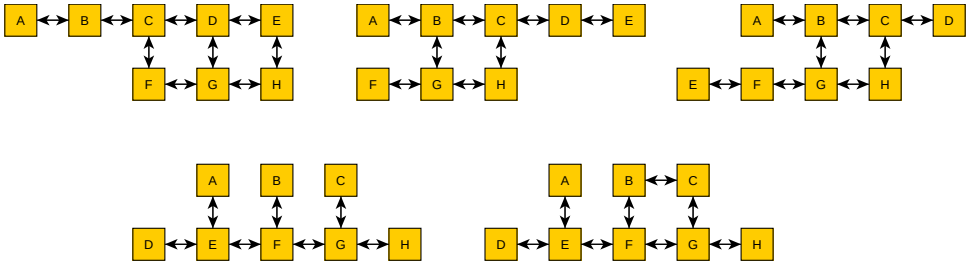


À 3 voisins maximum

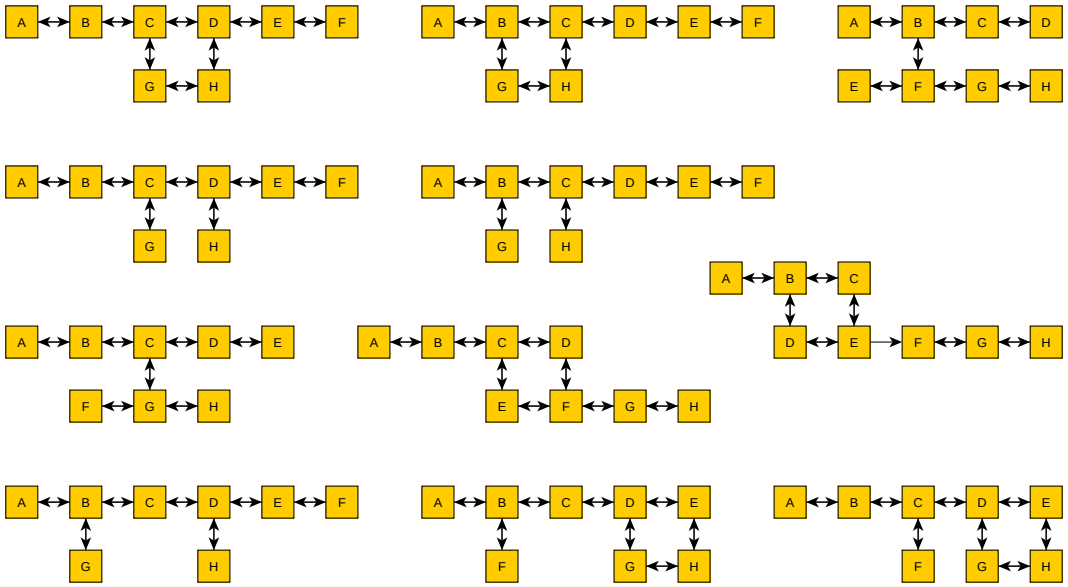
4 tuiles à nombre de voisins maximum :



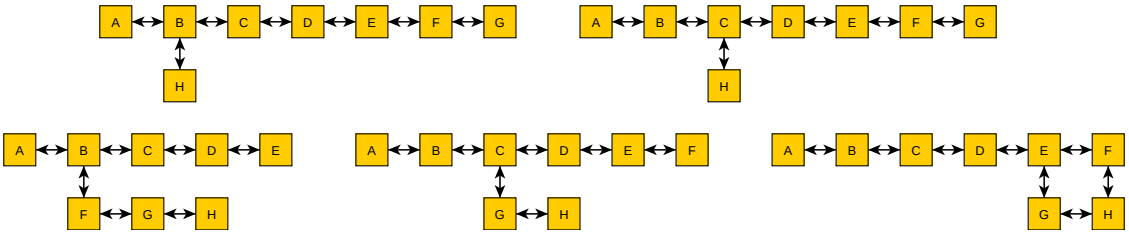
3 tuiles à nombre de voisins maximum :



2 tuiles à nombre de voisins maximum :

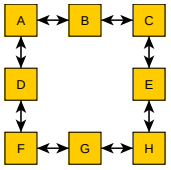


1 tuile à nombre de voisins maximum :

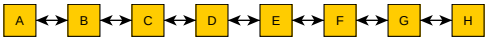


À 2 voisins maximum

8 tuiles à nombre de voisins maximum :



6 tuiles à nombre de voisins maximum :



Annexe C

Bibliographie personnelle

Brevets

- [Peyret et al., 2014d] : Peyret, T., Thevenin, M., Corre, G., Martin, K., Coussy, P. (2014). Architecture de tuile de calcul et modèle d'exécution associé pour Architecture Reconfigurable à Gros Grains (CGRAs) pour des codes possédant un flot de contrôle. *Dépôt en cours.*

- [Peyret et al., 2014e] : Peyret, T., Thevenin, M., Corre, G., Martin, K., Coussy, P. (2014). Tolérance aux fautes permanentes sans interruption de service par reconfiguration dynamique de composants électroniques. *Dépôt en cours.*

- [Moline et al., 2014] : Moline, Y., Thevenin, M., Corre, G., Peyret, T. (2014). Procédé et système d'extraction dynamique d'impulsions dans un signal temporel bruité. *Brevet numéro FR14 50568.*

Conférences internationales avec actes

- [Peyret et al., 2014b] : Peyret, T., Corre, G., Thevenin, M., Martin, K., Coussy, P. (2014). Efficient application mapping on CGRAs based on backward simultaneous scheduling/ binding and dynamic graph transformations. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 169−172.

- [Peyret et al., 2014a] : Peyret, T., Corre, G., Thevenin, M., Martin, K., Coussy, P. (2014). An Automated Design Approach to Map Applications on CGRAs. In *Great Lakes Symposium on VLSI (GLSVLSI'14).*

Conférences nationales ou francophones avec actes

- [Peyret et al., 2014c] : Peyret, T., Corre, G., Thevenin, M., Martin, K., Coussy, P. (2014). Ordonnancement, assignation et transformations dynamiques de graphe simultanés pour projeter efficacement des applications sur CGRAs. In *Conférence en Parallélisme, Architecture et Système (ComPAS'2014).*

Colloques nationaux

- Présentation d'un poster au Colloque national du GDR SOC-SIP du 11-13 juin 2014.

Bibliographie

[Abd-El-Barr, 2007] Abd-El-Barr, M. (2007). *Design and analysis of reliable and fault-tolerant computer systems*. World Scientific Pub Co Inc. Available from : `http://www.lavoisier.fr/livre/notice.asp?id=RSXWO6AKXXSOWW`. (Page 7)

[Abella et al., 2010] Abella, J., Carretero, J., Chaparro, P., and Vera, X. (2010). The split register file. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pages 6–9. Available from : `http://dl.acm.org/citation.cfm?id=1871156`. (Pages 15, 130 et 134)

[Alexander et al., 2008] Alexander, D. R., Hunt, K., Owens, M., and Lyke, J. (2008). Affordable Rad-Hard–An Impossible Dream ? In *22nd Annual AIAA/USU Conference on Small Satellites*, pages 1–5. Available from : `http://digitalcommons.usu.edu/smallsat/2008/all2008/73/`. (Page 1)

[Aliee and Zarandi, 2011] Aliee, H. and Zarandi, H. R. (2011). A Fault-tolerant, Dynamically Scheduled Pipeline Structure for Chip Multiprocessors. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'11)*, pages 324–337, Naples, Italy. Springer-Verlag. Available from : `http://dl.acm.org/citation.cfm?id=2041619.2041652`. (Pages 13 et 129)

[Ansaloni et al., 2008] Ansaloni, G., Bonzini, P., and Pozzi, L. (2008). Design and architectural exploration of expression-grained reconfigurable arrays. *Application Specific Processors, 2008. SASP 2008. Symposium on*. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4570782`. (Page 18)

[Azeem et al., 2011] Azeem, M. M., Piestrak, S. J., Sentieys, O., and Pillement, S. (2011). Error recovery technique for coarse-grained reconfigurable architectures. In *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 2–7. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5783133`. (Page 29)

[Baloch et al., 2006] Baloch, S., Arslan, T., and Stoica, A. (2006). Design of a Novel Soft Error Mitigation Technique for Reconfigurable Architectures. In *2006 IEEE Aerospace Conference*, pages 1–9. IEEE. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1655970`. (Pages 12 et 14)

[Ben Hammouda, 2014] Ben Hammouda, M. (2014). *High Level Synthesis for Circuit Generation that Supports Software-like Debugging*. PhD thesis, Université de Bretagne Occidentale (UBO). (Page 140)

[Berg et al., 2008] Berg, M., Poivey, C., Petrick, D., Espinosa, D. C. R., Lesea, A., LaBel, K. A., Friendlich, M. R., Kim, H. S., and Phan, A. (2008). Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA : Design, Test, and Analysis. *IEEE Transactions on Nuclear Science*, 55(4) :2259–2266. Available from : `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4636940`. (Pages 15 et 37)

[Blaauw et al., 2008] Blaauw, D., Kalaiselvan, S., Lai, K., Ma, W.-H., Pant, S., Tokunaga, C., Das, S., and Bull, D. (2008). Razor II : In situ error detection and correction for PVT and SER tolerance. In *IEEE International Solid-State Circuits Conference (ISSCC 2008)*, pages

400–402. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4523226. (Page 8)

[Bolchini et al., 2011] Bolchini, C., Miele, A., and Sandionigi, C. (2011). A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs. *IEEE Transactions on Computers*, 60(12) :1744–1758. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5674027. (Pages 1, 10, 13 et 37)

[Brenner et al., 2006] Brenner, J. A., van der Veen, J. C., Fekete, S. P., Oliveira Filho, J., and Rosenstiel, W. (2006). Optimal Simultaneous Scheduling, Binding and Routing for Processor-like Reconfigurable Architectures. In *Field Programmable Logic and Applications, International Conference on*. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4101024. (Page 23)

[Bridgford et al., 2008] Bridgford, B., Carmichael, C., and Tseng, C. W. (2008). Single-event upset mitigation selection guide. *Xilinx Application Note : XAPP987*, pages 1–7. Available from : http://application-notes.digchip.com/077/77-43115.pdf. (Page 4)

[Campi et al., 2007] Campi, F., Deledda, A., Mucci, C., Lodi, A., Pizzotti, M., Cirrarelli, L., Rolandi, P., Vitkovski, A., and Vanzolini, L. (2007). A dynamically adaptive DSP for heterogeneous reconfigurable platforms. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4211764. (Page 18)

[Carmichael, 2006] Carmichael, C. X. (2006). Application Note : Triple Module Redundancy Design Techniques for Virtex FPGAs. Technical report, Xilinx. Available from : http://www.xilinx.com/support/documentation/application_notes/xapp197.pdf. (Pages 1, 5, 10, 11 et 121)

[Chambre des Représentants Belgique, 2003] Chambre des Représentants Belgique (2003). Compte Rendu Intégral, CRIV 51. Technical report, Commission de l'Intérieur, des Affaires générales et de la Fonction publique, Belgique. Available from : http://www.lachambre.be/doc/CCRI/pdf/51/ic007.pdf. (Page 3)

[Cheatham et al., 2006] Cheatham, J. A., Emmert, J. M., and Baumgart, S. (2006). A survey of fault tolerant methodologies for FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 11(2) :501–533. Available from : http://portal.acm.org/citation.cfm?doid=1142155.1142167. (Page 13)

[Chen and Chien, 2008] Chen, J. C. and Chien, S.-Y. (2008). CRISP : Coarse-grained reconfigurable image stream processor for digital still cameras and camcorders. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(9) :1223–1236. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4579684. (Page 18)

[Chen and Mitra, 2014] Chen, L. and Mitra, T. (2014). Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems*, V(January) :1–22. Available from : https://dl.comp.nus.edu.sg/jspui/handle/1900.100/4289. (Page 20)

[D'Angelo et al., 1998] D'Angelo, S., Metra, C., Pastore, S., Pogutz, A., and Sechi, G. R. (1998). Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems. In *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on*, pages 233–240. IEEE. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=732171. (Pages 1 et 10)

[David et al., 2002a] David, R., Chillet, D., Pillement, S., and Sentieys, O. (2002a). A compilation framework for a dynamically reconfigurable architecture. In Glesner, M., Zipf, P., and Renovell, M., editors, *Field-Programmable Logic and Applications : Reconfigurable Computing Is Going Mainstream*, volume 2438/2002, pages 1058–1067. Springer Berlin Heidelberg. Available from : http://www.springerlink.com/index/0M2GT1H9JRE6CT8C.pdf. (Pages 18 et 24)

[David et al., 2002b] David, R., Chillet, D., Pillement, S., and Sentieys, O. (2002b). DART : a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. In *Parallel and Distributed Processing Symposium*, pages 2–9. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1016554. (Page 24)

[David et al., 2002c] David, R., Chillet, D., Pillement, S., and Sentieys, O. (2002c). Mapping future generation mobile telecommunication applications on a dynamically reconfigurable architecture. In *27th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Available from : http://r2d2.enssat.fr/bindocs/publications/dav-ICA02.pdf. (Page 24)

[De Sutter et al., 2008] De Sutter, B., Coene, P., Vander Aa, T., and Mei, B. (2008). Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. *ACM SIGPLAN Notices*, 43(7) :151. Available from : http://portal.acm.org/citation.cfm?doid=1379023.1375678. (Pages 20, 21, 23, 25 et 66)

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8). Available from : http://dl.acm.org/citation.cfm?id=360975. (Page 110)

[Ebeling et al., 1996] Ebeling, C., Cronquist, D. C., and Franklin, P. (1996). RaPiD - Reconfigurable Pipelined Datapath. *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 126–135. Available from : http://www.springerlink.com/index/f45122367533h852.pdf. (Page 18)

[Eisenhardt et al., 2011] Eisenhardt, S., Küster, A., Schweizer, T., Kuhn, T., and Rosenstiel, W. (2011). Spatial and Temporal Data Path Remapping for Fault-Tolerant Coarse-Grained Reconfigurable Architectures. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2011 IEEE International Symposium on*, pages 382–388. Ieee. Available from : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6104466. (Pages 29, 30, 31 et 37)

[Esmaeeli et al., 2011] Esmaeeli, S., Hosseini, M., Vahdat, B. V., and Rashidian, B. (2011). A multi-bit error tolerant register file for a high reliable embedded processor. In *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pages 532–537. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6122330. (Pages 13, 14, 121 et 128)

[Fazeli and Miremadi, 2008] Fazeli, M. and Miremadi, S. G. (2008). A Power Efficient Masking Technique for Design of Robust Embedded Systems against SEUs and SETs. In *Defect and Fault Tolerance of VLSI Systems, 2008 IEEE International Symposium on*, pages 193–201. Ieee. Available from : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4641173. (Pages 8 et 9)

[Friedman et al., 2009] Friedman, S., Carroll, A., Van Essen, B., Ebeling, C., Hauck, S., and Ylvisaker, B. (2009). SPR : an architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 191–200. Available from : http://dl.acm.org/citation.cfm?id=1508158. (Pages 20, 21 et 23)

[Goldstein et al., 2000] Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R. (2000). PipeRench : A reconfigurable architecture and compiler. *Computer*. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=839324. (Page 18)

[Golomb, 1996] Golomb, S. (1996). *Polyominoes : puzzles, patterns, problems, and packings*. Princeton Science Library. Available from : http://books.google.fr/books?isbn=0691024448. (Page 41)

[Grassi, 2008] Grassi, T. (2008). About using FPGAs in radiation environments. In *Hadron Calorimeter Working Group of the CMS Upgrade Workshop*. Available from : https://indico.cern.ch/event/44296/contribution/43/material/slides/0.pdf. (Page 6)

[Guilhemsang, 2011] Guilhemsang, J. (2011). *Test en ligne pour la détection des fautes intermittentes dans les architectures multiprocesseurs embarquées*. PhD thesis, Université de Nice-Sophia Antipolis. Available from : http://tel.archives-ouvertes.fr/tel-00640599. (Page 7)

[Guttmann, 2009] Guttmann, A. J. (2009). *Polygons, polyominoes and polycubes*, volume 775. Springer. Available from : http://books.google.fr/books?isbn=1402099266. (Page 42)

[Hamzeh et al., 2012] Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2012). EPIMap : using epimorphism to map applications on CGRAs. In *Design Automation Conference*, pages 1284–1291. Available from : http://www.public.asu.edu/~mhamzeh1/data/EPIM.pdf. (Pages 20, 21, 22, 23, 47, 52, 54, 55, 56, 58, 59, 66 et 96)

[Hamzeh et al., 2013] Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2013). REGIMap : register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Design Automation Conference*. Available from : http://www.public.asu.edu/~mhamzeh1/data/REGIMap.pdf. (Pages 20, 21, 23, 58, 59, 66 et 96)

[Hartenstein and Kress, 1995] Hartenstein, R. and Kress, R. (1995). A datapath synthesis system for the reconfigurable datapath architecture. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*, pages 479–484. Nihon Gakkai Jimu Senta. Available from : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=486359. (Pages 18 et 52)

[Hartenstein, 2001] Hartenstein, R. W. (2001). A decade of reconfigurable computing : a visionary retrospective. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 642–649. Available from : http://dl.acm.org/citation.cfm?id=367839. (Page 17)

[Haseloff, 2000] Haseloff, E. (2000). Latch-Up, ESD, and Other Phenomena. Technical report, Texas Instruments. Available from : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.6786. (Page 6)

[Heysters and Smit, 2003] Heysters, P. M. and Smit, G. J. (2003). Mapping of DSP algorithms on the MONTIUM architecture. *Parallel and Distributed Processing*. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1213333. (Page 18)

[Huang and Mccluskey, 2001] Huang, W.-j. and Mccluskey, E. J. (2001). Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance. *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*. Available from : http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1420910&isnumber=30703. (Pages 15 et 30)

[Jafri et al., 2010] Jafri, S. M. A. H., Piestrak, S. J., Sentieys, O., and Pillement, S. (2010). Design of a Fault-Tolerant Coarse-Grained Reconfigurable Architecture : A Case Study. In *Quality Electronic Design (ISQED), 2010 11th International Symposium on*. Available from : http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5450481. (Page 28)

[Jafri et al., 2013] Jafri, S. M. A. H., Piestrak, S. J., Sentieys, O., and Pillement, S. (2013). Design of the Coarse-Grained Reconfigurable Architecture DART with On-Line Error Detection. *Microprocessors and Microsystems*. Available from : http://linkinghub.elsevier.com/retrieve/pii/S0141933113002032. (Page 28)

[Jensen and Guttmann, 2000] Jensen, I. and Guttmann, A. J. (2000). Statistics of lattice animals (polyominoes) and polygons. *Journal of Physics A : Mathematical and General*, 33 :257–263. Available from : http://iopscience.iop.org/0305-4470/33/29/102. (Page 42)

[Johnson and Wirthlin, 2010] Johnson, J. and Wirthlin, M. (2010). Voter insertion algorithms for FPGA designs using triple modular redundancy. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 249–258. ACM. Available from : http://dl.acm.org/citation.cfm?id=1723154. (Page 10)

[Khawam et al., 2008] Khawam, S., Nousias, I., and Milward, M. (2008). The reconfigurable instruction cell array. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1) :75–85. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4407539. (Page 18)

[Krishnamohan and Mahapatra, 2005] Krishnamohan, S. and Mahapatra, N. R. (2005). Analysis and design of soft-error hardened latches. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI - GLSVSLI '05*, page 328, New York, New York, USA. ACM Press. Available from : http://portal.acm.org/citation.cfm?doid=1057661.1057740. (Page 8)

[Lanuzza et al., 2007] Lanuzza, M., Perri, S., Corsonello, P., and Margala, M. (2007). A new reconfigurable coarse-grain architecture for multimedia applications. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4291909. (Page 18)

[Lee et al., 2011] Lee, G., Choi, K., and Dutt, N. D. (2011). Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *Computer-Aided Design of Intergrated Circuits and Systems, IEEE Transactions on*, 30(5) :637–650. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5752434. (Pages 20, 23 et 66)

[Levi, 1973] Levi, G. (1973). A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4) :341–352. Available from : http://www.springerlink.com/index/10.1007/BF02575586. (Pages 23, 55 et 58)

[Lima et al., 2003a] Lima, F., Carro, L., and Reis, R. (2003a). Designing fault tolerant systems into SRAM-based FPGAs. In *Design Automation Conference, 2003. Proceedings*, pages 650–655. IEEE. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1219099. (Pages 10 et 12)

[Lima et al., 2003b] Lima, F., Carro, L., and Reis, R. (2003b). Reducing pin and area overhead in fault-tolerant FPGA-based designs. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 108–117. ACM. Available from : http://dl.acm.org/citation.cfm?id=611834. (Pages 10 et 12)

[MacWilliams and Sloane, N. J., 1977] MacWilliams, F. J. and Sloane, N. J., A. (1977). *The Theory of Error-Correcting Codes*. North-Holland Publishing Company. Available from : http://books.google.fr/books?isbn=0444850104http://books.google.fr/books?isbn=0444850090. (Page 10)

[Marshall et al., 1999] Marshall, A., Stansfield, T., Kostarnov, I., Vuillemin, J., and Hutchings, B. (1999). A reconfigurable arithmetic array for multimedia applications. *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143. Available from : http://dl.acm.org/citation.cfm?id=296444. (Page 18)

[Mccluskey, 1985] Mccluskey, E. J. (1985). Built-In Self-Test Structures. *IEEE Design & Test of Computers*, 2(2) :29–36. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4069539. (Page 13)

[Mei et al., 2005] Mei, B., Lambrechts, A., Verkest, D., Mignolet, J.-Y., and Lauwereins, R. (2005). Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design and Test of Computers*, 22(2) :90–101. Available from : http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1413142. (Pages 19 et 25)

[Mei et al., 2002] Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2002). DRESC : A retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002. (FPT). IEEE International Conference on*, pages 166–173. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1188678. (Pages 20, 21, 23, 25, 59 et 66)

[Mei et al., 2003a] Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003a). ADRES : An architecture with tightly coupled VLIW processor and coarse-grained recon-

figurable matrix. In Y. K. Cheung, P. and Constantinides, G., editors, *Field Programmable Logic and Application*, pages 61–70. Springer Berlin / Heidelberg. Available from : http://www.springerlink.com/index/03YT3XEH60R8971K.pdf. (Pages 18, 25 et 26)

[Mei et al., 2003b] Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003b). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *DATE '03 Proceedings of the conference on Design, Automation and Test in Europe*, 1 :296 – 301. Available from : http://link.aip.org/link/ICDTEA/v150/i5/p255/s1&Agg=doi. (Page 25)

[Miyamori and Olukotun, 1998] Miyamori, T. and Olukotun, K. (1998). REMARC : Reconfigurable multimedia array coprocessor. *IEICE Transactions on Information and Systems E82-D*. Available from : http://meseec.ce.rit.edu/eecc722-fall2006/papers/reconfigurable-computing/5/FPGA98.pdf. (Page 18)

[Moline et al., 2014] Moline, Y., Thevenin, M., Corre, G., and Peyret, T. (2014). Procédé et système d'extraction dynamique d'impulsions dans un signal temporel bruité. *Brevet numéro : FR14 50568*. (Page 145)

[Naseer et al., 2006] Naseer, R., Bhatti, R. Z., and Draper, J. (2006). Analysis of soft error mitigation techniques for register files in IBM Cu-08 90nm technology. In *Circuits and Systems, 2006. 49th IEEE International Midwest Symposium on* ,, pages 515–519. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4267189. (Pages 12, 14 et 121)

[Neumann, 1956] Neumann, J. V. (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata studies*, pages 329–378. Available from : http://www.archtypic.com/wiki/images/a/af/Von_Neumann_Probabilistic_Logics_and_the_Synthesis_of_Reliable_Organisms_from_Unreliable_Components.pdf. (Pages 10 et 122)

[Nicolaidis, 2000] Nicolaidis, M. (2000). *Les Limites technologiques du silicium et tolerance aux fautes*. PhD thesis, Institu National Polytechnique de Grenoble. Available from : http://tel.archives-ouvertes.fr/tel-00002907. (Page 4)

[Normand, 1996] Normand, E. (1996). Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6) :2742–2750. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=556861. (Page 3)

[Oliveira e Silva, 2007] Oliveira e Silva, T. (2007). Animal enumerations on the {4,4} Euclidean tiling. Available from : http://sweet.ua.pt/tos/animals/a44.html. (Page 41)

[Pani and Raffo, 2006] Pani, D. and Raffo, L. (2006). Stigmergic approaches applied to flexible fault-tolerant digital VLSI architectures. *Journal of Parallel and Distributed Computing*, 66(8) :1014–1024. Available from : http://linkinghub.elsevier.com/retrieve/pii/S0743731505002467. (Page 16)

[Parashar et al., 2013] Parashar, A., Pellauer, M., Adler, M., Ahsan, B., Crago, N., Lustig, D., Pavlov, V., Zhai, A., Gambhir, M., Jaleel, A., Allmon, R., Rayess, R., Maresh, S., and Emer, J. (2013). Triggered instructions : A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA'13*, pages 142–153. Available from : http://dl.acm.org/citation.cfm?id=2485935. (Pages 26, 27, 28, 47 et 115)

[Park et al., 2008] Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-S. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. Available from : http://dl.acm.org/citation.cfm?id=1454140. (Pages 18, 20, 21 et 23)

[Park et al., 2009] Park, H., Park, Y., and Mahlke, S. A. (2009). Polymorphic pipeline array : a flexible multicore accelerator with virtualized execution for mobile multimedia applications.

MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. Available from : http://dl.acm.org/citation.cfm?id=1669160. (Page 18)

[Paulin and Knight, 1989] Paulin, P. G. and Knight, J. P. (1989). Force-directed scheduling for the behavioral synthesis of ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6) :661–679. Available from : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=31522. (Page 57)

[Peterson and Weldon, 1972] Peterson, W. W. and Weldon, E. J. (1972). *Error-correcting codes*. The MIT Press. Available from : http://mitpress.mit.edu/books/error-correcting-codes. (Page 10)

[Peyret et al., 2014a] Peyret, T., Corre, G., Thevenin, M., Martin, K., and Coussy, P. (2014a). An automated design approach to map applications on CGRAs. In *Great Lakes Symposium on VLSI (GLSVLSI'14)*. Available from : http://dl.acm.org/citation.cfm?id=2591552. (Pages 83 et 145)

[Peyret et al., 2014b] Peyret, T., Corre, G., Thevenin, M., Martin, K., and Coussy, P. (2014b). Efficient application mapping on CGRAs based on backward simultaneous scheduling/-binding and dynamic graph transformations. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 169–172. IEEE. Available from : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6868652. (Pages 83 et 145)

[Peyret et al., 2014c] Peyret, T., Corre, G., Thevenin, M., Martin, K., and Coussy, P. (2014c). Ordonnancement, assignation et transformations dynamiques de graphe simultanés pour projeter efficacement des applications sur CGRAs. In *Conférence en Parallélisme, Architecture et Système (ComPAS'2014)*, Neuchatel. Available from : http://hal.archives-ouvertes.fr/hal-00985815. (Pages 83 et 145)

[Peyret et al., 2014d] Peyret, T., Thevenin, M., Corre, G., Martin, K., and Coussy, P. (2014d). Architecture de tuile de calcul et modèle d'exécution associé pour Architecture Reconfigurable à Gros Grains (CGRAs) pour des codes possédant un flot de contrôle. *Dépôt en cours*. (Pages 109, 117 et 145)

[Peyret et al., 2014e] Peyret, T., Thevenin, M., Corre, G., Martin, K., and Coussy, P. (2014e). Tolérance aux fautes permanentes sans interruption de service par reconfiguration dynamique de composants électroniques. *Dépôt en cours*. (Pages 136 et 145)

[Pillement et al., 2008] Pillement, S., Sentieys, O., and David, R. (2008). DART : A Functional-Level Reconfigurable Architecture for High Energy Efficiency. *EURASIP Journal on Embedded Systems*, 2008 :1–13. Available from : http://www.hindawi.com/journals/es/2008/562326/. (Pages 24 et 25)

[Pnevmatikatos, 1996] Pnevmatikatos, D. N. (1996). *Incorporating Guarded Execution into Existing Instruction Sets*. PhD thesis, University of Wisconsin - Madison. Available from : ftp://ftp.cs.wisc.edu/sohi/theses/pnevmati.pdf. (Page 110)

[Quinn et al., 2008] Quinn, H., Graham, P., Morgan, K., Krone, J., Caffrey, M., and Wirthlin, M. (2008). An introduction to radiation-induced failure modes and related mitigation methods for Xilinx SRAM FPGAs. *Engineering of Reconfigurable Systems and Algorithms (ERSA), International Conference on*, 836. Available from : ftp://ftp.lanl.gov/public/.snapshot/hourly.2/hquinn/quinn_intro_to_rad.pdf. (Pages 4 et 5)

[Raffin et al., 2010] Raffin, E., Wolinski, C., Charot, F., Kuchcinski, K., Guyetant, S., Chevobbe, S., and Casseau, E. (2010). Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture. In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 168–175. Available from : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5706261. (Pages 23 et 51)

[Rakossy et al., 2013] Rakossy, Z. E., Hiromoto, M., Tsutsui, H., Sato, T., Nakamura, Y., and Ochi, H. (2013). Hot-Swapping Architecture with Back-biased Testing for Mitigation of Permanent Faults in Functional Unit Array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 535–540, New Jersey. IEEE Conference Publications. Available from : `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513566`. (Pages 15, 19, 30, 32, 127, 129 et 134)

[Rakossy et al., 2014] Rakossy, Z. E., Merchant, F., Acosta-Aponte, A., Nandy, S. K., and Chattopadhyay, A. (2014). Scalable and Energy-Efficient Reconfigurable Accelerator for Column-wise Givens Rotation. In *22nd IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. (Page 117)

[Rakossy et al., 2012] Rakossy, Z. E., Naphade, T., and Chattopadhyay, A. (2012). Design and analysis of layered coarse-grained reconfigurable architecture. *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6. Available from : `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6416736`. (Page 117)

[Rau, 1994] Rau, B. R. (1994). Iterative modulo scheduling : An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. Available from : `http://dl.acm.org/citation.cfm?id=192731`. (Pages 21, 23 et 96)

[Redmond, 2001] Redmond, C. (2001). Winning the battle against latch-up in CMOS analog switches. *Analog Dialogue*, 05 :5–7. Available from : `http://files.tomek.cedro.info/electronics/doc/space/latchup/latchup.pdf`. (Pages 1, 7 et 8)

[Roosta, 2004] Roosta, R. (2004). A Comparison of Radiation-Hard and Radiation-Tolerant FPGAs for Space Applications. Technical report, NASA. Available from : `http://nepp.nasa.gov/docuploads/3C8F70A3-2452-4336-B70CDF1C1B08F805/JPLRad-TolerantFPGAsforSpaceApplications.pdf`. (Pages 1 et 6)

[Samudrala et al., 2004] Samudrala, P. K., Ramos, J., and Katkoori, S. (2004). Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. *Nuclear Science, IEEE Transactions on*, 51(5) :2957–2969. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1344451`. (Pages 10 et 125)

[Scholzel, 2010] Scholzel, M. (2010). HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 723–728. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5456957`. (Pages 10 et 16)

[Schweizer et al., 2011] Schweizer, T., Schlicker, P., Eisenhardt, S., Kuhn, T., and Rosenstiel, W. (2011). Low-Cost TMR for Fault-Tolerance on Coarse-Grained Reconfigurable Architectures. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 135–140. Ieee. Available from : `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6128567`. (Pages 10 et 29)

[She et al., 2010] She, X., Li, N., and Farwell, W. D. (2010). Tunable SEU-Tolerant Latch. *Nuclear Science, IEEE Transactions on*, 57(6) :3787–3794. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5658075`. (Page 8)

[Shukla et al., 2006] Shukla, S., Bergmann, N. W., and Becker, J. (2006). QUKU : A fast run time reconfigurable platform for image edge detection. *Reconfigurable Computing : Architectures and Applications*, 3985/2006 :93–98. Available from : `http://www.springerlink.com/index/DK6607L8086775N5.pdf`. (Page 19)

[Singh et al., 2000] Singh, H., Lee, M.-H., Lu, G., Kurdahi, F. J., Bagherzadeh, N., and Filho, E. M. C. (2000). MorphoSys : an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5) :465–481. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=859540`. (Page 18)

[Singh et al., 2006] Singh, K., Agbaria, A., Kang, D., and French, M. (2006). Tolerating SEU faults in the RAW architecture. In *Dependable Embedded Systems, 3rd International Workshop on*. Available from : `http://pdf.aminer.org/000/310/308/a_system_architecture_for_software_fault_tolerance.pdf`. (Page 28)

[Taylor et al., 2002] Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumper, V., Frank, M., Amarasinghe, S., and Agarwal, A. (2002). The Raw microprocessor : A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, pages 25–35. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=997877`. (Page 18)

[Trabelsi, 2008] Trabelsi, K. (2008). *Méthodes d'optimisation pour la conception sous contraintes de systèmes et de circuits électroniques*. PhD thesis, Université de Bretagne Sub (UBS). (Page 139)

[Vázquez-Luque et al., 2013] Vázquez-Luque, A., Marín, J., Terrón, J. A., Pombar, M., Bedogni, R., Sánchez-Doblado, F., and Gómez, F. (2013). Neutron Induced Single Event Upset Dependence on Bias Voltage for CMOS SRAM With BPSG. *Nuclear Science, IEEE Transactions on*, 60(6) :4692–4696. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6651671`. (Pages 1 et 8)

[Voyiatzis and Halatsis, 2005] Voyiatzis, I. and Halatsis, C. (2005). A Low-Cost Concurrent BIST Scheme for Increased Dependability. *IEEE Transactions on Dependable and Secure Computing*, 2(2) :150–156. Available from : `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1453533`. (Page 13)

[Wan et al., 2009] Wan, L., Dong, C., and Chen, D. (2009). A New Coarse-Grained Reconfigurable Architecture with Fast Data Relay and Its Compilation Flow. *Proceedings of Symposium on Application Accelerators in HPC*, 2 :3–5. Available from : `http://saahpc.ncsa.illinois.edu/09/papers/Wan_paper.pdf`. (Page 18)

[Wang et al., 2011] Wang, Y., Zhang, L., Han, Y., Li, H., and Li, X. (2011). Elastic CGRA : Circumventing Hard-faults Through Instruction Mitigation. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Available from : `http://www.carch.ac.cn/~LeiZhang/papers/dsn-2011-fa-wang.pdf`. (Pages 15, 30 et 31)

[Xilinx, 2010] Xilinx (2010). Space-Grade Virtex-4QV Family Overview. Technical report, Xilinx. Available from : `http://www.xilinx.com/support/documentation/data_sheets/ds653.pdf`. (Page 7)

[Xilinx, 2012] Xilinx (2012). Radiation-Hardened, Space-Grade Virtex-5QV Family Overview. Technical report, Xilinx. Available from : `http://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf`. (Page 7)

[Xilinx, 2014] Xilinx (2014). 7 Series FPGAs Overview (DS180). Available from : `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`. (Page 16)

[Zhao et al., 2011] Zhao, Q., Ichinomiya, Y., Amagasaki, M., Iida, M., and Sueyoshi, T. (2011). A Novel Soft Error Detection and Correction Circuit for Embedded Reconfigurable Systems. *Embedded Systems Letters, IEEE*, 3(99) :1–1. Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6009172`. (Pages 15 et 37)

[Zhou and Shen, 2007] Zhou, G. and Shen, X. (2007). A Coarse-Grained Dynamically Reconfigurable Processing Array (RPA) for Multimedia Application. In *Natural Computation, 2007. ICNC 2007.* Available from : `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4344829`. (Page 18)

Hardware architecture and associated programming flow for the design of digital fault-tolerant systems

Abstract

Whether in automotive with heat stress or in aerospace and nuclear field subjected to cosmic, neutron and gamma radiation, the environment can lead to the development of faults in electronic systems. These faults, which can be transient or permanent, will lead to erroneous results that are unacceptable in some application contexts. The use of so-called rad-hard components is sometimes compromised due to their high costs and supply problems associated with export rules.

This thesis proposes a joint hardware and software approach independent of integration technology for using digital programmable devices in environments that generate faults. Our approach includes the definition of a Coarse Grained Reconfigurable Architecture (CGRA) able to execute entire application code but also all the hardware and software mechanisms to make it tolerant to transient and permanent faults. This is achieved by the combination of redundancy and dynamic reconfiguration of the CGRA based on a library of configurations generated by a complete conception flow. This implemented flow relies on a flow to map a code represented as a Control and Data Flow Graph (CDFG) on the CGRA architecture by obtaining directly a large number of different configurations and allows to exploit the full potential of architecture.

This work, which has been validated through experiments with applications in the field of signal and image processing, has been the subject of two publications in international conferences and of two patents.

Keywords

Fault tolerance ; CGRA ; Triplication ; Scheduling ; Binding ; Mapping ; DFG ; CDFG.

Architecture matérielle et flot de programmation associé pour la conception de systèmes numériques tolérants aux fautes

---

Résumé

Que ce soit dans l'automobile avec des contraintes thermiques ou dans l'aérospatial et le nucléaire soumis à des rayonnements ionisants, l'environnement entraîne l'apparition de fautes dans les systèmes électroniques. Ces fautes peuvent être transitoires ou permanentes et vont induire des résultats erronés inacceptables dans certains contextes applicatifs. L'utilisation de composants dits « *rad-hard* » est parfois compromise par leurs coûts élevés ou les difficultés d'approvisionnement liés aux règles d'exportation.

Cette thèse propose une approche conjointe matérielle et logicielle indépendante de la technologie d'intégration permettant d'utiliser des composants numériques programmables dans des environnements susceptibles de générer des fautes. Notre proposition comporte la définition d'une Architecture Reconfigurable à Gros Grains (CGRA) capable d'exécuter des codes applicatifs complets mais aussi l'ensemble des mécanismes matériels et logiciels permettant de rendre cette architecture tolérante aux fautes. Ce résultat est obtenu par l'association de redondance et de reconfiguration dynamique du CGRA en s'appuyant sur une banque de configurations générée par une chaîne de programmation complète. Cette chaîne outillée repose sur un flot permettant de porter un code sous forme de *Control and Data Flow Graph* (CDFG) sur l'architecture en obtenant un grand nombre de configurations différentes et qui permet d'exploiter au mieux le potentiel de l'architecture.

Les travaux, qui ont été validés aux travers d'expériences sur des applications du domaine du traitement du signal et de l'image, ont fait l'objet de publications en conférences internationales et de dépôts de brevets.

Mots-clefs

Tolérance aux fautes, CGRA, Triplication, Ordonnancement, Assignation, DFG, CDFG.