THÈSE

présentée

DEVANT L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

pour obtenir

le grade de : Docteur de l'Université de Bretagne Occidentale
mention : Informatique

PAR

Damien Picard

Équipe d'accueil :	Université de Brest, CNRS, UMR 3192 Lab-STICC, ISSTB
École Doctorale :	SICMA
Composante universitaire :	Sciences et Techniques

TITRE DE LA THÈSE :

Méthodes et outils logiciels pour l'exploration architecturale d'unités reconfigurables embarquées

SOUTENUE LE 4 novembre 2010 devant la Commission d'Examen

COMPOSITION DU JURY :

MM	J.-L.	Dekeyser	Rapporteurs
	O.	Sentieys	
	J.-P.	Babau	Examinateurs
	S.	Pillement	
	G.	Sassatelli	
	L.	Lagadec	

Remerciements

Je remercie très chaleureusement Loic Lagadec d'avoir accepté de diriger et d'encadrer les travaux de cette thèse. Sa patience, son engagement dans les moments difficiles ainsi que la relation de confiance et d'amitié qui s'est instaurée entre nous sont pour beaucoup dans la réussite de ces travaux.

J'adresse mes remerciements à Jean-Luc Dekeyser et Olivier Sentieys qui ont accepté la charge de rapporteur, à Jean-Philippe Babau pour avoir accepté de présider le jury de soutenance, à Sébastien Pillement et Gilles Sassatelli qui ont accepté d'examiner ce travail.

Je remercie également Bernard Pottier pour la confiance qu'il m'a accordé en m'intégrant au projet européen MORPHEUS qui a été un point de départ déterminant pour mes travaux de recherche. En particulier, j'ai pu effectuer une mobilité dans l'équipe STMicroelectronics/ARCES sous la direction de Fabio Campi à qui j'adresse mes remerciements pour son soutien.

L'environnement de travail quotidien joue un rôle important, je tiens donc à remercier l'ensemble de l'équipe Architecture & Systèmes que j'ai cotoyé pendant ces trois années. Je remercie tout particulièrement Ciprian Teodorov en tant que compagnon de route dans cette aventure.

Cette thèse a été une période intense de ma vie qui a embarqué à son bord mes proches sans qui rien n'aurait été possible. Tout d'abord je remercie mes parents qui m'ont soutenu depuis le début et dont les encouragements permanents ont gardé à flot ma motivation. Je remercie également mon grand-père qui a toujours suivi de très près mes avancées et qui peut constater jusqu'où mène la poésie dans les mathématiques.

J'adresse mes remerciements à Yuki dont la sérénité communicative, même au cœur de la tempête rédactionnelle, m'a permis de garder le cap. Je remercie également Annie pour ses relectures minutieuses qui ont grandement contribué à l'amélioration de ce manuscrit.

Enfin, je remercie avec tendresse Anaëlle à qui je dédie ces travaux. Sa présence de tous les instants et la patience dont elle a su faire preuve durant ces trois années nous ont permis de naviguer sans encombre jusqu'à bon port.

Table des matières

Table des figures

Liste des tableaux

Liste des sigles et acronymes

ADL *Architecture Description Language*
ADRES *Architecture for Dynamically Reconfigurable Embedded Systems*
ALAP *As Late As Possible*
API *Application Programming Interface*
ASAP *As Soon As Possible*
ASIC *Application Specific Integrated Circuit*
ASIP *Application-Specific Instruction-set Processor*
BLIF *Berkeley Logic Interchange Format*
CABA *Cycle Accurate and Bit Accurate*
CDFG *Control Data Flow Graph*
CGRA *Coarse-Grained Reconfigurable Architecture*
CLB *Configurable Logic Block*
CSP *Communicating Sequential Process*
DFG *Data Flow Graph*
DMA *Direct Memory Access*
DNA *Direct Network Access*
Dnode *Data node*
DRAGE *Dynamic Reconfigurable Architecture Generation Environment*
DRESC *Dynamically Reconfigurable Embedded System Compiler*
DSP *Digital Signal Processing*
EDIF *Electronic Design Interchange Format*
eFPGA *embedded FPGA*
FIFO *First In First Out*
FIR *Finite Impulse Response*
FPGA *Field Programmable Gate Array*
FSL *Fast Simplex Link*
GA *Générateur d'Adresses*
GILES *Good Instant Layout of Erasable Semiconductors*
HCDFG *Hierarchical Control Data Flow Graph*

HDL *Hardware Description Language*
HL-CDFG *High-Level CDFG*
ICAP *Internal Configuration Access Port*
IOB *Input Output Block*
IP *Intellectual Property*
ISA *Instruction Set Architecture*
LISA *Language for Instruction Set Architecture*
LL-CDFG *Low-Level CDFG*
LUT *LookUp Table*
MAML *MAchine Markup Language*
MORPHEUS *Multi-purpose dynamically Reconfigurable Platform for intensive Heterogeneous processing*
MPCalc *Modèle du Plan de Calcul*
MPConf *Modèle du Plan de Configuration*
NOP *No OPeration*
PA *Point d'Arrêt*
PE *Processing Element*
PiCoGA *Pipelined Configurable Gate Array*
PO *Point d'Observation*
RAM *Random-Access Memory*
rASIP *reconfigurable Application-Specific Instruction-set Processor*
rDPA *reconfigurable DataPath Array*
rDPU *reconfigurable DataPath Units*
RISC *Reduced Instruction Set Computer*
RLC *Reconfigurable Logic Cell*
RSoC *Reconfigurable System on Chip*
RTL *Register Transfer Level*
SAD *Sum of Absolute Difference*
SoC *System on Chip*
SRAM *Static Random Access Memory*
SSA *Static Single Assignment*
UAL *Unité Arithmétique et Logique*
UF *Unité Fonctionnelle*
VHDL *VHSIC Hardware Description Language*
VHSIC *Very-High-Speed Integrated Circuit*
VLIW *Very Long Instruction Word*
VPR *Versatile Place and Route*
xMAML *eXtended MAML*
XML *eXtensible Markup Language*
XP *eXtreme Programming*

Chapitre 1

Introduction

1.1 Contexte

L'augmentation continue de la complexité des applications embarquées et leur évolution rapide exigent des systèmes de plus en plus performants et flexibles. Ce constat est particulièrement vrai dans les domaines applicatifs liés au traitement du signal tels que les télécommunications sans fil ou encore le multimédia. Du côté industriel, la production de circuits en réponse aux besoins s'inscrit dans un contexte fortement concurrentiel. La course à l'innovation des fabricants nécessite une réduction conséquente des délais de mise sur le marché du produit et une forte productivité des concepteurs [64].

La montée en performance des circuits est soutenue par le doublement chaque année de la capacité d'intégration des semi-conducteurs [112]. Ainsi, la réalisation de systèmes complets sur une seule puce (qualifiés de *System on Chip* (SoC)) est aujourd'hui la norme pour répondre aux besoins applicatifs. Les SoC sont principalement composés de circuits programmables et spécifiques (*Application Specific Integrated Circuit* (ASIC)). Ces derniers, de part leurs spécialisation pour le calcul spatial, sont dédiés à l'accélération des parties intensives des applications qui requièrent de hautes performances. Cependant, leurs coûts et leurs délais de conception/validation ne respectent pas les impératifs économiques et temporels de production. De plus, ils ne peuvent adresser les besoins évolutifs des applications qui imposent une spécialisation tardive du SoC et la possibilité de changer les fonctionnalités post-fabrication. Par conséquent, les méthodes de conception favorisent la réutilisation de composants préexistants, nommés *Intellectual Property* (IP), dont une part importante est programmable. Parmi eux on trouve des architectures reconfigurables qui offrent un compromis entre la flexibilité des processeurs et les performances des ASIC.

Une architecture reconfigurable est structurée sous forme de matrices d'unités de calcul interconnectées suivant différentes topologies (e.g. en *mesh*) [33]. Ses ressources de calcul et de routage sont configurables et autorisent donc un changement de fonctionnalité du circuit post-fabrication. La flexibilité apportée par la reconfiguration et les performances de l'organisation spatiale des ressources combinent les avantages du logiciel et du matériel. A cela s'ajoute une large variété d'architectures qui répondent au besoin de spécialisation pour certains domaines applicatifs. Les variations interviennent sur les ressources de routage et de calcul avec une granularité élevée pour des calculs orientés arithmétique (e.g. FFT, DCT, etc.) tandis qu'une granularité plus fine sera adaptée aux tâches de contrôle (e.g. machine à états finis). Le couplage de ces différents types d'architectures dans un même SoC amène la capacité de

balayer un large domaine applicatif et augmente la flexibilité.

Un exemple récent de *Reconfigurable System on Chip* (RSoC) est donnée par le projet *Multi-purpose dynamically Reconfigurable Platform for intensive Heterogeneous processing* (MORPHEUS) [124] qui a servi de cadre à nos travaux. Ce système intègre sous forme de coprocesseurs trois architectures reconfigurables de différentes granularités [125].

Dans ce contexte nos travaux s'intéressent à l'exploration architecturale des unités reconfigurables embarquées dans un RSoC.

1.2 Problématique de l'étude

Lors du développement d'un SoC, la tâche principale du concepteur est de sélectionner et d'intégrer les IP pour répondre aux besoins applicatifs tout en respectant les délais de mise sur le marché. Or la taille des espaces de conception à explorer pour chaque IP est vaste et la sélection des critères à prendre en compte est critique. Ainsi, dans le cas des architectures reconfigurables, la recherche d'adéquation avec le domaine applicatif implique l'exploration de critères tels que la granularité des ressources, le type de topologie ou encore les modes de reconfiguration dynamique. Outre ces aspects architecturaux, la facilité de programmation et la validation fonctionnelle est à considérer car elle influe sur la productivité des concepteurs d'applications.

Pour permettre une exploration rapide et efficace du vaste espace de conception des architectures reconfigurables, le concepteur a besoin de méthodes et d'outils pour : (1) évaluer les variantes architecturales suivant différentes métriques (e.g. temps de configuration, nombre de ressources, etc.) ; (2) programmer rapidement et facilement l'unité reconfigurable pour itérer sur plusieurs applications ; (3) valider fonctionnellement une exécution de l'application au niveau système puis par simulation *in situ* après synthèse. Les travaux présentés dans ce manuscrit adressent ces trois points par un flot d'exploration d'unités reconfigurables embarquées.

1.3 Contributions

Notre approche est fondée sur une modélisation à haut-niveau de l'unité reconfigurable qui permet la génération d'un prototype matériel et la mise à disposition de ses outils applicatifs. La méthodologie de modélisation décompose l'architecture en deux plans dont le premier correspond aux ressources de calcul/routage (plan de calcul) et le second à la structuration de la mémoire de configuration (plan de configuration). Le plan de calcul est modélisé dans le langage de description architecturale de l'outil Madeo-BET [81]. Le modèle produit est exploitable par ses outils de placement-routage génériques. Il supporte également une évaluation en fonction de critères de routabilité et de taux d'occupation des ressources. Le plan de configuration est découpé en zones reconfigurables indépendantes et multi-contextes. Il permet d'évaluer les temps de configurations des zones, la taille des binaires et le nombre de ressources matérielles nécessaires.

Les deux plans sont pris en entrée d'un générateur de code *VHSIC Hardware Description Language* (VHDL) comportemental qui produit un environnement complet de simulation. Celui-ci est composé du prototype de l'unité reconfigurable et d'un contrôleur microprogrammable qui gère les modes de reconfiguration dynamique partielle et multi-contextes.

L'unité reconfigurable générée est programmée par un flot de synthèse de haut-niveau qui prend en entrée la description d'un réseau de processus dont les comportements sont décrits dans une variante de C. Ce flot cible également des unités reconfigurables préexistantes par le couplage de sa partie basse avec des chaînes d'outils tiers (e.g. via la génération de *netlists* Verilog).

La validation fonctionnelle de l'application est réalisée à plusieurs niveaux et en deux étapes. La première étape est une validation *in situ* par une configuration du prototype matériel de l'unité. Les binaires de configuration sont générés à partir d'un placement-routage Madeo-BET et des contraintes du modèle matériel. Ils sont ensuite ordonnancés dans le contrôleur de configuration microprogrammable pour une exécution par simulation. La seconde étape est une validation fonctionnelle, indépendante de l'unité, dans un modèle de SoC. A ce niveau l'impact des activités systèmes sur l'exécution est pris en compte.

La synergie de nos outils est démontrée par l'exploration et le prototypage sur carte FPGA d'une architecture gros-grain. Elle est structurée en étages pipelinés et supporte une exécution virtualisée par l'utilisation de la reconfiguration dynamique.

Le bénéfice principal apporté par notre flot est la production automatisée d'un environnement d'évaluation d'une unité reconfigurable modélisée à haut-niveau. Ainsi, il offre aux concepteurs une capacité de validation précoce de concepts architecturaux en amont d'une réalisation optimale ASIC de l'unité.

1.4 Contexte de développement des outils

Les travaux présentés dans ce manuscrit s'appuient principalement sur les activités menées dans le cadre du projet européen MORPHEUS au sein du groupe « *Spatial Design* » [124]. Ce projet a regroupé un ensemble d'acteurs industriels et académiques pour la réalisation d'un RSoC hétérogène et de sa chaîne complète de programmation. La tâche du groupe « *Spatial Design* » était de contribuer au développement d'une chaîne de synthèse de haut-niveau pour cibler les unités reconfigurables du SoC. Le laboratoire LabSTICC/AS a mis à contribution son expertise dans la synthèse pour les architectures reconfigurables afin de proposer un *back-end* multi-cibles appelé Madeo+ [86]. Cet outil a servi de base aux travaux décrits dans ce rapport pour le développement d'un cadre logiciel qui couvre trois axes principaux : la programmation multi-cibles d'unités reconfigurables, la validation des applications à différents niveaux d'abstractions et la réalisation d'unités reconfigurables par une modélisation haut-niveau dans un langage de description d'architecture (*Architecture Description Language* (ADL)).

La figure 1.1 détaille le flot global du cadre outil présenté dans ces travaux. Il s'articule autour de trois parties principales centrées autour de Madeo+ et communiquant par l'utilisation d'une représentation intermédiaire de l'application (*Control Data Flow Graph* (CDFG)). Les descriptions de chaque partie, allant de gauche à droite, sont données dans les paragraphes ci-dessous.

Madeo/DRAGE : approche ADL pour le prototypage d'unités reconfigurables. Madeo-BET est un cadre de modélisation et de programmation d'architectures reconfigurables. Dans sa version initiale il propose une approche fondée sur un langage de description d'architecture pour modéliser une grande variété d'architectures reconfigurables. A partir de la description un modèle est généré sur lequel les outils de placement/routage opèrent.

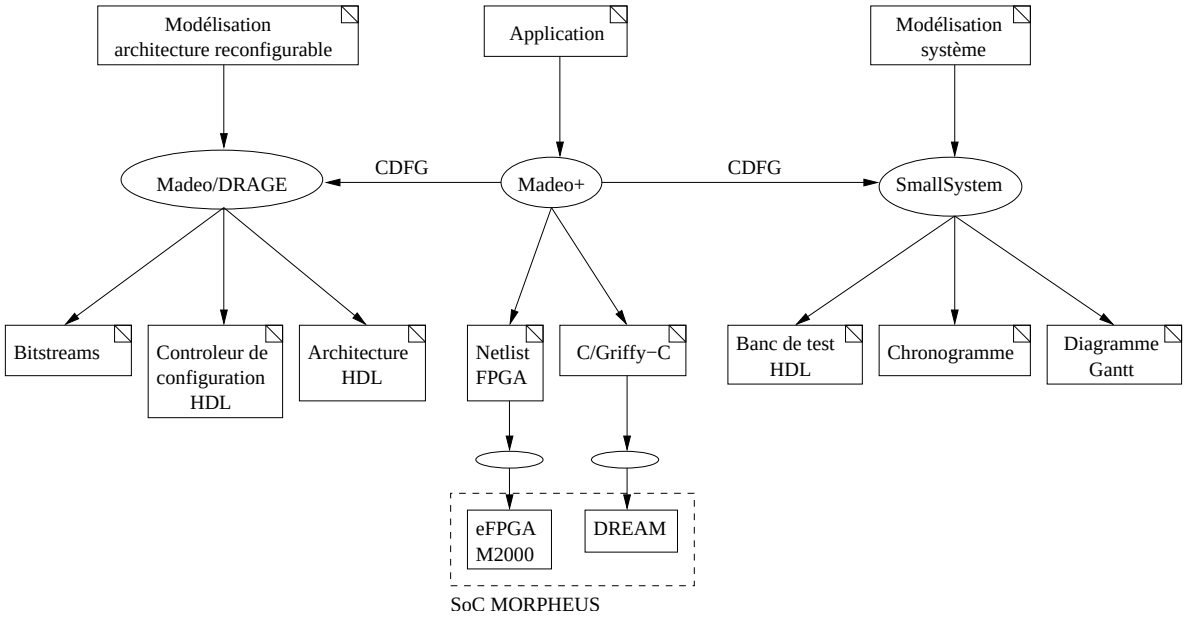


FIG. 1.1 – Flot global.

Dans ces travaux nous avons ajouté à Madeo-BET l'outil *Dynamic Reconfigurable Architecture Generation Environment* (DRAGE) pour obtenir le couplage Madeo/DRAGE. DRAGE permet de générer l'architecture matérielle (dans le langage VHDL) correspondant au modèle. Avant la génération de l'architecture, la structure du plan de configuration en zones reconfigurables est spécifiée (dimensionnement et nombre de contextes de configuration). Le modèle obtenu permet la génération d'un contrôleur de configuration microprogrammable et la spécialisation du synthétiseur de *bitstream*. Les *bitstreams* générés pour chaque domaine sont ordonnancés pour produire le microprogramme du contrôleur de configuration. L'ensemble de l'architecture reconfigurable est testé par un banc de test généré automatiquement.

Madeo+ : la synthèse multi-cibles. Cette partie constitue la chaîne de synthèse haut-niveau multi-cibles pour RSoC hétérogène. La spécification en entrée est décrite sous la forme d'un réseau de processus de deux types [122]. Le premier type correspond à des processus de calcul orientés flot de données et spécifiés dans un sous-ensemble du langage C. Ils réalisent les opérations de calculs sur les données. Le deuxième type correspond à des processus de communication qui adressent directement les mémoires locales de l'architecture reconfigurable. Ils servent à mémoriser des résultats intermédiaires qui transitent entre les processus de calcul et les résultats finaux.

L'outil Madeo+ synthétise, en fonction de la cible sélectionnée, soit une *netlist* Verilog liée à la librairie FlexEOS du *embedded FPGA* (eFPGA) M2000 [3] ou du code C/Griffy-C qui cible l'architecture DREAM [27]. Les fichiers en sortie sont pris en entrée des outils de synthèse liés aux cibles matérielles. Madeo+ abstrait l'hétérogénéité des unités reconfigurables par la définition d'une interface commune et l'utilisation d'une représentation intermédiaire partagée par les différents outils.

SmallSystem : simulation multi-niveaux. Cette partie concerne la validation fonctionnelle d'applications mises en œuvre sur un RSoC [121]. La méthodologie est fondée sur l'exécution de l'application intégrée dans un modèle système de la plate-forme d'exécution. Il est produit à partir d'un modèle de composants défini dans l'environnement du langage dynamique Smalltalk [49]. Cette approche, très similaire à celle de System-C, permet de bénéficier des avantages des langages dynamiques en terme de niveau d'abstraction et de capacité reflexive[1]. De plus, le laboratoire bénéficie d'un héritage logiciel important développé dans ce langage et directement ré-injectable dans ces travaux. L'exemple le plus significatif est l'outil Madeo [81].

La simulation multi-niveaux produit en sortie un diagramme de Gantt qui permet de visualiser l'ensemble des activités systèmes ainsi que des chronogrammes qui tracent les signaux de l'application à valider (si celle-ci est représentée à un niveau *netlist Register Transfer Level* (RTL)).

Un modèle de banc de test est utilisé pour définir les interactions entre l'application exécutée sur l'unité reconfigurable et le système. La simulation de l'application dans des outils tiers (e.g. ModelSim) est rendue possible par la génération automatique du banc de test en Verilog.

1.5 Plan du mémoire

Le chapitre 2 donne une vue générale du domaine des architectures reconfigurables. Leurs caractéristiques principales telles que la granularité, le couplage ou les modes de reconfiguration dynamiques sont illustrées par des travaux académiques et industriels.

Le chapitre 3 est centré sur les méthodes d'exploration pour les architectures reconfigurables. Nous abordons également les ADL et leur application au domaine. Un point particulier est effectué sur l'outil Madeo qui sert de base pour l'outil DRAGE développé dans nos travaux. Enfin, les différentes méthodes existantes sont comparées pour situer notre approche qui est détaillée dans les chapitres suivants.

Le chapitre 4 présente un flot d'exploration et de prototypage d'architectures reconfigurables nommé Madeo/DRAGE. Le positionnement de DRAGE par rapport à Madeo est donnée par son flot global. Le deux plans de modélisation d'une architecture : calcul et configuration sont décrit séparément. A cela s'ajoute l'extraction des *bitstreams* à partir d'un placement-routage dans Madeo-BET.

Le chapitre 5 détaille un flot de programmation de haut-niveau pour les architectures reconfigurables prototypées par DRAGE. L'application est décrite dans notre langage Avel sous la forme d'un réseau de processus communicants. Le modèle d'exécution considéré est une unité reconfigurable de SoC interfacée avec des mémoires locales. La description est prise en entrée du flot de synthèse (qui cible des architectures modélisées dans Madeo-BET) et générée par DRAGE ou des architectures tierces. Dans ce dernier cas le format de sortie est conforme aux outils de synthèse propriétaires.

Le chapitre 6 aborde la validation fonctionnelle des applications Avel. Pour cela elles sont simulées à plusieurs niveaux d'abstraction dans un modèle de RSoC. Celui-ci est décrit dans le cadriciel de composants objets *SmallSystem*. La validation par des outils est abordée par une génération de bancs de test Verilog à partir d'une modélisation des stimuli systèmes. Enfin, les aspects débogage sont traités par l'insertion de sondes dans le code de haut-niveau

[1] Capacité du langage à examiner et à modifier dynamiquement ses structures internes.

qui sont ensuite synthétisées. Elles sont interfacées à un module de débogage embarqué pour obtenir une contrôlabilité similaire à celle du logiciel (pause de l'exécution, reprise, etc.).

Les différentes parties du flot sont validées dans le chapitre 7 par la modélisation et l'exploration d'une architecture gros-grain. Elle est structurée en étages pipelinées et supporte leur virtualisation par reconfiguration dynamique partielle/multi-contextes. Différents degrés de virtualisation sont explorés par une réduction des tailles de matrices. Les performances sont évaluées par la mise en œuvre d'un filtre FIR-8 décrit en Avel. La faisabilité du prototypage matériel des architectures générées est démontrée par une mise en œuvre sur une carte Xilinx XUPV5-LX110T qui embarque un FPGA Virtex-5 [148, 154]. Enfin, la validation fonctionnelle de l'application est détaillée par les résultats de la simulation multi-niveaux.

Le dernier chapitre effectue la synthèse des points principaux de notre contribution et dresse les perspectives de recherche ouvertes par ces travaux.

Chapitre 2

Caractéristiques des architectures reconfigurables

2.1 Introduction

Les architectures reconfigurables sont devenues un compromis crédible entre les architectures programmables de type processeur et les ASIC. Elles ont pour avantage d'associer la programmabilité du logiciel à l'efficacité de l'exécution spatiale du matériel [39]. Ainsi, elles peuvent être considérées comme la réunion de deux mondes, celui de l'électronique et du logiciel [102].

De manière générale, sur le plan structurel, une architecture reconfigurable est composée d'un réseau d'unités de calcul programmables [18]. Elles mettent en œuvre des opérations allant de la fonction logique aux opérateurs arithmétiques. Ces unités sont interconnectées par un réseau de routage dont les ressources communiquent via des points d'interconnexion programmables. La fonctionnalité du circuit est déterminée par la programmation des éléments de calcul et de routage par un ensemble de bits, nommée binaire de configuration ou *bitstream*, qui détermine un contexte de configuration. Ce binaire est chargé dans les mémoires de configuration associées aux éléments et dont l'ensemble définit un plan de configuration. Le temps d'envoi du binaire dans le plan correspond à la latence de configuration.

La nature des architectures reconfigurables varie suivant un certain nombre de critères dont les principaux sont détaillés dans ce chapitre.

2.2 Granularité des architectures reconfigurables

La granularité des architectures reconfigurables détermine la taille (en bits) des mots de données traités par les ressources de routage et de calcul. On distingue deux types de granularité : grain-fin et gros-grain (ou grain épais).

2.2.1 Architectures à grain-fin

Les architectures à grain-fin opèrent sur une largeur de mot de 1-bit. Cependant, d'autres architectures utilisent une largeur supérieure (e.g. Garp [26] peut router des mots de 2 bits) mais qui reste inférieure à celles classiquement utilisées par les processeurs (8, 16, 32 bits). Cette granularité apporte un haut niveau de flexibilité qui permet de cibler des domaines

applicatifs orientés vers le calcul intensif ou le contrôle. Ainsi, les architectures commerciales de grandes capacités sont utilisées pour la mise en œuvre de SoC [20].

Les architectures de cette classe sont traditionnellement représentées par les circuits *Field Programmable Gate Array* (FPGA) introduits par la société Xilinx dans les années 1980 [146]. Par la suite, ce type d'architecture a été étudié et utilisé dans de nombreux travaux académiques pour la réalisation de plate-formes d'exécution [8, 9, 143] et de leurs outils applicatifs [18].

Les premiers FPGA étaient spécialisés pour la mise en œuvre de fonctions logiques spécifiées par l'interconnexion de portes logiques. Le dispositif adopté massivement pour la mise en œuvre des fonctions est la *LookUp Table* (LUT). Une LUT à N entrées est une mémoire qui contient la table de vérité de n'importe quelle équation logique à N entrées. Dans le but d'augmenter la capacité de calcul par unité de surface de silicium (qui définit la *densité de calcul*) et de réduire la complexité des opérations de placement-routage les LUT sont regroupées dans des cellules [5, 19, 15].

Dans la terminologie Xilinx une cellule est nommée *Configurable Logic Block* (CLB) et constitue une unité hiérarchique de calcul. Les fonctions logiques combinatoires ou séquentielles sont mises en œuvre par l'interconnexion des CLB. La figure 2.1 illustre la composition interne d'un CLB de l'architecture Xilinx XC4000 [149]. Un bloc correspond à la paramétrisation d'un schéma de base structuré autour d'une LUT dont la sortie peut être tamponnée dans un registre. Les FPGA les plus récents de la famille Virtex ont multiplié les niveaux hiérarchiques de regroupement et le nombre d'entrées des LUT au fur et à mesure des progrès technologiques. Un CLB du FPGA Virtex-6 contient deux sous-groupes, appelés *Slice*, qui eux même contiennent quatre LUT à six entrées [155].

Un FPGA est généralement structuré par la réplication régulière d'un motif de base (CLB et ressources de routage) pour former une architecture en îlots de calcul (voir figure 2.2). La matrice de CLB est elle-même entourée de blocs d'entrées-sorties configurables pour alimenter en données les calculs.

Les CLB sont intégrés dans une infrastructure de routage dont les fils sont regroupés dans des canaux de routage. Ces canaux sont segmentés par des interrupteurs configurables (tampons trois états, *pass transistor*) et incluent des fils de longueurs différentes en fonction du degré de localité des connexions [17]. Les blocs de connexion connectent les fils pour les longues distances aux fils pour les courtes distances. Ils sélectionnent également les fils connectés aux entrées-sorties des CLB. Au croisement de deux canaux, les fils sont aiguillés (granularité de 1 bit) par des matrices (bloc *switch*) qui permettent les changements de direction. Les entrées-sorties des CLB sont connectées aux bus par des multiplexeurs et des interrupteurs configurables. La figure 2.2 illustre un exemple de structure de routage configurable dont les bits de configuration sont maintenus par des mémoires de type *Static Random Access Memory* (SRAM).

Flexibilité versus spécialisation Les FPGA s'avèrent inefficaces pour la mise en œuvre d'applications orientées vers le calcul arithmétique dont la taille des mots traités est supérieure à 1 bit. En effet, la flexibilité apportée par la granularité fine des ressources implique un certain nombre de désavantages en terme de performance, surface et consommation en comparaison de solutions spécifiques [79]. En particulier, les FPGA souffrent d'un sur-coût au niveau des ressources de routage qui occupent jusqu'à 90% de la surface du circuit [38, 58]. Le principe

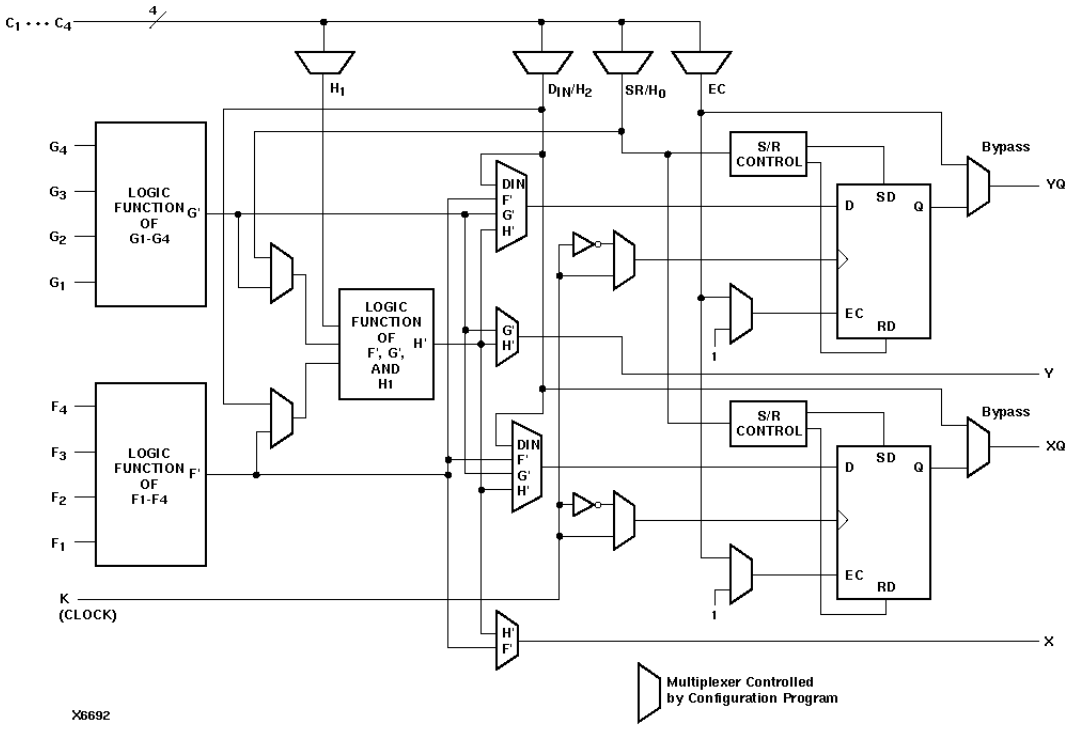


FIG. 2.1 – Structure d'un bloc logique configurable du FPGA Xilinx XC4000. Le bloc est composé de 2 LUT à 4 entrées et d'une LUT à 3 entrées. Les sorties des LUT peuvent être tamponnées dans deux registres. Une chaîne de propagation de retenue est utilisée pour optimiser la mise en œuvre en cascade d'additionneurs complets [149].

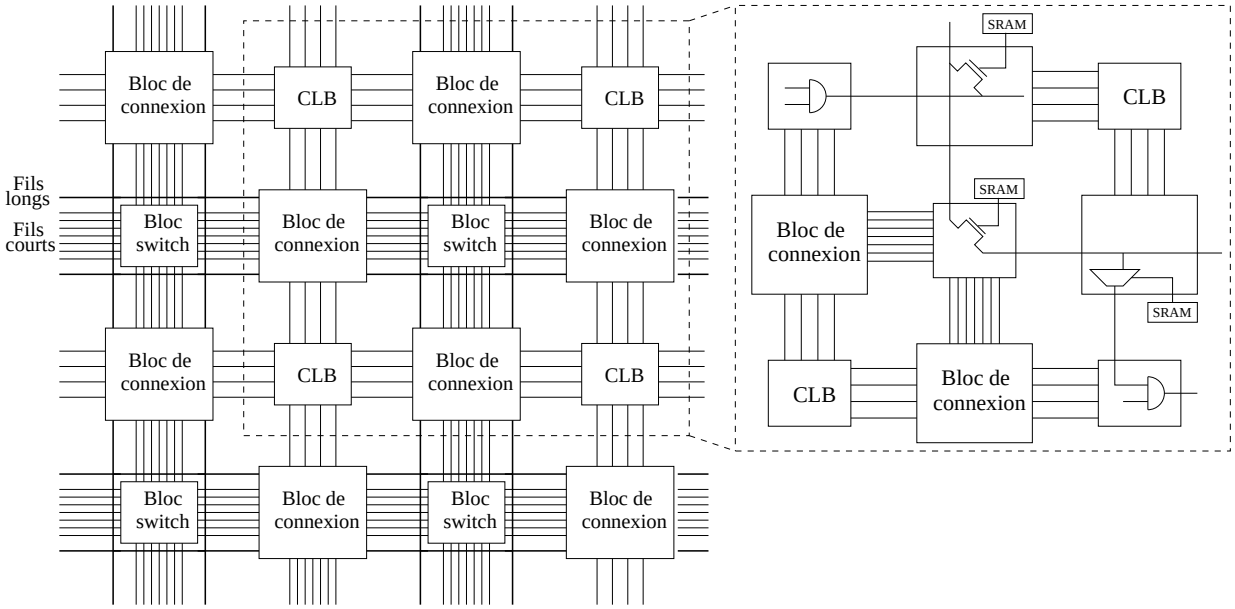


FIG. 2.2 – Architecture d'un FPGA en îlots de calcul.

de routage entre CLB n'exploite pas la structure régulière des opérateurs arithmétiques [56, 57]. Cela résulte en une faible densité de calcul avec une occupation en surface importante. La dégradation des performances est principalement due aux coûts de routage (en nombre d'interrupteurs programmables) et aux délais de propagation élevés.

Un autre aspect concerne la taille des contextes de configuration. Le nombre élevé de dispositifs configurables dans les FPGA impose une taille importante des binaires de configurations. Ce paramètre influe sur l'augmentation des temps de configuration et sur la quantité de mémoire nécessaire au stockage des contextes.

Pour réduire les sur-coûts engendrés par la granularité fine des ressources, les FPGA commerciaux embarquent des blocs de calcul de granularité supérieure. Par exemple, le Virtex-6 intègre des blocs *Digital Signal Processing* (DSP) spécialisés pour la mise en œuvre d'opérations de traitement du signal. Ce type d'opérateur améliore la densité de calcul, la consommation et les performances de l'architecture. Ces plate-formes hétérogènes offrent un certain degré de spécialisation tout en gardant une orientation générique. Cependant, lorsque le domaine applicatif est très restreint la polyvalence s'avère inutile car facteur d'inefficacité. C'est pourquoi des architectures reconfigurables à gros-grain spécialisées dans le calcul arithmétique ont été développées.

2.2.2 Architectures à gros-grain

Les architectures reconfigurables à gros-grain sont spécialisées pour des traitements arithmétiques [57]. Leurs ressources de calcul et de routage opèrent sur une granularité de mot supérieure à 1 bit et se situent à un niveau opérateur. La capacité de calcul d'une cellule de calcul varie d'un regroupement de quelques LUT à des *Unité Arithmétique et Logique* (UAL) de largeur 32 bits. En comparaison d'un FPGA, l'utilisation d'opérateurs dédiés amène une densité de calcul supérieure. En effet, un opérateur arithmétique équivaut à plusieurs CLB interconnectés par de multiples interrupteurs configurables. Les topologies de routage sont généralement spécialisées pour une exécution flot de données qui nécessite très peu de contrôle. Dans certains cas un contrôleur dédié gère le séquencement. Cette approche diffère des FPGA sur lesquels sont synthétisés le contrôle et la logique. La spécialisation et le routage au niveau mot impliquent la réduction du nombre de bits de configuration et par conséquent des latences de configuration. Cependant, ces avantages sont obtenus au prix de la flexibilité et confinent les architectures à gros-grain dans des domaines applicatifs restreints.

Les architectures Garp [62] et *Pipelined Configurable Gate Array* (PiCoGA) [98] sont représentatives d'une augmentation de la granularité par l'utilisation de canaux de routage de largeur 2. Ces architectures se situent à mi-chemin entre le grain-fin et le gros-grain et sont qualifiées de grain moyen. Cependant leurs cellules de calcul fondées sur des LUT sont peu adaptées à la mise en œuvre d'opérateurs arithmétiques e.g les multiplieurs. Dans ce cas l'utilisation d'opérateurs dédiés s'avère indispensable pour augmenter les performances.

L'architecture KressArray est une des premières architectures reconfigurables à gros-grain [60]. Elle est composée de ressources de calcul, nommées *reconfigurable DataPath Units* (rDPU), agencées dans une matrice nommée *reconfigurable DataPath Array* (rDPA). La figure 2.3 donne les éléments principaux de l'architecture.

Le réseau de routage est hiérarchique avec deux niveaux d'interconnexion : lignes courtes pour des connexions locales entre rDPU et lignes longues pour des connexions globales. Chaque rDPU est composée d'une UAL 32-bits dont les entrées et les sorties sont tamponnées dans

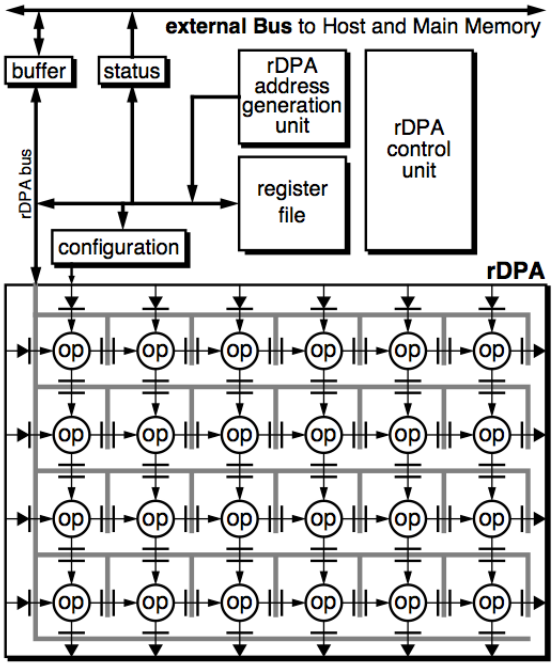


FIG. 2.3 – Architecture du KressArray [60].

des registres. L'exécution est dirigée par les données (*data driven*), c'est-à-dire, que l'opération est déclenchée lorsque tous les opérandes sont disponibles en entrée. L'alimentation en donnée du rDPA et sa configuration sont gérées par l'unité de contrôle (*rDPA control unit*). Chaque rDPU est adressable individuellement à travers une unité de génération d'adresse (*rDPA address generation unit*). Les données sont transférées sur le bus du rDPA depuis le bus externe ou le banc de registre qui peut jouer le rôle de cache.

L'architecture Systolic Ring a pour vocation d'être intégrée dans un SoC [127]. Une de ses particularités principales est sa topologie en anneau de nœuds de calcul pipelinés, comme illustrée par la figure 2.4. L'élément central est un processeur hôte *Reduced Instruction Set Computer* (RISC) qui joue le rôle de contrôleur de configuration global. Les nœuds de calcul, appelées *Data node* (Dnode), contiennent chacun une UAL 16 bits, un multiplieur, un banc de registres et une unité de contrôle (voir figure 2.4). Un Dnode a deux modes de reconfiguration dynamique : global ou local. Dans le premier cas, les unités sont configurées par l'exécution de microinstructions au niveau du contrôleur de configuration. Dans le deuxième cas les Dnode sont configurés localement par l'exécution des microinstructions au niveau de leur unité de contrôle. Ce dernier mode diminue la complexité du processeur RISC dans le cas d'une architecture avec un nombre de nœuds élevés (e.g. 256 Dnode).

L'architecture PiCoGA-III du processeur reconfigurable DREAM est une évolution de l'architecture PiCoGA vers une granularité supérieure [27]. Elle est composée d'une unité de contrôle qui active les étages d'une matrice d'éléments de calcul appelés *Reconfigurable Logic Cell* (RLC). Elle gère également la reconfiguration dynamique de la matrice sur un mode multi-contextes (voir section 2.4.1). Chaque étage met en œuvre un étage d'un graphe flot de données et tamponne les sorties des opérateurs pour créer un *pipeline*. Un RLC est composé d'une UAL et d'une LUT de 64 bits qui prennent en entrée deux opérandes de largeur 4 bits

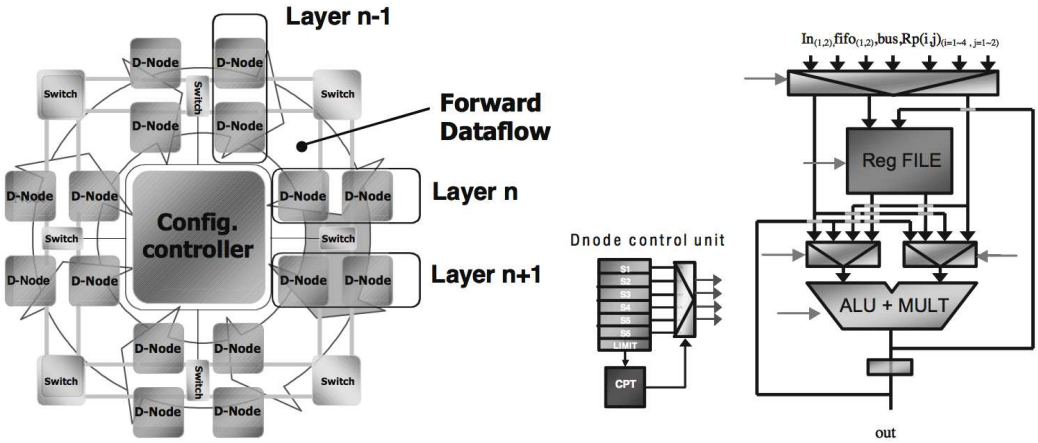


FIG. 2.4 – Architecture Systolic Ring [127]. La topologie est un anneau de Dnode groupés en *layers* pipelinés. Ils sont traversés par un flot de données circulaire à double-sens (*forward* et *feedback*).

(voir figure 2.5). Similairement au PiCoGA, les RLC sont interconnectées par des canaux de routage de largeur 2 bits.

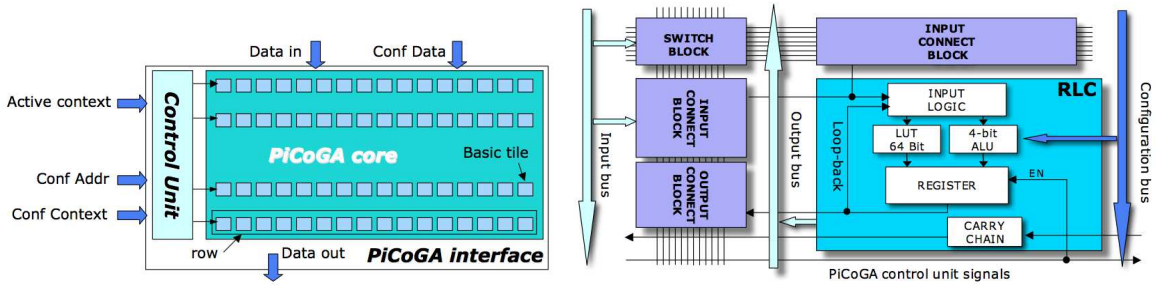


FIG. 2.5 – Architecture de PiCoGA-III [27]. Les RLC sont organisés en étages pipelinés pour la mise en œuvre de DFG.

Un autre exemple d'architecture structurée en étage est PipeRench [50]. Chaque étage est composé d'un ensemble de cellules de calcul, ou *Processing Element* (PE), connectées aux étages adjacents. Un PE contient une UAL (mise en œuvre par des LUT), des registres pour tamponner les données et une chaîne de propagation de retenue. La granularité des PE est paramétrique, cependant les meilleures performances sont obtenues pour une largeur de 8 bits et 16 PE par étages. L'originalité de cette architecture réside dans sa capacité à virtualiser ses étages d'opérateurs. Chacun d'entre eux est reconfigurable dynamiquement, ce qui permet l'exécution de plusieurs étages différents d'un graphe de flot de données (*Data Flow Graph* (DFG)) sur les mêmes PE par multiplexage temporel. Ainsi, ce mode de reconfiguration partielle rend possible la mise en œuvre de DFG dont la profondeur dépasse le nombre de ressources matérielles disponibles.

De part l'augmentation des densités d'intégration [112], certains travaux s'orientent vers une montée en complexité et en granularité des cellules de calcul. Cette tendance est illustrée par une formule de D. Patterson qui considère que « les processeurs sont les transistors de

demain » [118]. Par exemple, l'architecture RAW [133] est structurée par une matrice à deux dimensions de tuiles dont chacune contient des routeurs de communication, un processeurs RISC 32 bits à 8 étages pipelinés de type MIPS, une unité de calcul en virgule flottante, un cache de données de 32 Ko et un cache d'instructions de 96 Ko. Ainsi, la granularité des applications varie de l'opérateur arithmétique jusqu'au processus. Les tuiles sont programmables par des langages haut-niveau conventionnels (C ou Java) et les communications sont réalisées via des librairies (e.g. MPI). Ceci préserve l'héritage logiciel des processeurs généralistes.

2.3 Couplage processeur

L'architecture définie par le couplage d'un processeur avec une architecture reconfigurable est qualifiée de *processeur reconfigurable*. L'objectif est de combiner la flexibilité et les performances des circuits reconfigurables à la programmabilité des processeurs embarqués.

Le degré de couplage d'une architecture reconfigurable avec un processeur est principalement caractérisé par son intégration dans le flot d'exécution du processeur et son type d'accès mémoire. Nous présentons deux types de couplage allant du plus fort au plus faible qui sont : l'unité fonctionnelle et le coprocesseur.

2.3.1 Couplage fort : unité fonctionnelle

Le couplage de type *unité fonctionnelle* met en œuvre le concept d'*instruction set metamorphosis* qui offre une alternative à l'accélération par ASIC et un moyen de spécialisation des processeurs [9]. Le principe est d'étendre le jeu d'instruction du processeur par des fonctions matérielles qui correspondent aux portions de codes intensives de l'application. Les fonctions à accélérer sont mises en œuvre sur l'unité reconfigurable. Celle-ci est intégrée dans le chemin de données du processeur similairement à une UAL traditionnelle. L'activation de l'unité est contrôlée par des instructions spécifiques ajoutées au processeur. Les transferts de données sont réalisés via les registres du processeur, assurant un accès rapide au détriment de la bande passante disponible. Ce type de couplage implique généralement des matrices reconfigurables de petites tailles pour des tâches de faible granularité.

P-RISC est une des premières architectures à intégrer une unité fonctionnelle reconfigurable [126]. Elle est composée d'un processeur RISC (MIPS) qui embarque une architecture reconfigurable à grain-fin qualifiée d'unité fonctionnelle programmable. La figure 2.6(a) illustre le couplage de l'unité reconfigurable avec le processeur. L'unité est une matrice de LUT interconnectées par un réseau de routage similaire au FPGA (voir figure 2.6(b). Elle met en œuvre des fonctions combinatoires limitées à deux entrées et une sortie. Cette restriction définit un interfaçage avec le processeur similaire aux autres unités fonctionnelles et facilite également l'extraction des fonctions par le compilateur. La latence d'exécution d'une fonction combinatoire impacte la fréquence du processeur. De ce fait, la profondeur des fonctions est limitée à un maximum de 20 niveaux de portes logiques.

Dans le cas de processeurs spécialisés pour un domaine applicatif (e.g. processeur DSP) l'unité reconfigurable apporte une flexibilité supérieure. L'architecture XiRisc [99] est composée d'un cœur de processeur RISC de type VLIW (voir figure 2.7). Il couple des unités fonctionnelles orientées vers le traitement du signal (e.g. MAC) à une unité reconfigurable à grain-fin pipelinée et nommée PiCoGA [98]. Ainsi, le processeur met en œuvre deux flots d'exécution indépendants auxquels s'ajoute le flot de l'unité reconfigurable. En comparaison

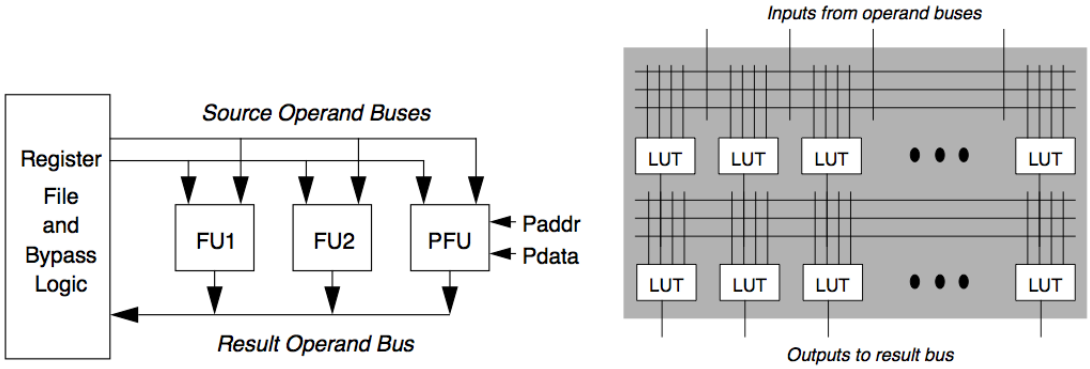


FIG. 2.6 – (a) Intégration de l'unité reconfigurable comme unité fonctionnelle du processeur. (b) Structure de l'unité reconfigurable [126].

de P-RISC, XiRisc autorise des extensions plus complexes dont les latences sont de plusieurs cycles. De plus, une extension bénéficie d'un maximum de quatre entrées et de deux sorties. La cohérence des accès au banc de registres est garantie par un système de verrous lors d'opérations d'écriture.

Parmi les autres processeurs reconfigurables qui utilisent un couplage de type unité fonctionnelle on peut citer Chimaera [61], ConCISe [75] et OneChip [144].

Il est intéressant de noter que le successeur de *OneChip* (*OneChip-98*) adresse la limitation des accès mémoire de l'unité par un accès indépendant tel que développé dans les couplages de type coprocesseur [71]. En effet, de part la complexité croissante des applications et l'augmentation de la charge de calcul affectée aux unités reconfigurables, le couplage de type unité fonctionnelle atteint ses limites. Les contraintes matérielles liées à ce couplage imposent des unités reconfigurables de faible capacité avec une bande passante limitée par des échanges à travers le banc de registres. Ces deux problématiques sont adressées par le couplage de type coprocesseur qui favorise des matrices de grandes tailles (e.g. utilisation de FPGA commerciaux) avec une capacité d'accès direct à la mémoire.

2.3.2 Couplage faible : coprocesseur

Le couplage coprocesseur apporte une autonomie plus forte de l'architecture reconfigurable en comparaison de l'unité fonctionnelle. L'exécution d'une fonction accélérée est indépendante du flot d'exécution du processeur. Ce type de couplage permet d'exploiter une exécution concurrente entre les deux flots d'exécution. Le degré de concurrence est modéré par la gestion des conflits d'accès mémoire et la capacité du processeur à poursuivre son exécution lors d'une accélération. Le coprocesseur a un accès indépendant à la mémoire et garde éventuellement un accès partiel à des registres spécifiques.

L'architecture Garp [26] couple un processeur MIPS à une unité reconfigurable à grain-fin de type FPGA utilisée comme coprocesseur. La figure 2.8 illustre la structure générale de Garp. Le chargement et l'exécution des configurations sur la matrice reconfigurable sont gérés par le processeur. Pour cela, le jeu d'instruction du MIPS a été étendu avec la possibilité de transférer des données entre les registres du processeur et la matrice. Lorsque la matrice est active le processeur se met en attente d'un signal de terminaison. Contrairement à P-RISC,

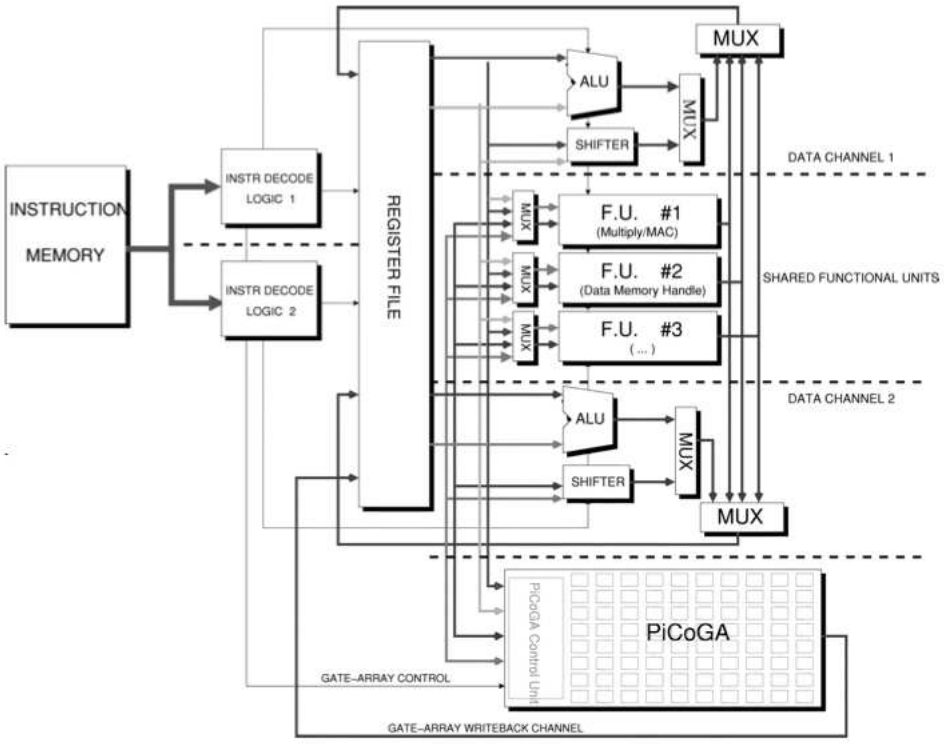


FIG. 2.7 – L'architecture du processeur reconfigurable XiRisc couple un processeur RISC VLIW à la matrice reconfigurable pipelinée PiCoGA [98].

la matrice reconfigurable bénéficie d'un accès direct au cache de données et à la mémoire externe. Cependant, le partage de la mémoire entre le processeur et l'unité reconfigurable pose des problèmes de cohérence et de synchronisation.

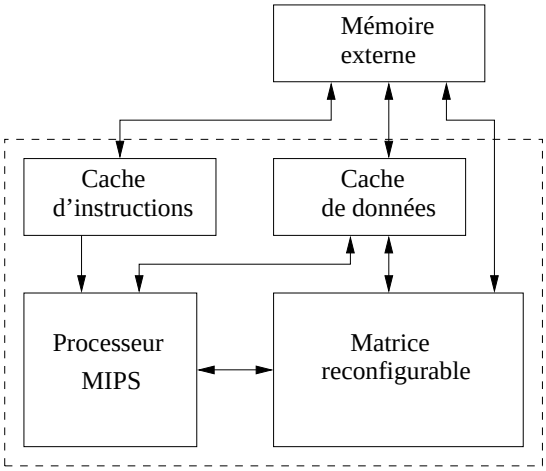


FIG. 2.8 – Vue schématique de l'architecture Garp qui couple un MIPS à une matrice reconfigurable *custom* à grain-fin. La matrice reconfigurable est un coprocesseur avec un accès direct au cache de données et à la mémoire principale [26].

Pour diminuer la dépendance de l'unité reconfigurable vis à vis du processeur et augmenter la bande passante, des architectures comme Montium [129] ou DREAM [27] associent des mémoires locales à l'unité. Chaque mémoire est couplée à un générateur d'adresses qui alimente en parallèle la fonction accélérée. D'autres architectures telles que PACT XPP [11] ou PipeRench [50] consomment directement le flot de données sans réorganisation.

Le couplage de type coprocesseur est très utilisé dans le cadre des SoC et s'insère dans un flot de conception orienté IP. Les SoC qui intègrent une ou plusieurs unités reconfigurables sont qualifiés de SoCs reconfigurables (RSoC). Un exemple de RSoC est l'architecture MORPHEUS [142, 124] dont l'organisation globale est donnée par la figure 2.9.

MORPHEUS intègre trois unités reconfigurables de différentes granularités qui sont le eFPGA M2000 [3], le processeur reconfigurable DREAM [27] et l'architecture à gros-grain PACT XPP [11]. Elles sont contrôlées par un processeur ARM de type RISC. L'ensemble des synchronisations et le passage de paramètres entre le processeur et les unités reconfigurables sont réalisés suivant le paradigme Molen [140].

Le processeur est également en charge d'alimenter en données les unités reconfigurables. Chaque unité est couplée à un ensemble de mémoires locales qui contiennent les données prises en entrée des calculs et mettent à disposition du système les résultats produits. Le processeur programme le contrôleur *Direct Memory Access* (DMA) pour des transferts sur le bus AMBA ou bien le *Direct Network Access* (DNA) pour le réseau sur puce. La reconfiguration dynamique des unités est gérée par un contrôleur de configuration (PCM) qui allège la charge du processeur.

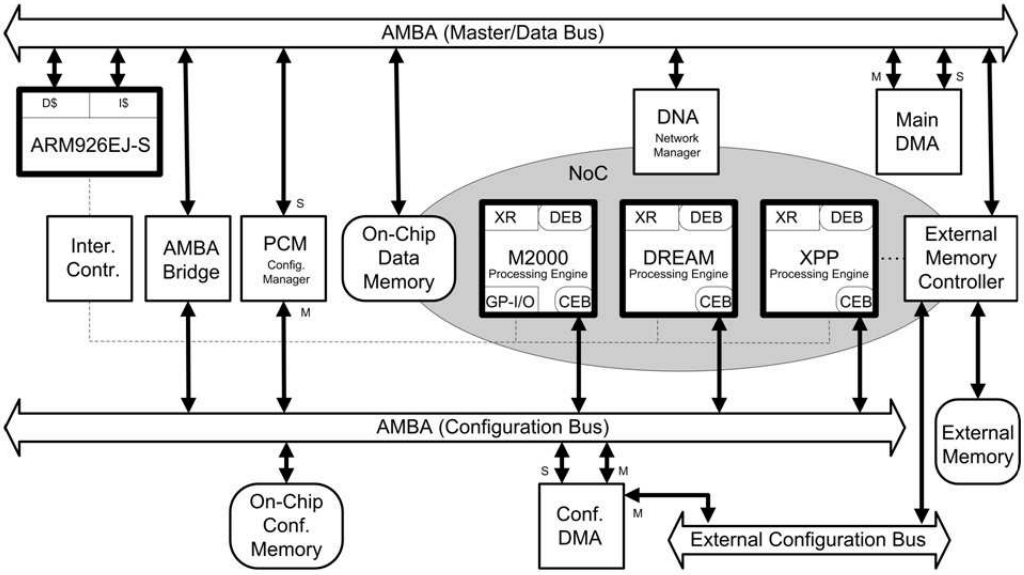


FIG. 2.9 – Vue simplifiée de l'architecture du RSoC MORPHEUS. Le processeur embarqué ARM est couplé à trois coprocesseurs reconfigurables qui sont le eFPGA M2000, le processeur reconfigurable DREAM et l'architecture à gros-grain PACT XPP [142].

2.4 La reconfiguration dynamique

On distingue deux types de configuration qualifiés de *statique* ou *dynamique*. La première implique une phase de chargement de la configuration puis une phase d'exécution lorsque la précédente est terminée. Lorsqu'une nouvelle configuration est réalisée, l'exécution en cours (si elle existe) est interrompue, l'architecture est réinitialisée puis l'opération de chargement débute. Ce mode de reconfiguration ne conserve pas l'état des mémoires entre deux étapes de chargement de la configuration.

Le terme *reconfiguration dynamique* définit la capacité de reconfigurer à l'exécution une architecture reconfigurable sans réinitialisation complète. Ainsi, entre deux étapes de configuration l'état des mémoires est conservé. Cette opération est un support pour la mise en œuvre d'applications qui nécessitent un nombre de ressources supérieur à celles disponibles. En effet, elle permet le multiplexage temporel des ressources de l'architecture.

La figure 2.10 illustre un flot de reconfiguration dynamique d'une architecture. Similairement au concept de mémoire virtuelle [65] le *bitstream* d'une application est partitionné en pages. Celles-ci sont configurées successivement en fonction d'un ordonnancement qui correspond au flot d'exécution de l'application.

Les pages de configuration qui font partie d'une même application sont susceptibles d'utiliser les résultats produits par d'autres pages. Une solution répandue est l'utilisation de registres internes pour maintenir les paramètres de la page suivante [41, 123]. Une autre possibilité consiste à utiliser des mémoires externes à la matrice reconfigurable (e.g bancs de registre, mémoires *First In First Out* (FIFO), etc.) [110, 28].

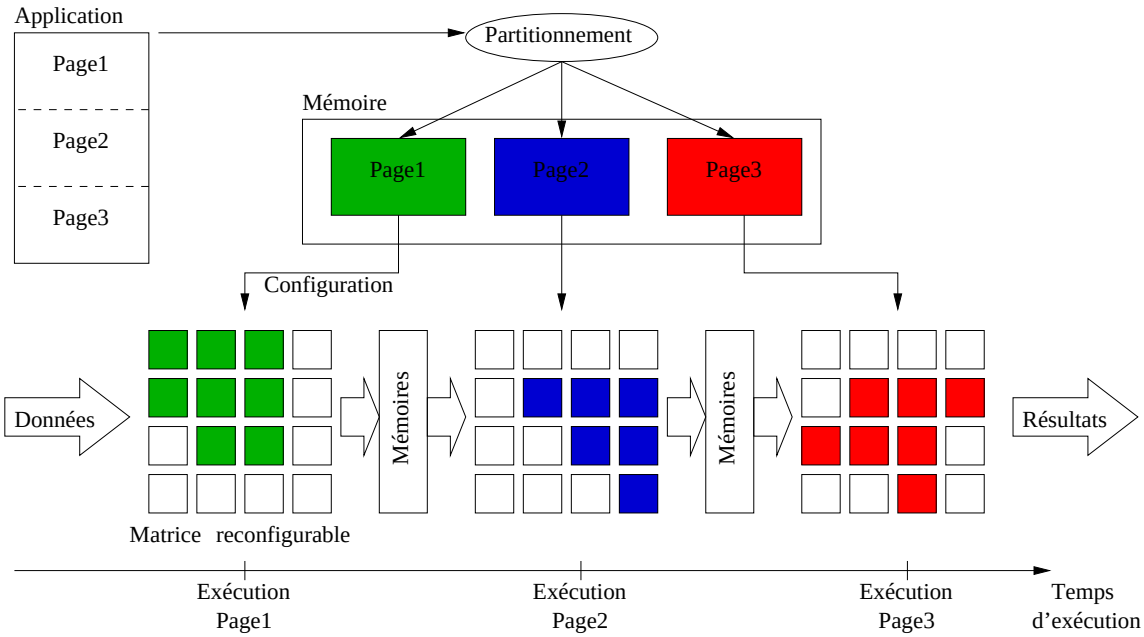


FIG. 2.10 – Reconfiguration dynamique d'une matrice reconfigurable. Une application est partitionnée en pages qui sont configurées successivement sur la matrice reconfigurable. Les transferts de données entre les pages sont réalisés à travers des mémoires.

2.4.1 Les modes de reconfiguration dynamique

2.4.1.1 Hiérarchie des mémoires de configuration et multi-contextes

Le cumul des latences de reconfiguration de la matrice reconfigurable pour l'exécution de chaque page pénalise la durée globale de l'exécution d'une application. La durée du chargement varie en fonction de la taille du contexte et de la distance de la mémoire par rapport à la matrice reconfigurable. Pour réduire les latences d'accès des hiérarchies mémoires similaires aux processeurs sont mises en œuvre. La nature et le nombre de niveaux hiérarchiques varient en fonction des architectures considérées. Dans le cas d'un RSoC [125] on peut distinguer trois niveaux pour le stockage des binaires de configuration, comme schématisé par la figure 2.11.

Le premier niveau correspond à une mémoire de configuration couplée faiblement à la matrice. Les transferts depuis cette mémoire sont les plus coûteux et sont réalisés à travers un bus de configuration. Le deuxième niveau est un cache de configuration placé à proximité de la matrice reconfigurable avec un débit élevé et des latences d'accès faibles [27]. Enfin, le dernier niveau est mis en œuvre par l'augmentation de la capacité du plan de configuration pour stocker plusieurs contextes de configuration. Ainsi, le mode de reconfiguration de l'architecture passe du *simple contexte* au *multi-contextes* [36, 138]. La configuration d'un contexte intervient en parallèle de l'exécution et chaque contexte est activé en quelques cycles par le contrôleur de configuration. Ainsi, les latences de configuration et de lecture en mémoire sont masquées partiellement ou complètement par recouvrement temporel.

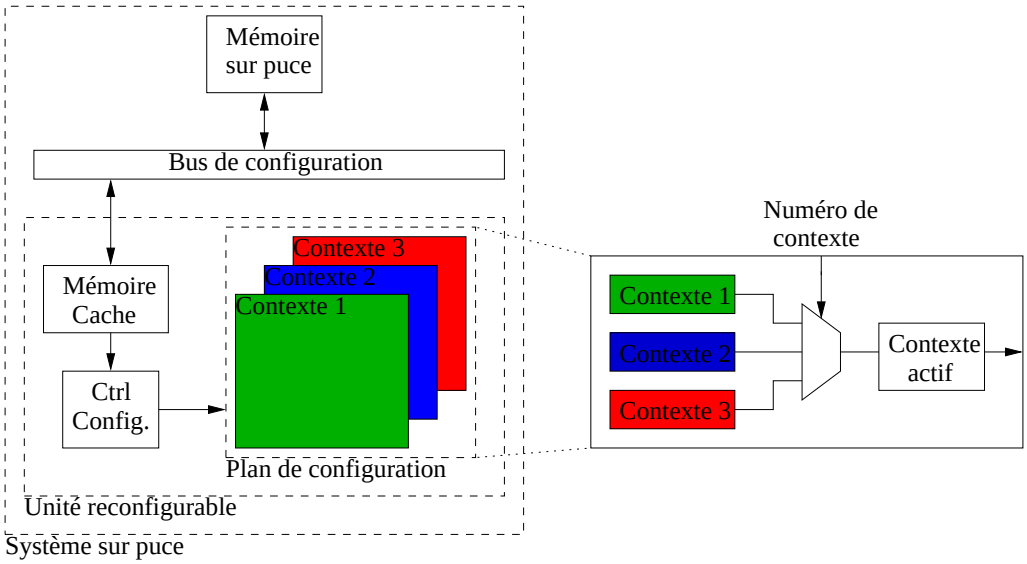


FIG. 2.11 – Hiérarchie mémoire pour le stockage des binaires de configuration ou *bitstream*.

2.4.1.2 Reconfiguration partielle

Les modes de reconfiguration simple et multi-contextes impliquent une configuration complète de la matrice et, de ce fait, l'activation d'un seul contexte à la fois. Or, ce type de configuration limite l'utilisation optimale des ressources et les capacités d'adaptation des applications. En effet, elle ne permet pas l'allocation dynamique de ressources non-utilisées par le contexte actif à un autre contexte. Il est également impossible de modifier partiellement une application pour des besoins d'adaptation à de nouvelles contraintes (par exemple, la substitution d'un filtre par un autre dans une chaîne de traitement du signal).

La reconfiguration partielle adresse ces problématiques et permet de configurer indépendamment des sous-ensembles du plan de configuration. Ainsi, le plan de configuration est divisé en zones adressables par le contrôleur de configuration. La granularité des zones varie d'une cellule de base à des groupements de cellules en fonction des architectures. De la même manière que le *multi-contextes*, le chargement d'un contexte n'impacte pas les autres parties de la matrice et permet un recouvrement entre l'exécution et la configuration. Les latences de configuration sont d'autant plus réduites que les tailles des contextes sont diminuées par rapport à une configuration complète.

La figure 2.12 illustre un exemple de reconfiguration partielle d'une matrice reconfigurable pour deux applications. L'application A est composée des deux contextes A et A' qui correspondent à deux modules connectés. L'application B est représentée par le contexte B. La première étape correspond à la configuration de ressources inutilisées pour augmenter le taux d'occupation de la matrice. Ainsi, les applications A et B s'exécutent en parallèle sans interférences. L'étape 2 montre la reconfiguration partielle de l'application A avec l'activation du contexte A" à la place de A'. Dans ce dernier cas, le remplacement d'un module connecté à un autre est plus ou moins complexe suivant la granularité de l'architecture. Par exemple, cette opération sur un FPGA du commerce (e.g. Xilinx Virtex) nécessite une phase de re-routage dynamique des modules [67, 117], tandis que certaines architectures à gros-grain offrent des supports matériels dédiés (e.g. communications inter-modules par mémoires et infrastructure

de routage spécialisée) [59, 50, 13].

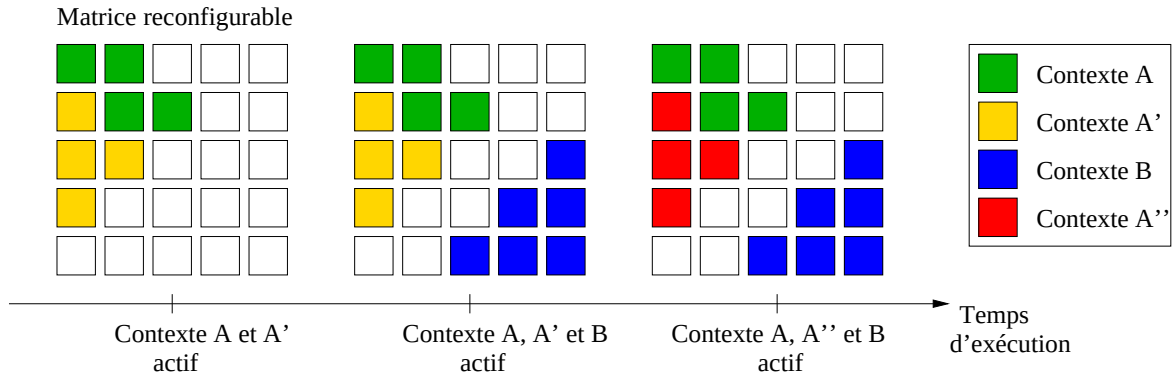


FIG. 2.12 – Reconfiguration partielle d'une matrice reconfigurable.

2.4.2 Impact des modes de reconfiguration sur l'exécution

L'opération de reconfiguration dynamique est décomposée en trois étapes principales (voir figure 2.11) : (1) lecture du binaire de configuration en mémoire et écriture dans le cache de configuration ; (2) configuration par le contrôleur ; (3) activation et exécution du contexte. La figure 2.13 donne trois exemples de flot d'exécution en fonction des modes de configuration supportés.

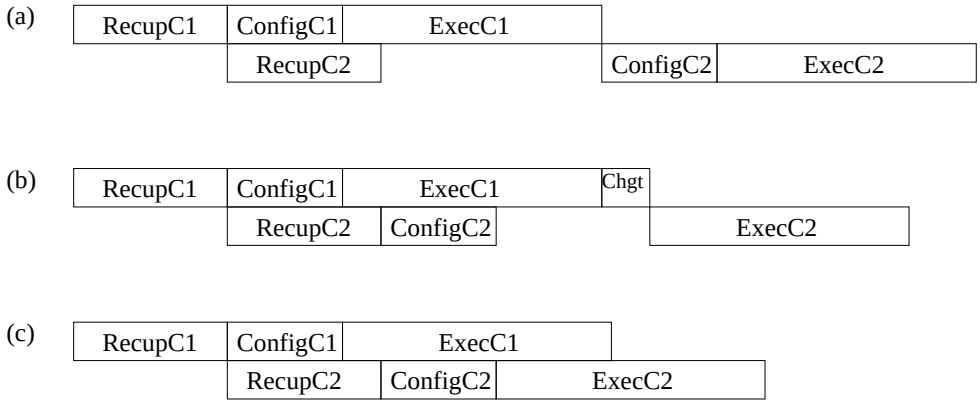


FIG. 2.13 – Scénarios de reconfiguration dynamique d'une architecture.

Le scénario (a) correspond à une reconfiguration dynamique classique. L'écriture du contexte C2 dans le cache intervient en parallèle de la configuration et de l'exécution de C1. Un sur-coût provient de l'étape de configuration réalisée avant chaque exécution. Le scénario (b) illustre une reconfiguration dynamique multi-contextes. La configuration du contexte C2 est réalisée en parallèle de l'exécution de C1. Ce mode permet de minimiser la pénalité des latences de configuration. Cependant un seul contexte est actif contrairement à la reconfiguration partielle illustrée par le scénario (c). L'exécution de C2 démarre dès que la configuration est terminée sous réserve qu'il n'y ait pas de dépendances avec C1.

2.5 Conclusion

Evolution du domaine. Le domaine des architectures reconfigurables a subi une évolution et une diversification importante depuis l'introduction du premier FPGA dans les années 80. On constate une utilisation généralisée des solutions reconfigurables dans le domaine du traitement du signal où les architectures à gros-grain (*Coarse-Grained Reconfigurable Architecture* (CGRA)) offrent des solutions performantes. En effet, la granularité des ressources est passée du bit à des tailles de mot classiquement utilisés dans les processeurs généralistes. Les CGRA sont généralement couplées à un processeur qui assure les opérations de contrôle, les fonctions arithmétiques intensives étant placées sur la matrice reconfigurable. Les débits élevés nécessaires au calcul sont assurés par un couplage faible à travers des mémoires locales. Les performances et la densité des CGRA en font des candidats de choix comme unités de SoC. Cependant leurs ressources sont spécialisées et peu flexibles en comparaison du grain-fin.

Les FPGA commerciaux viennent contrebalancer la forte spécialisation des CGRA par une hétérogénéité des ressources. Par exemple, le Virtex 6 se décline en plusieurs familles avec un ratio d'opérateurs à gros-grain et à grain-fin variable en fonction du domaine applicatif ciblé. A l'inverse des CGRA, ces architectures de très grande taille ne s'embarquent pas dans un SoC mais peuvent servir de support de prototypage. Cette solution est particulièrement intéressante dans un contexte d'augmentation croissante du prix de fabrication des circuits spécifiques.

L'espace de conception. Dans ce chapitre nous avons caractérisé les architectures en fonction de leur granularité, de leur couplage processeur et des modes de reconfiguration supportés. Ce dernier paramètre est particulièrement important car il conditionne l'exécution des applications. L'analyse de ces caractéristiques permet de dégager un certain nombre d'axes représentés par la figure 2.14. Sur chaque axe est placé un ensemble de choix architecturaux ou relatifs à l'exécution qui caractérisent l'espace de conception des architectures.

La recherche d'une architecture optimale pour une application donnée consiste à explorer l'ensemble de ces paramètres. Cette tâche est qualifiée d'exploration de l'espace de conception. Le nombre de possibilités à considérer est très important et représente une tâche considérable pour les concepteurs. Pour diminuer la charge de travail et augmenter la productivité, des outils ont été développés. Ils permettent d'automatiser une grande partie des évaluations. Le chapitre suivant présente différentes méthodes d'exploration fondées sur une modélisation à haut-niveau de la cible.

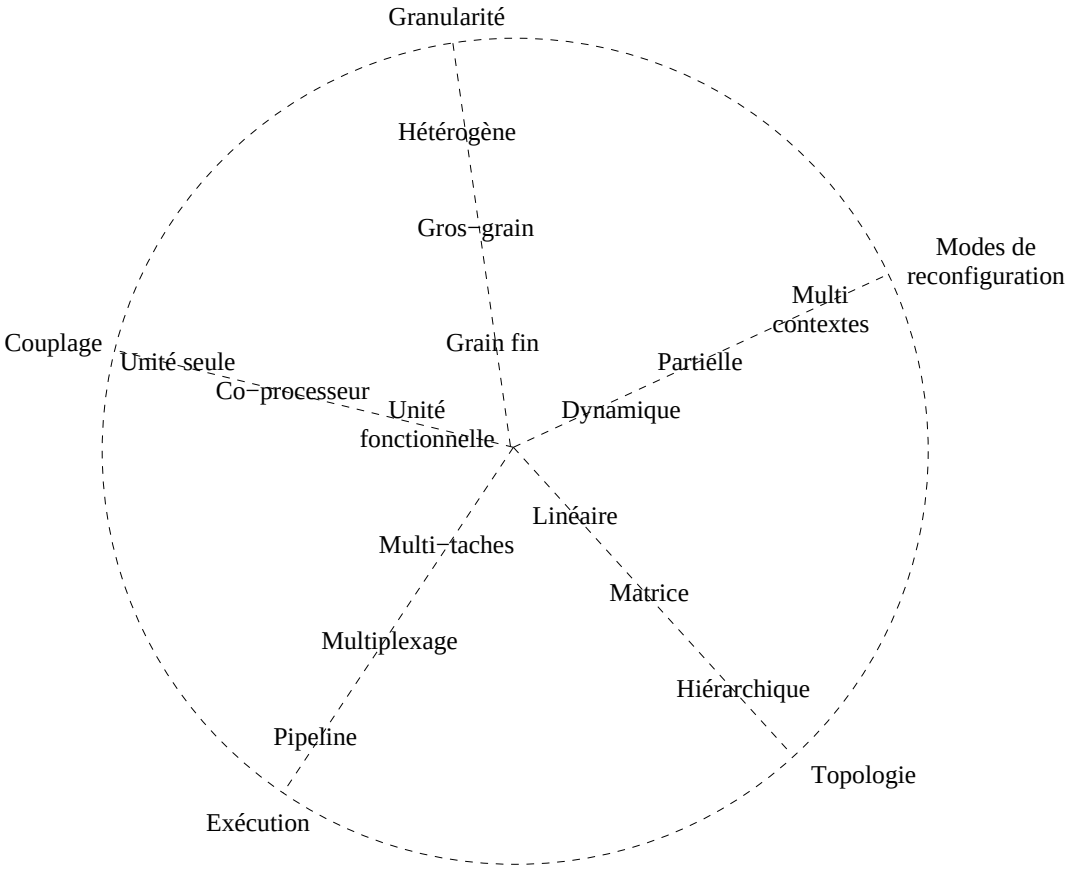


FIG. 2.14 – Caractéristiques des architectures reconfigurables.

Chapitre 3

Flot d'exploration centré sur un langage de description d'architectures

3.1 Introduction

Les avancées technologiques des semi-conducteurs en terme de taux d'intégration permettent la production de circuits de plus en plus complexes et hétérogènes [112]. Intégrer des systèmes complets sur une même puce (SoC) est aujourd'hui la norme. Cependant cette complexité implique une augmentation des coûts non-récurrents de conception et de validation des circuits. A cela s'ajoute la réduction des temps de vie des circuits et de ce fait des temps de mise sur le marché. Tous ces facteurs conjugués et inter-dépendants imposent une forte productivité des concepteurs pour être compétitif.

Les solutions traditionnelles, qui imposent une grande part de développements manuels à un niveau d'abstraction bas, limitent les phases amont de recherche de solutions architecturales optimales. Ce décrochage entre la montée en complexité des circuits et la capacité des méthodes/outils de conception à les adresser est caractérisé par le terme de « fossé de productivité ». Ce fossé tend à s'agrandir au fur et à mesure des évolutions technologiques et pose de nouveaux défis aux concepteurs [42].

Dans le cadre des systèmes sur puce, cette tendance concerne autant les outils logiciels que les aspects matériels. Par exemple, le développement et la conception d'un composant reconfigurable spécifique nécessite le développement de l'architecture dans un langage de description matérielle (e.g. VHDL ou Verilog) et le développement des outils de compilation/simulation associés. Ce type de méthode *ad-hoc* et non automatisée n'est pas une solution viable pour garantir des cycles prospectifs courts et une forte productivité.

La réduction du fossé de productivité nécessite la mise à disposition d'outils et de méthodes pour explorer efficacement l'espace de conception. Ces outils doivent permettre d'une part une exploration rapide à un haut niveau d'abstraction de l'espace de conception matériel et d'autre part la génération automatique du cadre d'outils logiciels (simulateurs, compilateurs/synthétiseurs, etc.). Pour répondre à ces besoins, des méthodes d'exploration fondées sur une modélisation à haut-niveau de la cible ont été développées. Ce type d'approche est très répandu dans le domaine des processeurs embarqués à jeu d'instruction spécifique (*Application-Specific Instruction-set Processor* (ASIP)). L'exploration architecturale et la génération des

outils sont réalisées à partir d'une description du processeur dans un ADL.

Pour préciser le principe d'un flot ADL nous présentons dans un premier temps son application aux ASIP dans un contexte SoC. Les caractéristiques principales des ADL sont ensuite détaillées. Puis une transposition aux domaines des architectures reconfigurables est effectuée. Le flot présenté a pour vocation de prendre en charge des architectures de différentes granularités, topologie et modes de reconfiguration dynamique. Pour déterminer l'existence actuelle d'un tel flot générique, les méthodes existantes et représentatives d'exploration des architectures reconfigurables seront détaillées.

3.2 Du flot ADL pour les ASIPs aux architectures reconfigurables

3.2.1 Principe général dans un cadre SoC

La figure 3.1 illustre un flot traditionnel de conception conjointe matériel-logiciel pour un système sur puce (SoC). Le flot prend en entrée une application qui, après profilage, est partitionnée entre les éléments matériels (e.g. ASIC, unités reconfigurables) pour les tâches intensives, et les éléments programmables (e.g. processeur, DSP) pour le logiciel. Dans un flot classique la structure du SoC est prédéfinie et ses composants (IP) ainsi que leurs outils de programmation sont extraits de librairies pré-caractérisées [103]. En effet, étant donnés les coûts et les durées de conception il est impératif de favoriser la réutilisation. Dans le cas des processeurs ou des unités reconfigurables embarquées, des composants prédéfinis paramétriques sont utilisés. Cependant ils se restreignent à un cadre de spécialisation limitée à une architecture particulière i.e. les variations sont opérées sur un squelette d'architecture rigidifié. Or, dans le cadre des SoC, l'obtention de la performance et de la flexibilité requiert une spécialisation fine des unités programmables mais au coût d'un redéveloppement long et fastidieux, donc onéreux.

Pour répondre aux objectifs de productivité énoncés précédemment, des travaux ont développé des flots d'exploration de l'espace de conception centrés sur l'utilisation de ADL [31]. Ces langages servent à spécifier une architecture à un haut-niveau d'abstraction pour en produire un modèle manipulable par les outils. Le modèle permet la génération automatique de prototypes matériels et de l'environnement logiciel de programmation de l'architecture. La figure 3.2 illustre un flot général d'exploration de l'espace de conception centré sur une modélisation par un ADL.

Dans le contexte des SoC, ces flots d'exploration sont majoritairement utilisés pour les processeurs embarqués [109, 107, 137]. La figure 3.1 situe l'utilisation de la description ADL dans un flot traditionnel de conception de processeur embarqué.

La génération automatique d'un environnement complet de prototypage permet d'obtenir des cycles de prospection itératifs courts. Les changements architecturaux sont réalisés rapidement à haut-niveau et leurs impacts sont quantifiés par le retour des résultats de simulation.

3.2.2 Les langages de description d'architectures

Les ADL sont utilisés dans les domaines du logiciel et du matériel pour maîtriser la complexité croissante des systèmes. Dans les deux domaines, la description d'une architecture

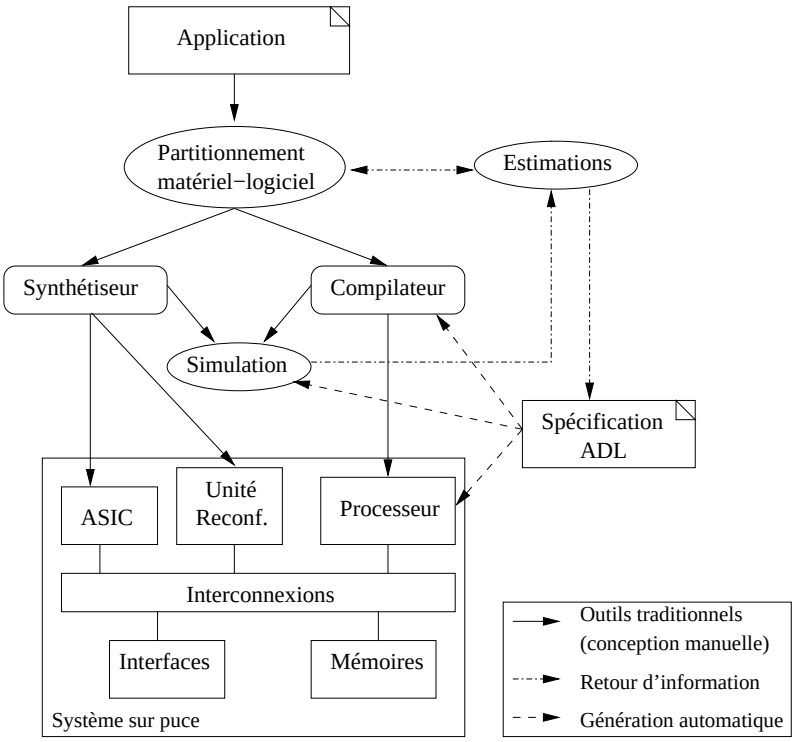


FIG. 3.1 – Flot traditionnel de conception conjointe pour un système sur puce [109].

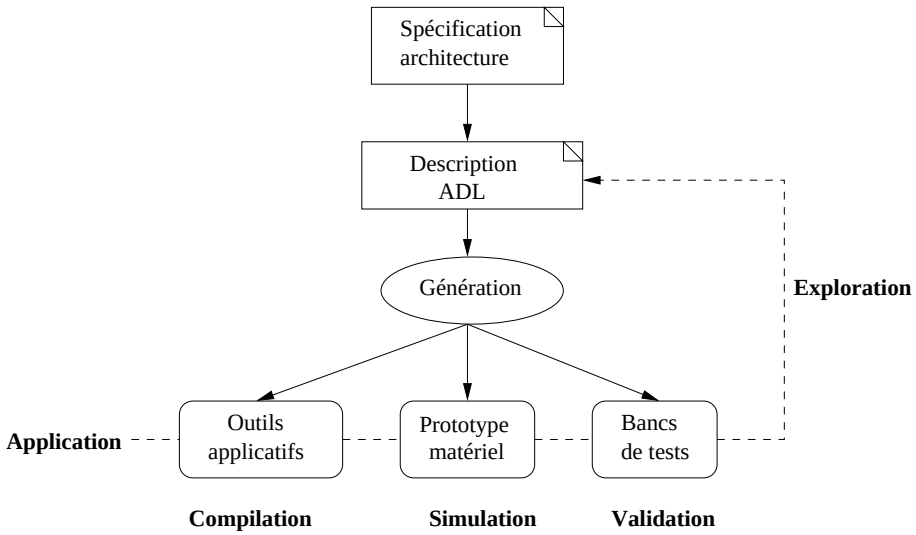


FIG. 3.2 – Flot d'exploration de l'espace de conception centré sur un ADL. La génération automatique du prototype matériel et de son environnement de programmation permet d'obtenir des cycles d'exploration itératifs courts.

consiste à définir un ensemble de composants dont les comportements ainsi que l'ensemble des mécanismes d'interactions sont spécifiés dans le langage. La différence majeure réside dans la nature des composants manipulés.

Dans le cas d'un logiciel les composants peuvent être des processus, des fonctions ou d'autres entités logicielles présentes dans les programmes. Les interactions entre les composants sont généralement des appels fonctionnels ou encore un passage de valeur par variable [31]. En revanche dans le cas d'une architecture matérielle les composants sont des entités matérielles (e.g. unités de calcul, processeur) interconnectées par des bus et dont le comportement se situe à un niveau circuit (e.g. jeu d'instruction d'un processeur). Les interactions entre les composants correspondent à des envois de signaux binaires sur les bus d'interconnexion. Dans ce dernier cas l'ADL se rapproche d'un langage de description matériel (e.g. VHDL) tout en gardant un niveau d'abstraction élevé.

La distinction entre les ADL et les langages de programmation, de description matérielle et de modélisation, n'est pas ferme et définitive [108]. L'orientation d'un ADL vers un type de langage dépend de la nature de son contenu. Pour le domaine des ASIP, trois types sont définis en fonction de l'orientation de la description : structurelle, comportementale ou mixte [108]. Pour illustrer cette classification nous présentons dans les paragraphes suivants quelques ADL représentatifs pour l'exploration des ASIP.

3.2.2.1 Description structurelle

MIMOLA est orienté vers une description à un niveau RTL de la structure du processeur. Cette approche capture le comportement et la structure du circuit tout en masquant la mise en œuvre au niveau porte. Cependant, à ce niveau de détail, la complexité de la description a tendance à augmenter conjointement avec celle de l'architecture modélisée. En effet, similairement à VHDL, pour chaque composant du processeur une description est nécessaire.

L'environnement de modélisation MIMOLA [96] est orienté vers la conception micro-architecturale d'architectures programmables ou non. Il est composé d'un synthétiseur de haut-niveau (MSSH), d'un générateur de microcode (MSSQ), d'un compilateur de test (MSST) et d'un simulateur algorithmique (MSSB) et RTL (MSSU). Tous ces outils opèrent sur une représentation intermédiaire (TREEMOLA) générée à partir d'une description dans l'ADL MIMOLA. Une description est composée de trois parties qui sont : l'algorithme du microprogramme pour le contrôle du processeur, sa description structurelle et des règles optionnelles de liaison/transformations.

La partie algorithmique est décrite dans une variante du langage PASCAL. Des extensions orientées vers le matériel sont disponibles telles que : la référence à des mémoires physiques (e.g. registres), adressage au bit près, appel fonctionnel à des composants matériels, etc. La partie matérielle est décrite sous la forme d'une *netlist* de *modules* interconnectés. Chaque module est défini par son interface, qui déclare ses ports, et son comportement.

Le code 3.1 donne un exemple de déclaration d'une UAL. La construction *CONNECTION* connecte la sortie de l'UAL à un accumulateur qui lui-même est connecté à *in1*. La règle de liaison définie par *LOCATION_FOR_PROGRAMCOUNTER* permet au générateur de code de localiser le compteur de programme. La règle de remplacement, *REPLACE_ALWAYS*, s'applique sur l'algorithme et contraint la réécriture de portions de code. Cela permet d'utiliser dans l'algorithme des opérateurs non-présents dans l'architecture (e.g. un multiplieur).

Les informations du jeu d'instructions (encodage et syntaxe) du processeur modélisé sont définies implicitement et doivent être extraites à partir de la *netlist*. Or, la flexibilité du

```
1  MODULE Alu(IN i1, i2: (15:0) ; OUT outp: (15:0);
2  IN ctr: (1:0));
3  CONBEGIN
4    outp <- CASE ctr OF
5                    0: i1 + i2;
6                    1: i1 - i2;
7                    2: i1 AND i2;
8            END;
9  CONEND;
10
11 CONNECTIONS Alu.outp -> ACCU.inp; ACCU.outp -> Alu.i1;
12
13 LOCATION_FOR_PROGRAMCOUNTER PCReg;
14 REPLACE_ALWAYS &a * 2 WITH &a + &a;
```

Listing 3.1 – Exemple de description MIMOLA.

niveau structurel RTL complexifie la tâche des outils. Pour faciliter l'extraction et augmenter le niveau d'abstraction certains ADL, comme nML [44], sont orientés vers une description explicite des instructions.

3.2.2.2 Description comportementale

Le langage nML [44] est orienté vers la description de la couche *Instruction Set Architecture* (ISA) du processeur par une capture sémantique des instructions. Par rapport à MIMOLA, les détails structurels de l'architecture sont moins précis qu'au niveau RTL. Ainsi, le langage établit une correspondance quasi directe entre le manuel ISA de référence et la description ADL. nML est utilisé par les environnements d'exploration CHESS/CHECKERS [51] et CBC/-SIGH/SIM [43]. Il est également disponible dans une version commerciale [32].

Les descriptions architecturales dans le langage nML sont fondées sur une structuration hiérarchique des instructions. Le principe est inspiré de la notion d'héritage objet avec une factorisation au plus haut niveau hiérarchique des comportements communs. Cette organisation permet de réduire la complexité du jeu d'instruction et également le nombre de descriptions nécessaires. Le plus bas niveau correspond à des portions d'instructions (e.g. addition, opérations logiques, etc.) et sont qualifiées d'*instructions partielles* (IPs). Elles sont destinées à être composées dans les niveaux supérieures suivant deux règles : AND et OR. La première règle définit un groupement d'IPs et la deuxième correspond à une alternative entre plusieurs IPs. Le résultat de la description est un arbre de composition d'IPs dont la traversée de bas en haut forme une instruction complète.

Le code 3.2 montre un exemple de description nML. Les deux IPs *computemove* et *jump* son combinées par une règle OR dans *instruction*. Ainsi, *instruction* peut prendre la valeur de l'une des deux. L'IP *computemove* combine, par une règle AND, les deux IPs *compute* et *move* en les incluant dans ses paramètres. De ce fait, l'exécution de *computemove* implique l'exécution des deux IPs. Dans cet exemple, l'IP *move* correspond à une feuille de l'arbre d'instructions.

Une IP est définie par trois attributs : *image* correspond à l'encodage binaire de l'instruction ; *syntax* à la syntaxe assembleur ; *action* décrit le comportement d'une IP. A partir de ces informations il est possible de générer la syntaxe et l'encodage complet du jeu d'instruction. A cela s'ajoute la structuration du processeur par l'interconnexion des instructions par trois types de mémoires : *Random-Access Memory* (RAM), registre et stockage transitoire (les

```
1  opn instruction= computemove | jump
2
3  mem M[2**24, long]
4  mem tmp1[1, long]
5  mem tmp2[1, long]
6
7  opn computemove(c:compute, m:move, a1 : card(24), a2 : card(24))
8    image = ``001"::c.image::m.image
9    syntax = format(''%s || %s', c.syntax, m.syntax)
10   action = {
11     tmp1 = M[a1];
12     c.action;
13     m.action;
14     M[a2] = tmp2
15   }
16
17 opn move()
18   syntax = ``move''
19   image = ``000001''
20   action = {tmp2 = tmp1}
```

Listing 3.2 – Exemple de description nML.

données sont maintenues pendant un nombre de cycles limité). Ce dernier type est utilisable pour la modélisation du chemin de données en étages de *pipeline*. Cependant, la modélisation de *pipeline* avec un contrôle complexe (e.g. vidage et arrêt) reste difficile.

3.2.2.3 Description mixte

Le langage *Language for Instruction Set Architecture* (LISA) [120] est orienté vers la production de simulateurs cycle-prêt et de compilateurs C reciblables. Il est disponible dans la chaîne d'outils commerciaux de CoWare (acquis par Synopsis) [131].

LISA s'inspire du principe de description hiérarchique des instructions introduit par nML. Ainsi, les comportements communs sont factorisés aux plus hauts niveaux hiérarchiques. Contrairement à nML, LISA modélise plus précisément les *pipelines* par des constructions spécifiques. Ainsi, il représente un compromis entre une orientation structurelle (MIMOLA) et comportementale (nML).

Une description LISA est composée de *ressources* et d'*opérations*. Les ressources représentent les éléments matériels tels que les registres, les mémoires et la structuration des *pipelines*. Le code 3.3 donne un exemple de déclaration des ressources d'un processeur simplifié (*RESSOURCE*). La syntaxe ainsi que le système de type sont inspirés par le langage C. Les mots-clés placés avant la déclaration des mémoires (e.g. *REGISTER*) sont utilisés pour identifier l'élément lors de phase de débogage. Dans cet exemple le processeur est composé d'un compteur de programme (*pc*), un fichier de registre (*R*) et de deux caches (*pmem* et *dmem*). Un *pipeline* (*PIPELINE*) est défini par un nom (*pipe*) et l'énumération de ses étages. Les registres associés au *pipeline* sont définis dans *PIPELINE_REGISTER*.

Les *opérations* représentent le comportement, la structure et le jeu d'instructions de l'architecture modélisée. Le code 3.3 donne un exemple de déclaration de l'opération de décodage d'instruction du processeur. La déclaration commence par une assignation à l'étage *DC* du *pipeline pipe*. Le corps est divisé en plusieurs parties. *DECLARE* correspond à des déclarations locales d'identifiants. Les opérations à exécuter sont associées à l'étage *EX* et sont identifiées par *opcode*. *CODING* et *SYNTAX* définissent l'encodage binaire de l'instruction et

```
1  RESSOURCE {
2    PROGRAM_COUNTER int pc;
3    REGISTER int R[0..31];
4
5    PROGRAM_MEMORY char pmem[0..0x100000];
6    DATA_MEMORY char dmem[0..0x100000];
7
8    PIPELINE pipe = {FE; DC; EX; MEM; WB};
9    PIPELINE_REGISTER IN pipe {
10     ir_t ir;
11     int npc, reg_a, reg_b, imm, alu;
12     REGISTER bool cond;
13   };
14 }
15
16 OPERATION i_type IN pipe.DC {
17   DECLARE {
18     GROUP opcode = {ADDI || SUBI};
19     GROUP rs1, rd = {fix_register};}
20   CODING = {opcode rs1 rd immediate}
21   SYNTAX {opcode rd ``,'' rs1 ``,'' immediate}
22   BEHAVIOR {reg_a = rs1; imm = immediate; cond = 0;}
23   ACTIVATION{opcode, writeback}
24 }
```

Listing 3.3 – Déclaration des ressources et d'une opération dans une description LISA.

la syntaxe assembleur de l'instruction (équivalent à *image* et *syntax* de nML). *BEHAVIOR* décrit le comportement de l'instruction en C. Enfin, *ACTIVATION* spécifie les opérations à activer par rapport à l'étage courant du *pipeline*.

3.2.3 Flot ADL pour les architectures reconfigurables

Les sections précédentes ont présenté le principe des flots ADL pour la conception de processeurs spécifiques embarqués. Ils offrent un moyen efficace d'exploration de l'espace de conception par la génération des outils et du prototype architectural. L'intérêt des industriels (e.g. Synopsys [130]) et la mise à disposition de produits commerciaux démontrent la crédibilité de ce type d'approche [32, 131]. Actuellement, les flots ADL sont principalement appliqués à la conception d'accélérateurs programmables de SoC. Dans cette section nous montrons comment transposer ce type de flot aux architectures reconfigurables.

Le flot de la figure 3.2 est une transposition d'un flot ADL pour les ASIP aux architectures reconfigurables. Les différences majeures entre les deux types de flots se situent au niveau de la nature des descriptions architecturales et des outils produits en sortie du flot (voir figure 3.3).

Les éléments constitutifs (e.g. LUT, routage) ainsi que l'organisation spatiale d'une architecture reconfigurable (e.g. réseau d'opérateurs) diffèrent d'un processeur. Par conséquent, cette différence est répercutée dans les moyens d'expression du langage ADL.

La figure 3.3 décrit les trois types d'outils générés automatiquement par le flot. De part leur nature spatiale similaire aux ASIC, les *back-end* applicatifs pour les architectures reconfigurables sont spécialisés pour la conception matérielle. L'ensemble des outils applicatifs est constitué d'un synthétiseur pour produire le circuit (aussi appelé *netlist*), d'un placeur-routeur pour réaliser l'allocation des ressources (calcul et routage) dans l'architecture et d'un générateur de binaires de configuration qui produit l'application finale. Celle-ci est ensuite

envoyée dans la mémoire de configuration de l'architecture reconfigurable prototypée par un contrôleur spécifique.

La simulation de l'architecture s'effectue à un niveau matériel par la génération d'un prototype en VHDL comportemental. A cela s'ajoute la génération du contrôleur de configuration qui prend en compte les différents modes de configuration préalablement spécifiés (reconfiguration partielle et multi-contextes). Ces deux éléments constituent un prototype complet de l'architecture reconfigurable qui permet d'une part d'évaluer l'architecture suivant différentes métriques (e.g. performance) et, d'autre part, d'en valider le comportement par l'exécution d'applications.

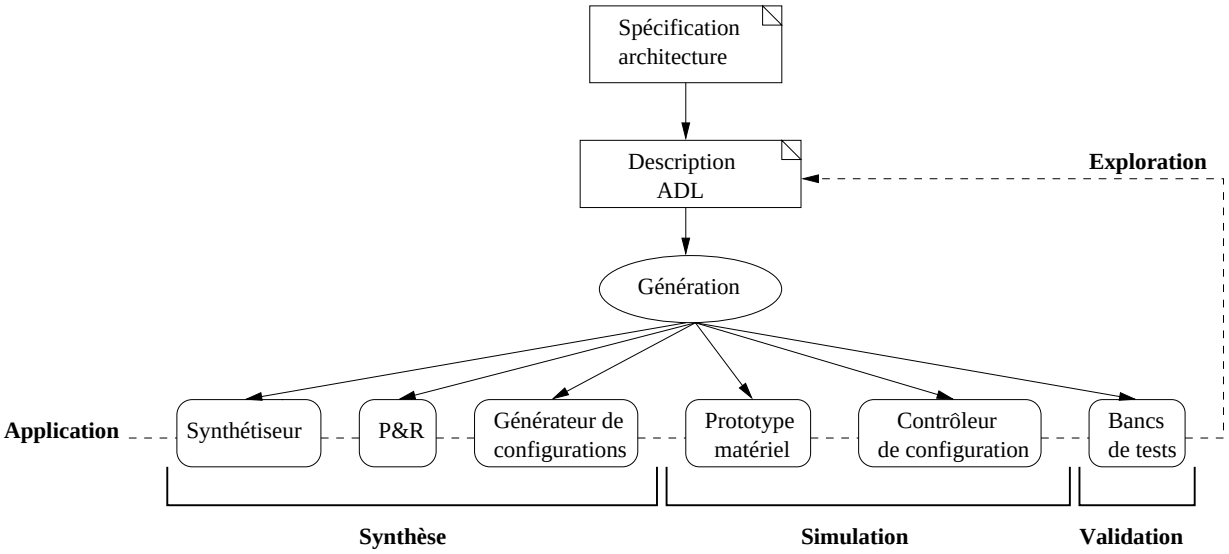


FIG. 3.3 – Flot d'exploration de l'espace de conception centré sur un ADL pour les architectures reconfigurables.

3.3 Méthodes d'exploration pour les architectures reconfigurables

La recherche d'une adéquation algorithme-architecture dans un espace de conception est réalisée en fonction de critères définis par le concepteur. Ces critères représentent les paramètres essentiels à faire varier pour obtenir une solution architecturale optimale.

La figure 3.4 montre trois axes d'exploration pour une architecture reconfigurable : application, plan de configuration, plan de calcul. Le premier axe correspond au domaine applicatif pour lequel une architecture doit être caractérisée. Le second est spécifique aux architectures reconfigurable et détermine les modes de reconfiguration disponibles (dynamique partielle et multi-contextes). Le troisième correspond aux type de ressources de calcul qui définit une adéquation avec certains types d'opérations. Pour une solution architecturale donnée, les trois critères peuvent varier indépendamment les uns des autres.

On peut distinguer trois méthodes pour réaliser l'exploration d'une architecture reconfigurable modélisée en ADL dénommées simulation, synthèse et estimations. La première est

fondée sur la simulation de l'architecture matérielle. Celle-ci est générée à partir de sa modélisation [24, 90, 29]. La production des binaires de configuration nécessite la mise à disposition d'outils spécifiques à l'architecture. Certains flots définissent des outils qui s'adaptent à une architecture paramétrique prédéfinie ou à une famille architecturale (e.g. les FPGA) [78, 24, 29]. Cependant, à notre connaissance il n'existe pas de générateur de binaire de configuration qui couvre un domaine plus vaste qu'une famille d'architectures.

A cause du temps requis pour la simulation matérielle, cette solution est utilisée dans un espace d'exploration restreint. A ce niveau d'abstraction, l'objectif est d'obtenir un maximum de précision sur une architecture dont les caractéristiques principales sont déjà définies en amont.

La deuxième méthode évalue la mise en œuvre d'une application sur un modèle d'architecture. L'application est synthétisée puis placeur-routeur pour en extraire des mesures de performance. Le niveau d'abstraction étant plus élevé, les cycles d'exploration sont raccourcis. Ce type de méthode tire également partie d'outils de mise en œuvre génériques qui prennent automatiquement en compte les variations architecturales [16, 81].

Enfin, la troisième méthode remplace la phase de synthèse par des estimations de performance [21]. Les algorithmes d'estimations sont généralement plus rapides que ceux de synthèse et permettent de gagner du temps. Cependant, le niveau d'abstraction est très haut et la précision des évaluations est inférieure à celle des deux méthodes précédentes. Cette approche est principalement utilisée pour la comparaison rapide de plusieurs architectures.

Ces trois méthodes sont complémentaires et peuvent être couplées pour explorer efficacement l'espace de conception. Elles représentent chacune une étape de raffinement dans la recherche d'une architecture optimale.

Elles sont déterminées par les résultats d'évaluation qui placent la solution dans un référentiel de métriques (e.g. surface, performance, consommation). Ce placement fournit au concepteur un critère décisionnel sur l'atteinte des objectifs fixés (figure 3.4).

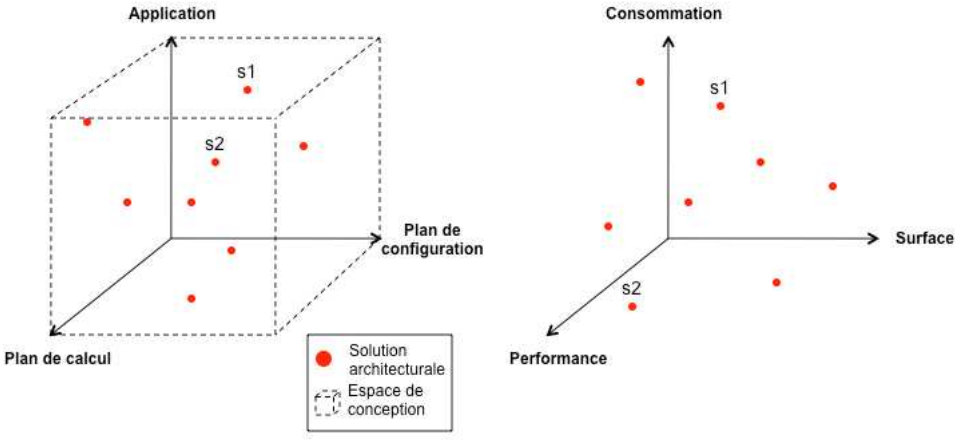


FIG. 3.4 – Représentation de l'espace de conception des architectures reconfigurables dans un référentiel à trois dimensions.

Les sections suivantes présentent plusieurs outils qui proposent différentes méthodes d'exploration. Une attention particulière est donnée à l'outil Madeo utilisé dans nos travaux [88].

3.3.1 Exemples de flots d'exploration

Les flots d'exploration diffèrent en fonction de leur spécialisation pour une architecture particulière et en fonction d'une orientation plus générique. Dans le premier cas l'exploration consiste à produire des variations architecturales pour une famille d'architectures (e.g. FPGA de type Xilinx) ou pour une architecture paramétrique préexistante. Dans le deuxième cas la modélisation offre une plus grande flexibilité et permet d'adresser différentes familles d'architectures reconfigurables (e.g granularité, topologie, etc.).

3.3.1.1 VPR/GILES

L'outil *Versatile Place and Route* (VPR) [16] est un outil générique de placement-routage spécialisé pour les FPGA. Il permet la comparaison de plusieurs architectures afin de sélectionner la plus adaptée à une application donnée. Le flot de l'outil est illustré par la figure 3.5

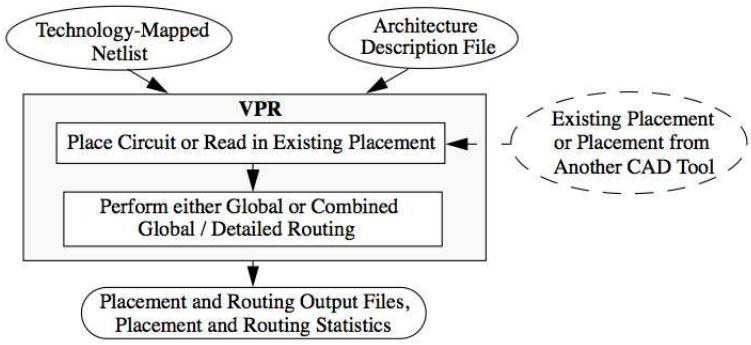


Fig. 3.5 – Flot de l'outil VPR [16].

VPR prend en entrée une description de l'architecture dans un ADL structurel pour paramétrer un modèle générique de FPGA en îlots de calcul. Les paramètres spécifiés sont principalement le nombre de blocs logiques (similaire au CLB Xilinx), le nombre de LUT par bloc logique, la taille des LUT, le nombre de blocs d'entrées-sorties, le type d'aiguilleur et la largeur des canaux de routage. A partir de cette modélisation un graphe de ressources est généré sur lequel sera placée-routée une *netlist*. Celle-ci est produite par un synthétiseur logique (e.g. par ABC [132]) à partir d'une spécification *Berkeley Logic Interchange Format* (BLIF) [1]. Les ressources de la *netlist* sont ensuite groupées dans les blocs logiques du FPGA par T-VPACK. VPR peut également prendre en entrée un placement préexistant. Les résultats du flot sont un fichier de placement-routage et des statistiques sur le taux d'occupation des ressources, la largeur des canaux de routage, le chemin critique, etc.

Dans [78], I. Kuon présente un couplage de VPR à l'outil *Good Instant Layout of Erasable Semiconductors* (GILES) pour la génération matérielle des FPGA modélisés. L'objectif est d'automatiser complètement la réalisation physique d'un FPGA en îlots de calcul. GILES prend en entrée une description dans l'ADL VPR et génère le dessin du masque du circuit. Il peut ensuite être envoyé en fonderie pour une fabrication.

La programmation du FPGA est réalisée via le flot T-VPACK VPR étendu pour la génération des binaires de configuration. Le circuit est configuré par un contrôleur de configuration paramétré pour le FPGA modélisé [77].

Le couplage de VPR et GILES permet la génération des outils applicatifs et de l'architecture matérielle d'un FPGA en îlots de calcul à partir d'une modélisation ADL. VPR guide la conception par des phases amont d'exploration. De plus, il met à disposition automatiquement les outils de placement-routage pour l'architecture modélisée. Ce flot rassemble tous les avantages du flot ADL décrit en section 3.2.1. Cependant, dans le contexte actuel d'augmentation des prix des masques de circuit, l'orientation vers une réalisation matérielle est discutable. Ainsi, la génération de l'architecture dans les flots ADL est principalement utilisée à des fins d'exploration en simulation ou de prototypage sur un FPGA du commerce.

L'approche VPR/GILES reste très restreinte car elle ne cible qu'une seule famille d'architectures. Des travaux se sont intéressés à l'extension de VPR pour la prise en compte de blocs à gros-grain [12]. Mais à notre connaissance aucun outil de génération matérielle des architectures n'est disponible pour cette évolution.

3.3.1.2 ADRES

Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) est une architecture paramétrique composée d'un processeur VLIW fortement couplé à une matrice reconfigurable à gros-grain [104, 24]. Le flot d'exploration est illustré par la figure 3.6. Il se décompose en trois parties qui correspondent à un flot de compilation reciblable, la simulation et la synthèse d'une instance paramétrée de ADRES. Les outils sont spécialisés en fonction d'une description *eXtensible Markup Language* (XML) de l'architecture qui joue le rôle d'ADL. Elle spécifie principalement le nombre d'*Unité Fonctionnelle* (UF) de la partie *Very Long Instruction Word* (VLIW) et reconfigurable, la taille des bancs de registres locaux aux UF, les opérations supportées par les UF et la topologie de la matrice reconfigurable.

L'architecture est programmée par une description C de l'application prise en entrée du frontal IMPACT puis du compilateur *Dynamically Reconfigurable Embedded System Compiler* (DRESC) [105]. Les fichiers produits en sortie sont utilisés pour générer un simulateur de jeu d'instruction. Il permet d'évaluer les performances de l'architecture. A cela s'ajoute un fichier binaire pris en entrée de deux simulations matérielles. La première est réalisée dans ModelSim à partir de la conversion en VHDL synthétisable de la description XML. Elle permet une simulation RTL fine et évalue la consommation de l'architecture. De plus, les fichiers VHDL peuvent être synthétisés pour estimer la réalisation physique du circuit. La deuxième simulation correspond à la génération Esterel de l'instance avec une modélisation en machine à état fini. Elle a pour objectif une vérification comportementale plus rapide qu'à un niveau RTL.

Le flot d'exploration ADRES propose une approche qui couvre la génération des outils de compilation, de simulation et la réalisation matérielle d'une architecture. Malgré une modélisation à haut-niveau, l'espace d'exploration est limité à un processeur VLIW couplé à une matrice à gros-grain de type ADRES.

3.3.1.3 DesignTrotter

DesignTrotter est un ensemble d'outils d'exploration de l'espace de conception des SoC [93]. Une de ses parties est dédiée aux architectures reconfigurables avec une exploration en deux étapes [21] comme illustré par la figure 3.7.

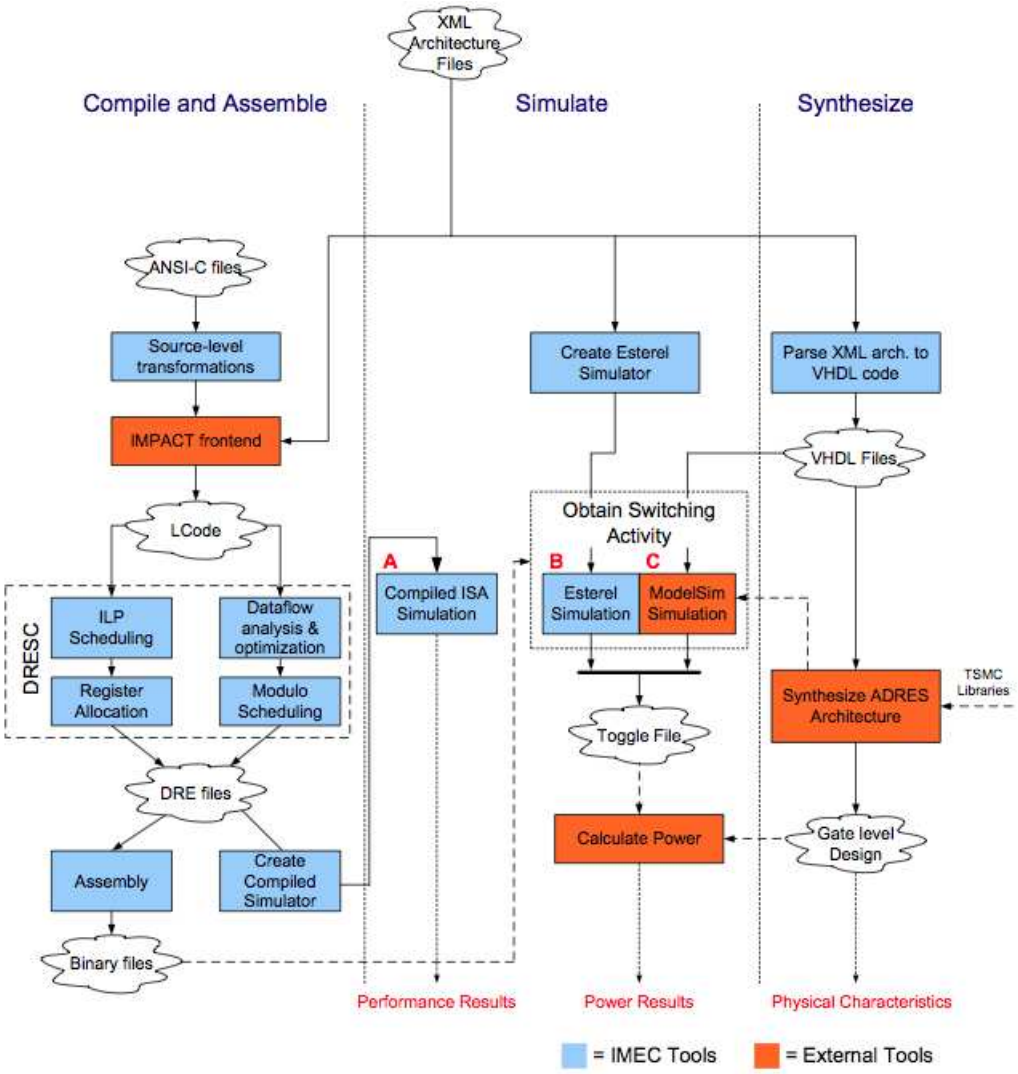


FIG. 3.6 – Flot d'exploration de l'architecture à gros-grain ADRES [24].

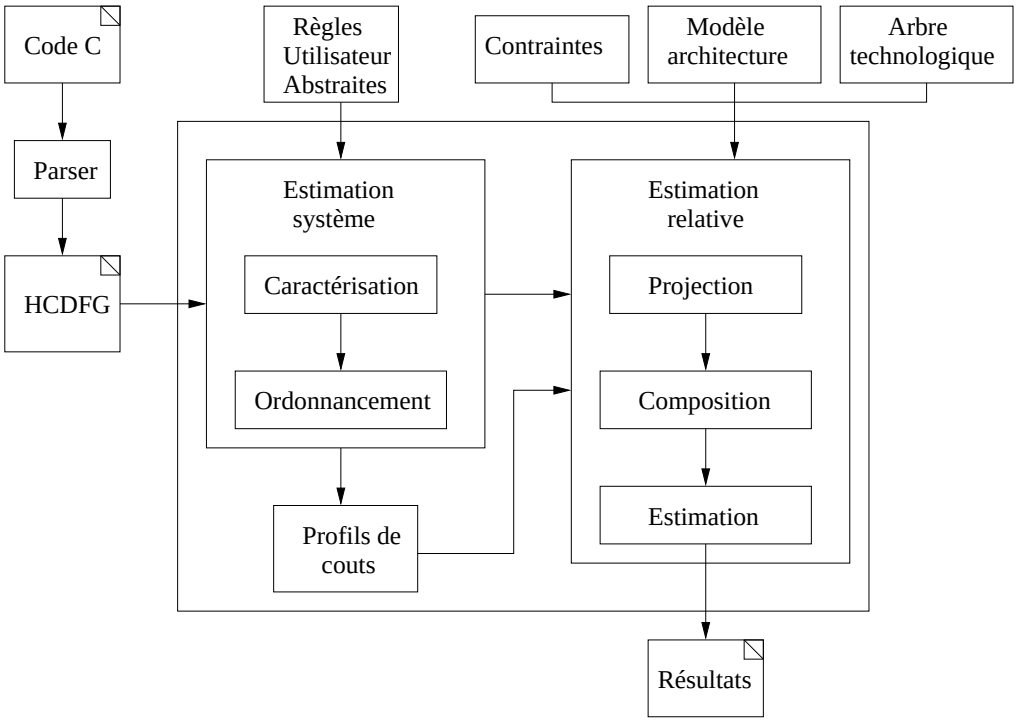


FIG. 3.7 – Flot d'estimation relative pour les architectures reconfigurables [21].

Le flot prend en entrée une application spécifiée dans le langage C qui est convertie en graphe de flot de données et de contrôle hiérarchique (*Hierarchical Control Data Flow Graph* (HCDFG)). Cette représentation intermédiaire sert de support pour les estimations systèmes. Une première étape caractérise les opérations prédominantes dans l'application (opérateurs, communications mémoire, etc.). Ces informations sont ensuite utilisées pour produire des ordonnancements sous contrainte de ressources et de temps définis dans des profils de coûts.

L'estimation relative prend en entrée les profils de coûts pour une évaluation de l'application sur un panel d'architectures reconfigurables. L'objectif est de guider le concepteur sur des choix architecturaux tels que la structuration hiérarchique de l'architecture, la taille des *clusters*, le nombre et la nature des ressources. Les estimations sont effectuées sur un modèle fonctionnel produit à partir d'une description de l'architecture. L'ADL utilisé permet la description de la fonctionnalité des ressources ainsi que leur organisation hiérarchique. L'étape de *projection* effectue le lien entre les ressources nécessaires à l'application et celles disponibles. La *composition* affine l'étape précédente par une prise en compte des profils de coûts. Enfin, l'*estimation* caractérise les performances de l'application pour l'architecture considérée.

L'outil *DesignTrotter* est une méthode d'exploration à haut-niveau pour définir une adéquation algorithme-architecture. En revanche, il est spécialisé dans le domaine de l'estimation et ne prend pas en compte le prototypage matériel de la cible ainsi que ses outil applicatifs. Ce choix est principalement motivé par une volonté d'explorer rapidement l'espace de conception sans simulation matérielle lente.

3.3.1.4 La plate-forme MOZAIC

La plate forme MOZAIC [90] est centrée sur la modélisation et la conception d'architectures reconfigurables dynamiquement. Elle n'est pas limitée à une architecture particulière et adresse des granularités variables voire hétérogènes. La figure 3.8 détaille le flot MOZAIC.

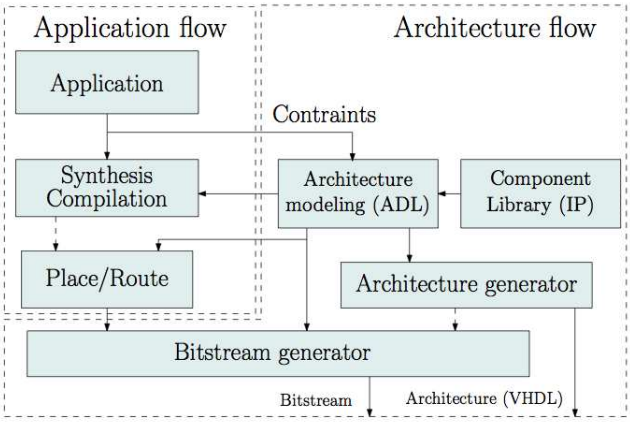


Fig. 3.8 – Flot MOZAIC de modélisation et de conception d'une architecture reconfigurable dynamiquement [92].

Le flot se décompose en deux parties : application et architecture. La partie applicative est fondée sur le couplage de MOZAIC avec des chaînes d'outils préexistants qualifiées de *plate-formes hôtes*. Par exemple, une plate-forme hôte pour FPGA prendrait en entrée une description VHDL ou Verilog synthétisée par ABC [132], qui est ensuite placée/routée dans l'outil VPR [16]. Les résultats du placement routage sont traités par une partie de MOZAIC qui génère le binaire de configuration pour l'architecture modélisée. Le ciblage d'une architecture de différente nature (e.g. gros-grain) nécessite le redéveloppement du générateur. Par conséquent, MOZAIC n'intègre pas la notion de reciblage du générateur de code natif telle que définie dans le cadre des ASIP.

Le langage *eXtended MAML* (xMAML) [92] utilisé pour la modélisation est une extension de *MAchine Markup Language* (MAML) [80]. Ce dernier est initialement dédié à la description de processeurs massivement parallèles. xMAML permet la spécification structurelle d'architectures hétérogènes reconfigurables partiellement. Pour cela la matrice est divisée en *domaines* de reconfiguration indépendants. Il prend également en compte la préemption de l'exécution de tâches. Ce mécanisme s'appuie sur une isolation des ressources de calcul par l'ajout d'un contexte de configuration [91]. L'ensemble de ces paramètres est pris en compte pour la génération VHDL de l'architecture. Les opérations de reconfiguration dynamique et d'ordonnancement des contextes sont gérées par un contrôleur générique spécialisé par la description [90].

L'exploration est réalisée sur l'impact des mécanismes de reconfiguration dynamique sur la surface, la consommation et les performances de l'architecture. Les évaluations sont également réalisées sur l'architecture matérielle générée en VHDL à partir du modèle.

3.3.1.5 Flot pour ASIP reconfigurable

Le flot d'exploration présenté dans [29] utilise une extension de l'ADL LISA pour la modélisation d'une matrice reconfigurable couplée à un processeur. L'architecture ainsi définie est qualifiée de ASIP reconfigurable (*reconfigurable Application-Specific Instruction-set Processor* (rASIP)). La figure détaille le flot de génération de l'architecture et de ses outils.

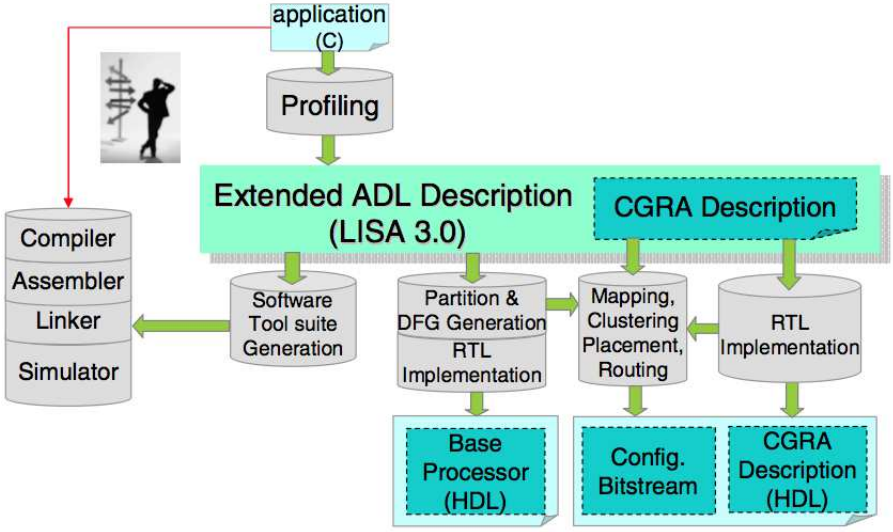


FIG. 3.9 – Flot d'exploration pour une architecture rASIP modélisée en LISA [29].

Le flot prend en entrée une application décrite dans le langage C. Similairement au flot ADL conventionnel (voir section 3.2.1), elle est profilée puis partitionnée entre le logiciel et le matériel. Les fonctions intensives sont mises en œuvre par des instructions spécifiques. Le chemin de données du processeur est ensuite partitionné pour déterminer les fonctions à placer sur la partie reconfigurable. Elles sont générées sous forme de DFG pris en entrée du flot de synthèse de la matrice. Les binaires de configuration produits sont ensuite simulés sur l'architecture matérielle générée.

La modélisation de la matrice reconfigurable est spécialisée pour le gros-grain afin de répondre aux besoins des rASIP les plus récents (voir section 2.2). Des constructions spécifiques sont ajoutées à l'ADL mixte LISA pour adresser le domaine du reconfigurable. Elles définissent les éléments de calcul, leur organisation hiérarchique et topologique ainsi que leurs interconnexions. A partir de la description ADL, les outils de compilation/synthèse, les descriptions RTL du processeur et de la matrice reconfigurable sont automatiquement générés.

Ce flot illustre l'extension d'un flot ADL pour les processeurs embarqués pour une application aux architectures reconfigurables. A partir de la description haut-niveau les prototypes matériels ainsi que leurs outils applicatifs sont automatiquement générés. Cette approche présente un grand intérêt pour les architectures reconfigurables mais dans ce cas précis elle est restreinte à une exploration du domaine des rASIP.

3.3.2 Madeo : un cadre d'outils génériques pour la modélisation et la programmation d'architectures reconfigurables

L'outil Madeo est un cadre d'outils génériques pour la modélisation et la programmation d'architectures reconfigurables. Il a été développé au sein du laboratoire LabSTICC-AS principalement par L. Lagadec [81]. La généricité des outils est assurée par une conception fondée sur le paradigme logiciel orienté objet [88].

Madeo est structuré en deux parties principales qui sont Madeo-FET et Madeo-BET. La figure 3.10 donne une vue d'ensemble du flot Madeo.

Madeo-FET est un *front-end* de programmation capable de synthétiser une description fonctionnelle et non typée d'une application [89]. Madeo-BET intervient à un niveau architectural et met à disposition un *back-end* générique (*floorplanner*, placeur-routeur). Il est indépendant des outils constructeurs et opère sur n'importe quel modèle d'architecture reconfigurable produit à partir d'une spécification dans le langage de description d'architecture Madeo-ADL (voir section 3.3.2.1). Ainsi, l'approche générique des outils Madeo autorise le placement-routage des applications synthétisées par Madeo-FET sur l'ensemble des modèles d'architectures produits par Madeo-BET [87].

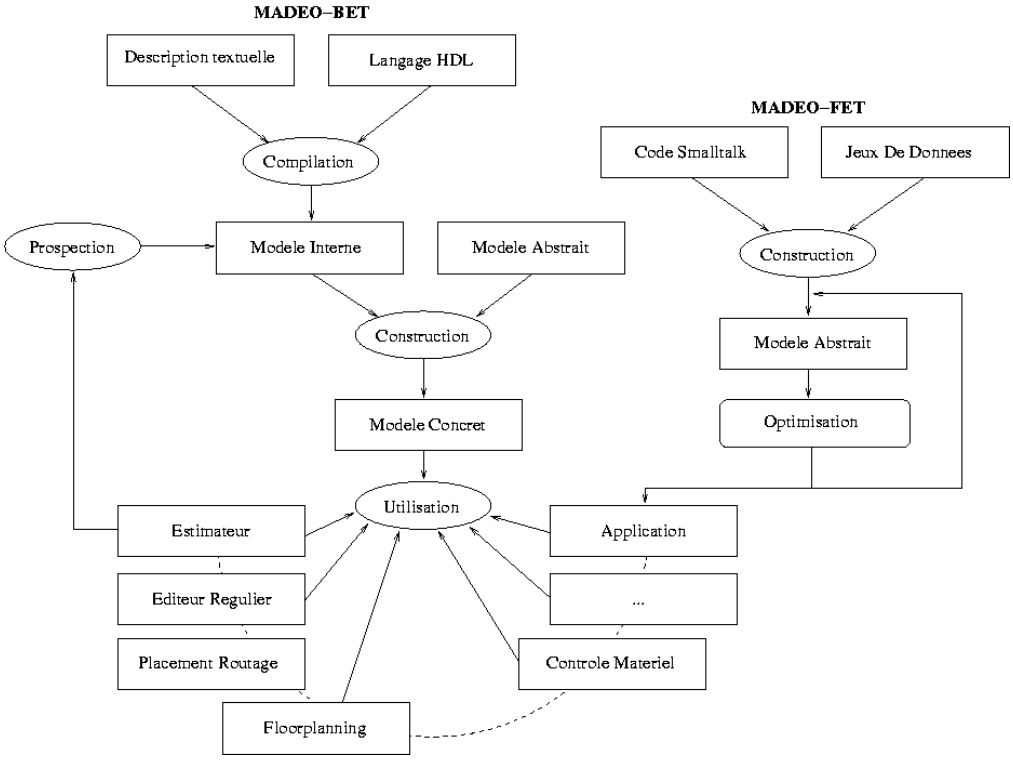


FIG. 3.10 – Flot de conception de Madeo.

Notre outil est client de la partie Madeo-BET. Il exploite d'une part ses capacités de modélisation via Madeo-ADL et d'autre part ses outils de *back-end*.

```
1 (ARRAY
2   (DOMAIN x1 y1 x2 y2)
3     {éléments}
4 )
```

Listing 3.4 – Syntaxe d'un ensemble de domaines.

3.3.2.1 Modélisation d'architectures reconfigurables avec Madeo-ADL

La description ADL est compilée pour produire un modèle interne qui correspond à l'instanciation du modèle abstrait d'architecture de Madeo. Le modèle abstrait favorise la modélisation d'une large gamme d'architectures puisque celui-ci fournit un ensemble paramétrique d'éléments constitutifs d'une architecture reconfigurable abstraite. Le modèle concret, qui correspond à une instance particulière du modèle abstrait, est utilisé comme base de connaissance pour les outils. Autrement dit, il centralise l'ensemble de la connaissance structurelle de l'architecture notamment les interconnexions entre éléments, la position des éléments, la composition des hiérarchies, etc. Ces informations sont utilisées pour le placement-routage d'applications synthétisées sous forme de logique BLIF ou *Electronic Design Interchange Format* (EDIF) par le synthétiseur logique Madeo-FET.

Cette section donne les principes de description du langage structurel Madeo-ADL et quelques éléments de syntaxe. Pour plus d'informations à propos de l'utilisation de Madeo le lecteur pourra se référer au tutorial [82].

Les structures hiérarchiques

Matrice à deux dimensions. Madeo-ADL est orienté vers la description d'architectures reconfigurables et prend en compte leur nature régulière. Une architecture est représentée sous la forme d'un tableau répliquant un motif appelé *tuile*. Le code 3.4 montre la syntaxe pour déclarer un tableau (*ARRAY*) de groupes d'éléments répliqués nommés domaines. Un domaine est défini par le mot-clé *DOMAIN*.

Les paramètres d'un domaine spécifient le coin inférieur gauche (x1, y1) et le coin supérieur droit (x2, y2) d'une matrice de tuiles identiques. La figure 3.11 illustre le principe de réplication des tuiles dans un domaine. Dans cet exemple une tuile est formée d'une LUT et de quatre bus de routages connectés à un aiguilleur. La réplication régulière de ce motif de base forme une architecture de type FPGA (blocs d'entrées-sorties exclus).

Le composite. Une tuile est un élément hiérarchique qui structure le motif de base de l'architecture reconfigurable. La hiérarchie est définie par un élément composite qui contient une liste de sous-éléments (atomiques ou hiérarchiques) et leurs interconnexions (voir code 3.4). Contrairement aux éléments atomiques (e.g. registres, multiplexeurs, etc.) un élément composite ne définit pas explicitement ses entrées et sorties. Ses connexions internes et externes sont spécifiées dans sa liste de connexions.

Les éléments atomiques

La fonction. La fonction est un élément de calcul qui peut être de différentes natures en fonction de sa description. Elle est définie par un ensemble d'entrées-sorties et optionnellement par un ensemble d'opérations possibles. Si ces opérations ne sont pas définies, par défaut la

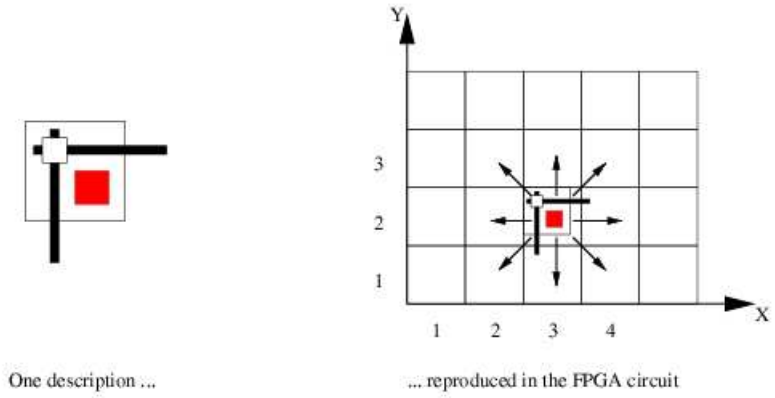


FIG. 3.11 – Réplication de la description.

```
(COMPOSITE
  ({liste des éléments} )
 {liste des connexions}
)
```

Listing 3.5 – Syntaxe de déclaration d'un élément hiérarchique composite.

fonction est considérée comme une LUT et sera configurée par de la logique (e.g. BLIF). Dans le cas contraire la fonction est considérée comme un opérateur de calcul prédéfini sur lequel sera lié un élément de librairie (e.g. EDIF).

Les fils. Les fils sont déclarés par le mot-clé *WIRE* (voir code 3.7). Leur largeur peut être unitaire (fils simples), ou n-aires et dans ce cas le fil est considéré comme un bus. Le mot-clé *EXPANDED* permet au routeur de discrétiser chaque fil d'un bus et d'effectuer une allocation par fil.

L'aiguilleur. L'aiguilleur définit les interconnexions possibles entre plusieurs bus. Il est spécifié par une énumération des fils utilisés et sa topologie (voir code 3.8).

Un exemple d'aiguilleur à topologie disjointe [95] est donné par le code 3.9.

Accès aux éléments

Nommage des éléments. Le nommage des éléments est requis lorsque ceux-ci sont agrégés au sein d'un composite. Les noms sont principalement utilisés pour accéder aux entrées-sorties des éléments.

```
(FUNCTION
  (INPUTS liste des entrées)
  (OUTPUTS liste des sorties)
  {(AMONG opérations)}
)
```

Listing 3.6 – Syntaxe de déclaration d'une fonction.

```
1 Définition d'un fil : (WIRE ( WIDTH largeur du fil ))
2
3 Définition d'un bus : (WIRE ( WIDTH largeur du fil) EXPANDED)
```

Listing 3.7 – Syntaxe de déclaration des fils.

```
1 (SWITCHBLOCK
2   (RESSOURCES
3     ( nom du fil1
4       nom du fil2
5       ... ))
6     liste des connexions)
```

Listing 3.8 – Syntaxe de l'aiguilleur.

```
1 ((SWITCHBLOCK
2   (RESOURCES
3    (south
4    east
5    north
6    west))
7   'self north connectTo: self east'
8   'self north connectTo: self south'
9   'self north connectTo: self west'
10  'self west connectTo: self east'
11  'self west connectTo: self south'
12  'self east connectTo: self south')
13 NAMED switch)
```

Listing 3.9 – Exemple d'aiguilleur à topologie disjointe.

```
1 (élément NAMED nom)
```

Listing 3.10 – Syntaxe de NAMED.

```
1 ((LINK '(self relativeAt: {position de cellule cible}) {nom fil}')
2 NAMED nom du second fil)
```

Listing 3.11 – Syntaxe d'un lien entre deux éléments.

```
1 (REFERENCE nom de l'élément)
```

Listing 3.12 – Syntaxe d'une référence à un élément.

Les alias. Les composites peuvent partager des ressources, par exemple un fil qui les connecte. Pour éviter la redéfinition de ces éléments des alias sont utilisés (voir code 3.11).

Les références. Les références permettent l'utilisation d'éléments déjà définis dans d'autres descriptions et favorisent une factorisation de la description (voir code 3.12).

Interconnexion des éléments

Les éléments d'une cellule sont connectés entre eux dans la partie connexion d'un composite. Il est également possible de connecter les éléments de plusieurs tuiles différentes par l'utilisation du mot-clé *relativeAt*. Dans l'exemple du code 3.13 la sortie de l'élément B est connectée à l'entrée d'un élément A qui se situe à la position $(x, y+3)$ avec (x, y) la position de la tuile courante.

3.3.2.2 Le modèle abstrait d'architecture reconfigurable de Madeo-BET

Le modèle abstrait de Madeo-BET est une hiérarchie de classes qui définit les éléments constitutifs d'une architecture reconfigurable. Ces éléments sont spécialisés en fonction de la spécification Madeo-ADL de l'architecture. Pour une description détaillée sur la structuration du modèle de classes de Madeo le lecteur pourra se référer à [81].

3.4 Conclusion

Les sections précédentes ont détaillé des flots d'exploration d'architectures reconfigurables qui diffèrent suivant un certain nombre de points. Ces différences sont complémentaires et ne permettent pas d'établir un classement qualitatif entre les solutions.

La figure 3.12 donne les axes principaux qui caractérisent les méthodes d'exploration.

Le tableau 3.1 effectue une synthèse des flots présentés dans la section précédente en fonction des axes de la figure 3.12. Au niveau de la cible architecturale nous ne prenons

```
1 (CONNECTION
2   élémentA connect: sortie1 de objetA to: un fil
3   élémentB connect: #out to: ((self relativeAt: 0@3) élémentA at: #in)
4   ...)
```

Listing 3.13 – Syntaxe de CONNECTION.

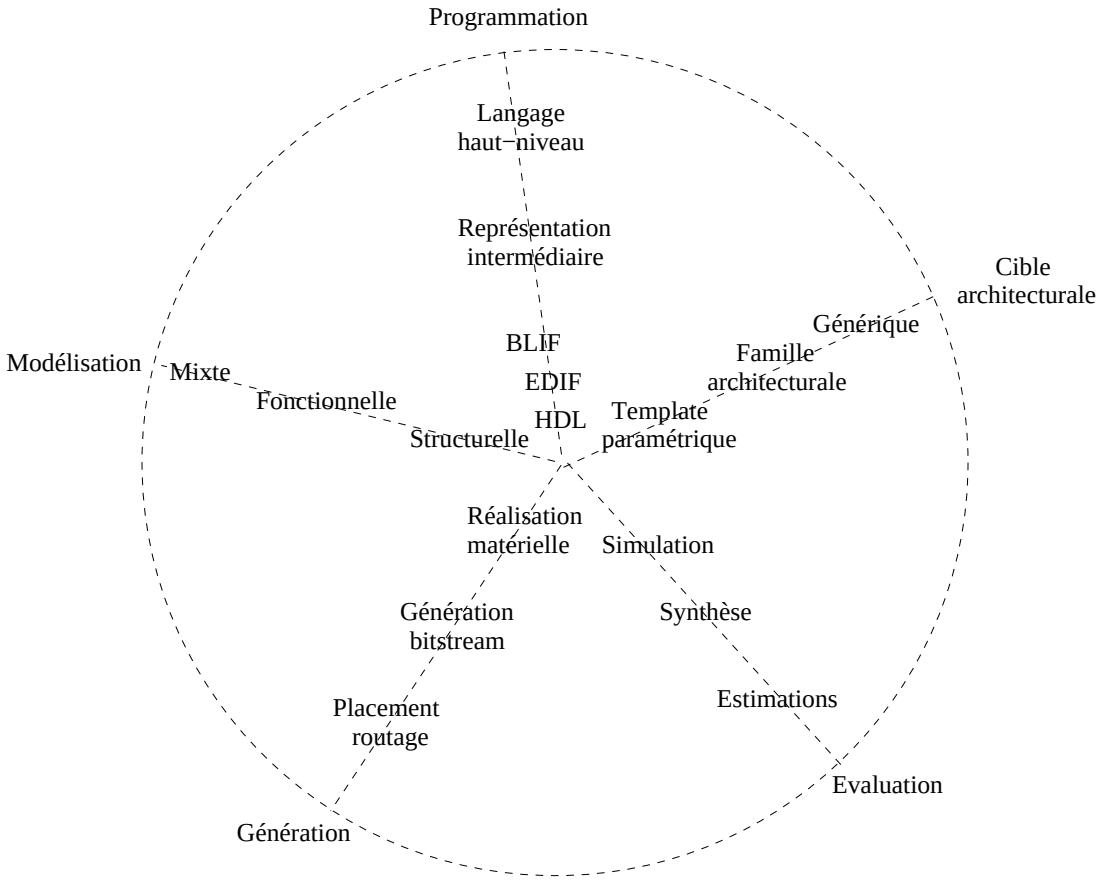


FIG. 3.12 – Caractéristiques des méthodes d'exploration de l'espace de conception des architectures reconfigurables.

Outils	Cible architecturale	Génération	Programmation	Modélisation	Evaluation
VPR/GILES [16, 78]	Famille FPGA	Outils P&R Réalisation physique Bitstream	BLIF	Structurelle	Synthèse Simulation
ADRES [24]	CGRA paramétriques	Compilateur Outils P&R Simulateur Matérielle Bitstream	ANSI-C	Mixte	Synthèse Simulation
DesignTrotter [21]	Générique	-	ANSI-C	Fonctionnelle	Estimations
MOZAIC [90]	Générique	Matérielle	Flots externes	Mixte	Simulation
rASIP [29]	Famille CGRA	Compilateur Outils P&R Simulateur Matérielle Bitstream	ANSI-C	Mixte	Synthèse Simulation
Madeo [81]	Générique	Outils P&R Synthèse logique	BLIF EDIF Smalltalk	Structurelle	Synthèse

TAB. 3.1 – Synthèse comparative des différents flots d'exploration présentés précédemment.

en compte que la matrice reconfigurable et ne considérons pas le processeur éventuellement couplé.

Les outils d'exploration les plus proches du flot ADL défini en section 3.2.3 associent les évaluations synthèse/simulation. Ces deux axes correspondent d'une part à la génération des outils applicatifs et d'autre part à celle d'un prototype matériel. Cependant, les outils existants sont liés à des flots spécifiques, à une famille architecturale ou à une architecture paramétrique. Par conséquent, leur capacité d'exploration architecturale est restreinte. Une réponse à ce problème est la modélisation générique apportée par les outils Madeo [81] et MOZAIC [90]. Néanmoins, ils ne répondent pas complètement aux critères du flot ADL défini en section 3.2.3. Par exemple, Madeo ne génère pas de prototype matériel et MOZAIC ne prend pas en compte la mise à disposition automatique des outils applicatifs.

Pour définir un flot ADL dédié aux architectures reconfigurables, tel que défini dans la section 3.2.3, nous proposons une extension de l'outil Madeo. Le flot ainsi composé tire profit de la modélisation générique et de la capacité de spécialisation des outils de placement-routage de Madeo. De plus, il vient compléter la méthode d'évaluation par l'ajout d'une capacité de simulation matérielle de la cible. Le chapitre 4 détaille les caractéristiques de cette extension.

Chapitre 4

Exploration et prototypage d'architectures reconfigurables

4.1 Introduction

Pour une exploration rapide et efficace de l'espace de conception des unités reconfigurables, il est nécessaire de faciliter l'évaluation itérative de variantes architecturales. Cela passe par la mise à disposition d'outils qui permettent une modélisation à haut-niveau et facilement modifiable de l'architecture. Ils doivent également prendre en charge la génération de prototypes pour en permettre l'évaluation et alléger la tâche du concepteur.

L'approche de notre flot d'exploration est fondée sur une modélisation en trois plans qui représentent les trois axes de variation de l'espace de conception de la figure 3.4. L'architecture reconfigurable est constituée de deux plans, calcul et configuration, auxquels s'ajoute le plan application. Ces trois plans sont illustrés par la figure 4.1.

Plan de calcul. Ce plan contient l'ensemble des informations structurelles et comportementales de l'architecture décrite et modélisée en ADL. Le modèle est composé hiérarchiquement d'un ensemble de ressources de calcul et de routage possiblement hétérogènes (bloc de calcul de granularité variable, bloc d'entrées-sorties). Ce plan modélise également les mots de configurations associés à chaque élément configurable. Le modèle de binaire de configuration obtenu est équivalent à la couche ISA d'un processeur et contraint la génération des binaires de configuration. En revanche, il ne conditionne pas l'organisation structurelle de la mémoire de configuration de l'architecture. Cet aspect est modélisé par le plan de configuration.

Plan de configuration. Ce plan modélise et structure la mémoire de configuration de l'architecture en zones reconfigurables. Chaque zone est reconfigurable dynamiquement par des pages de configuration et peut avoir plusieurs contextes de configuration. L'ensemble des contraintes définies dans ce plan sont indépendantes de l'organisation interne des mémoires de configuration (modèle de binaire de configuration) et peuvent varier indépendamment du plan de calcul.

Application. Ce plan correspond à l'application à mettre en œuvre sur la solution architecturale explorée. Sa description est prise en entrée d'un *front-end* de synthèse de haut-niveau connecté aux outils de *back-end* qui sont spécialisés en fonction des plans de

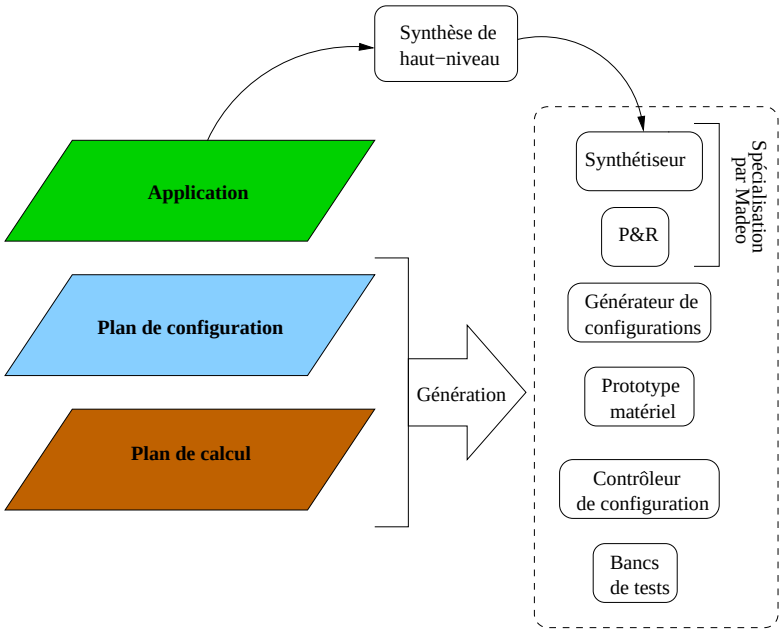


FIG. 4.1 – Structuration en trois couches d'une architecture reconfigurable.

calcul et de configuration. De ce fait, l'application peut varier indépendamment de l'architecture. Le modèle de programmation d'application ainsi que le *front-end* de synthèse de haut-niveau sont détaillés dans le chapitre 5.

Les variations de chacun des plans ont un impact sur les aspects architecturaux et les outils applicatifs. La cohérence des plans est assurée par la prise en compte automatique de ces changements lors des étapes de génération de l'architecture et des outils.

DRAGE : extension du flot Madeo-BET.

Ce chapitre présente l'outil DRAGE qui est une extension du flot Madeo-BET pour la génération de prototypes d'architectures reconfigurables et de leurs outils applicatifs. Cet outil offre aux concepteurs une capacité de validation précoce de concepts architecturaux et la caractérisation des architectures par différentes métriques.

Les architectures sont générées en VHDL comportemental simulable dans un simulateur cycle prêt (par exemple ModelSim) à des fins de validation. Elles peuvent également être synthétisées sur un FPGA du commerce pour une exécution dans le matériel (e.g. une carte Virtex-5 [148], voir chapitre 7.4).

Madeo-BET correspond à la partie de Madeo dédiée à la modélisation des architectures reconfigurables. Cet outil produit un *Modèle du Plan de Calcul* (MPCalc) exploitable par les outils de *back-end* mais qui n'est pas simulable à un niveau matériel. En effet, le modèle Madeo-BET est un modèle structurel de l'architecture ayant pour vocation de caractériser sous forme d'un graphe de ressources l'ensemble de ses éléments. Ce graphe a pour sommet les ressources de calcul et pour arcs les ressources de routage permettant ainsi le placement-routage d'applications. Il ne contient donc aucune information comportementale sur ses ressources.

La figure 4.2 illustre l'extension DRAGE du flot Madeo-BET pour la génération d'un environnement complet matériel (prototype d'architecture) et logiciel (générateur de binaire

de configuration). Cet environnement est composé des trois plans de modélisation spécialisés pour la simulation d'une solution architecturale (voir section 3.3). Le flot se décompose en deux parties avec d'une part la génération d'un MPCalc et d'autre part la génération d'un *Modèle du Plan de Configuration* (MPConf).

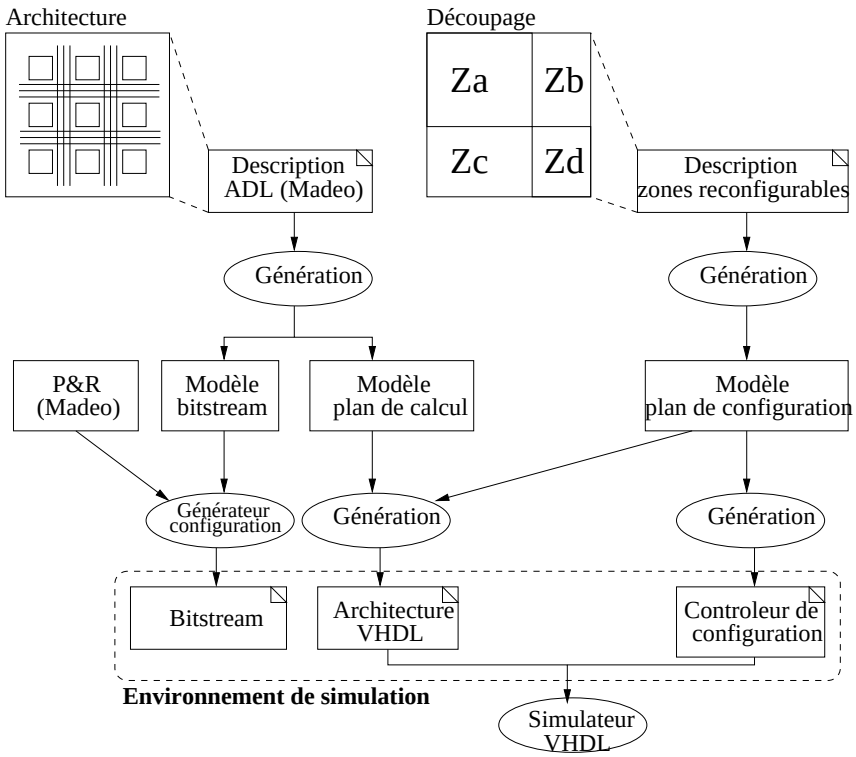


FIG. 4.2 – Détail du flot DRAGE pour la génération d'un environnement complet de simulation pour le prototypage d'une architecture reconfigurable.

Plan de calcul. Le MPCalc est généré à partir d'un modèle d'architecture produit par Madeo-BET. Il centralise les informations structurelles et comportementales du plan de calcul alors que le modèle Madeo-BET définit uniquement les aspects structurelles (topologie, connectiques, etc.). Le MPCalc est obtenu par l'instanciation des classes d'une hiérarchie qui représentent de manière abstraite les éléments du plan de calcul. Le résultat de la génération est un MPCalc conforme à l'architecture modélisée à haut-niveau dans Madeo-BET et capable de produire une description VHDL. Additionnellement, un modèle de binaire de configuration (ou de *bitstream*) spécifique au MPCalc est produit. Ce modèle structure les mots de configuration du plan et établit une correspondance entre les éléments du modèle Madeo-BET et ceux du MPCalc. Cette information est utilisée pour la génération des binaires de configuration à partir du placement-routage d'une application.

Plan de configuration. La génération du MPConf prend en entrée une description du découpage global du plan en zones reconfigurables. A partir de ces informations une zone reconfigurable sera instanciée par une caractérisation de sa surface (exprimée en nombre de

tuiles) et son nombre de contextes de configuration. Le MPConf contraint la génération de code VHDL de l'architecture et celle du contrôleur de configuration.

L'environnement complet est généré à partir des deux modèles : MPCalc et MPConf. Lors de la simulation, l'architecture générée est configurée par un contrôleur de configuration spécialisé en fonction du MPConf. Ce contrôleur prend en entrée les binaires de configuration produits en fonction du modèle de *bitstream*.

4.2 Modélisation et génération du plan de calcul

Le modèle Madeo-BET contient les informations structurelles de l'architecture reconfigurable modélisée. Cette section détaille la hiérarchie de classes instanciée pour produire un MPCalc qui centralise la structure et le comportement de l'architecture.

4.2.1 Hiérarchie de classes du modèle du plan de calcul

La hiérarchie de classes du MPCalc peut être étendue pour intégrer de nouveaux éléments. Cette organisation est conforme à la notion de cadriciel orienté objet exploitée par Madeo-BET (voir section 3.3.2). Chaque classe de la hiérarchie du MPCalc est lié à une classe de celle de Madeo-BET. Par conséquent, l'enrichissement de l'une sera toujours réalisé sous condition d'un ajout du même élément dans l'autre.

La figure 4.3 illustre la hiérarchie de classes du MPCalc.

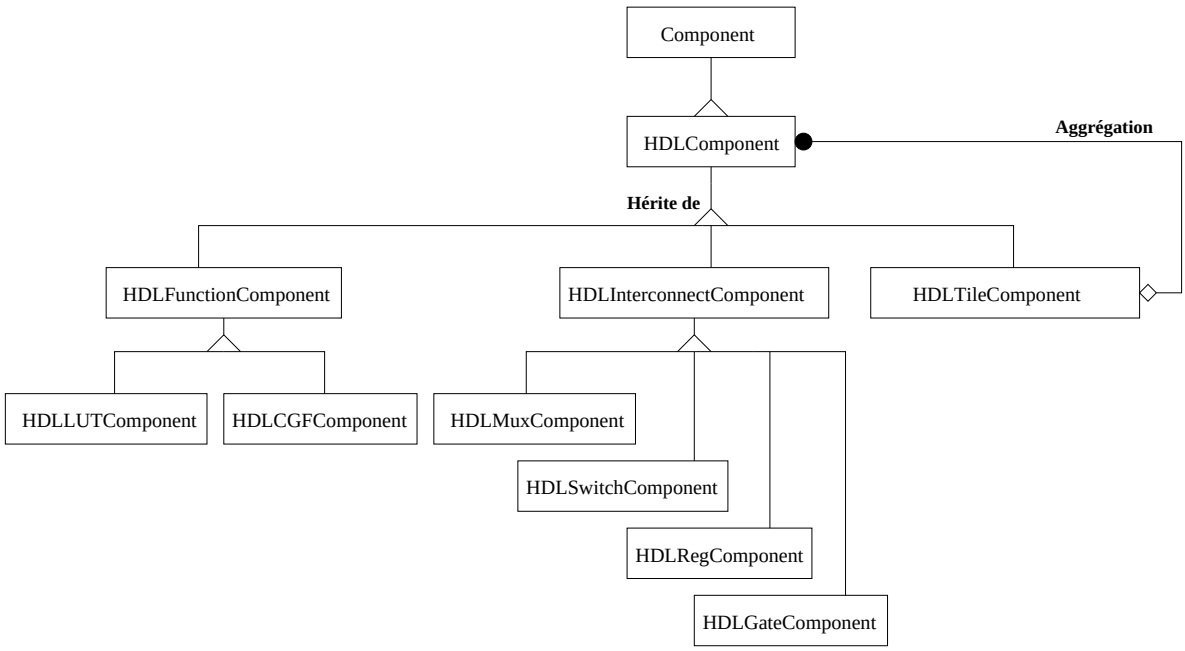


Fig. 4.3 – Hiérarchie de classes du modèle du plan de calcul.

Les éléments du MPCalc sont nommés composants et héritent tous de la classe *Component* qui représente un composant abstrait. La classe *HDLComponent* est le premier niveau de spécialisation pour la génération de code VHDL. La prise en charge d'un autre langage *Hardware*

Madeo-ADL	Classes de Madeo-BET	Classes du MPCalc
FUNCTION	RaFunction	HDLLUTComponent/HDLCGFComponent
MULTIPLEXER	RaMultiplexer	HDLMuxComponent
REGISTER	RaRegister	HDLRegComponent
CONNECTION	RaPassTransistor	HDLGateComponent
COMPOSITE	RaComposite	HDLTileComponent

FIG. 4.4 – Correspondance entre les mots-clés de Madeo-ADL, les classes du modèle Madeo-BET et celles du MPCalc.

Description Language (HDL) équivalent implique la redéfinition d'une hiérarchie de classes qui hérite de *Component*.

Un composant est défini principalement par les attributs suivants :

- **Nom** : le nom correspondant à celui de l'entité qui sera créée dans le cas d'une génération de code VHDL.
- **Entrées et sorties** : les entrées-sorties sont définies par un nom, une largeur exprimée en bits et une direction. La collection d'entrée inclue également le bus de configuration dont la largeur est déterminée par la taille du mot de configuration du composant.
- **Comportement** : le comportement d'un élément est variable suivant sa nature atomique ou hiérarchique. La génération des comportements est détaillée en section 4.2.2.
- **Position** : la position est celle de l'élément dans la matrice du modèle de Madeo-BET. Elle est exprimée par une abscisse et une ordonnée et permet de faire le lien entre les deux modèles.
- **Domaine de configuration** : cet attribut précise à quel domaine de configuration l'élément appartient. Cette notion est développée dans la section 4.3.1.
- **Mot de configuration** : cet attribut correspond au contenu de la mémoire de configuration du composant. Il est assigné lors de l'étape préliminaire à la génération d'un binaire de configuration. Cette notion est développée dans la section 4.4.
- **Sous-entités** : dans le cas d'un composant hiérarchique, cet attribut est la collection de ses sous-éléments.
- **Modèle de configuration** : dans le cas d'un composant hiérarchique, le modèle de configuration établit une correspondance entre le mot de configuration de l'élément et celui de ses sous-éléments.
- **Règles de routage** : dans les composants hiérarchiques, ces règles déterminent l'interconnexion des ports du composant avec les autres composants du modèle.

HDLComponent est spécialisé en trois types de composants. Le premier type, *HDLFunctionComponent*, représente les composants réalisant des fonctions de calcul tels que les LUT (*HDLLUTComponent*) ou les opérateurs à gros-grain de type UAL (*HDLCGFComponent*). Le deuxième type sont les éléments d'interconnexions qui représentent les ressources de routage du modèle. On peut citer les multiplexeurs (*HDLMuxComponent*), les aiguilleurs (*HDLSwitchComponent*), les registres (*HDLRegComponent*) et les interrupteurs (*HDLGateComponent*). Le dernier type est un composant structurel qui maintient l'organisation hiérarchique du modèle. Il peut contenir n'importe quel composant : hiérarchique ou atomique. Il forme ainsi les tuiles de l'architecture reconfigurable (*HDLTileComponent*).

Le tableau 4.4 résume la correspondance entre les mots-clés de Madeo-ADL, les classes du modèle Madeo-BET et celles du MPCalc de DRAGE.

4.2.2 Génération du modèle du plan de calcul

Le code VHDL comportemental de l'architecture modélisée est généré à partir du MPCalc et du MPConf. Le MPCalc est généré par un parcours hiérarchique de l'ensemble des éléments du modèle produit par Madeo-BET. Cette section décrit le principe de génération des deux types d'éléments d'une architecture : hiérarchique et atomique.

4.2.2.1 Génération des éléments hiérarchiques du MPCalc

Le modèle d'architecture produit par Madeo-BET est un graphe dirigé hiérarchique dont les sommets sont des composants et les arcs des connexions entre les ports d'entrées-sorties. Le plus haut niveau hiérarchique correspond à l'aboutement de tuiles. Chaque tuile contient un ensemble de sous-éléments, hiérarchiques ou atomiques, qui forme des sous-graphes. Les feuilles du graphe sont les éléments atomiques. La figure 4.5 illustre une architecture à trois niveaux hiérarchiques. Dans cet exemple, une tuile est composée d'éléments atomiques que sont les multiplexeurs et la fonction ainsi que d'un élément hiérarchique qui est l'aiguilleur (ses sous-éléments sont un ensemble d'interrupteurs).

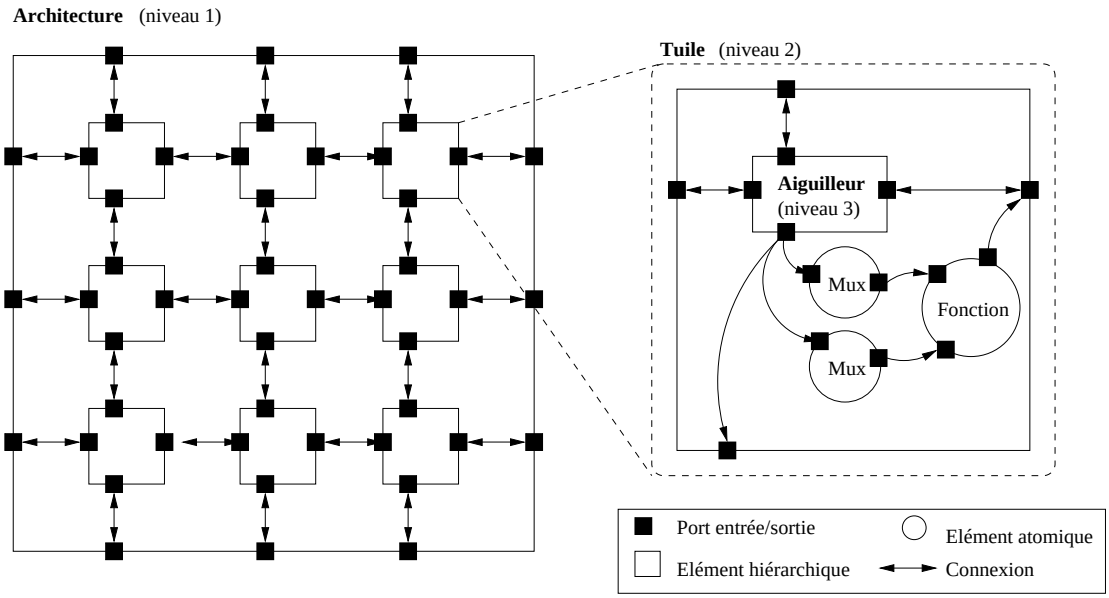


FIG. 4.5 – Graphe dirigé hiérarchique d'un modèle Madeo-BET d'architecture reconfigurable.

La génération du MPCalc est détaillée par l'algorithme 1. La fonction *générerComposant* prend en entrée un élément du modèle Madeo-BET et instancie un composant équivalent dans la hiérarchie de classes du MPCalc. Cette fonction est appliquée au premier niveau hiérarchique du modèle Madeo-BET pour produire le MPCalc.

La génération est effectuée par un parcours en profondeur des sous-éléments de chaque niveau hiérarchique du modèle. Si l'élément est atomique alors son comportement est généré et cet élément est ajouté aux sous-éléments de la hiérarchie courante. Puis pour chaque niveau hiérarchique les interconnexions entre ses sous-éléments sont déterminées par un parcours complet du sous-graphe. Cette étape détermine les connexions entre les sous-éléments locaux et déclare les ports d'entrées-sorties du composant hiérarchique.

Un élément composite d'un modèle Madeo-BET ne définit pas de ports d'entrées-sorties (voir section 3.3.2.1). Un port est déclaré lorsque la connexion a pour destination l'extérieur de la hiérarchie. Puis une règle de connexion est générée et transmise au niveau hiérarchique supérieur.

La dernière étape associe au composant créé son modèle de configuration. Dans le cas d'un élément atomique ce modèle est défini directement dans la classe. Pour un élément hiérarchique il est déterminé en fonction de ses sous-éléments. La modélisation et la production des *bitstream* sont développées en section 4.4.

Algorithme 1 Algorithme pour la génération du MPCalc.

```
1  Fonction genérerComposant(élémentCourant : élément du modèle
   Madeo-BET ) : HDLComponent
2  Variable
3      compHiérarchique : HDLTileComponent
4      compAtomique : HDLTileComponent
5      compRésultat : HDLComponent
6  Début
7      Si typeDe(élémentCourant) = Hiérarchique Alors
8          compHiérarchique ← créerComposantHiérarchique(élémentCourant)
9          PourChaque sous-élément De élémentCourant Faire
10             compGénéré ← générerComposantDepuis(sous-élément)
11             ajouter compGénéré aux sous-éléments de compHiérarchique
12         générerInterconnexionsPour(sous-éléments de compHiérarchique)
13         compRésultat ← compHiérarchique
14     Sinon
15         Si typeDe(élémentCourant) = Atomique Alors
16             compAtomique ← créerComposantAtomique(élémentCourant)
17             générerComportementPour(compAtomique)
18             compRésultat ← compAtomique
19     produireModèleDeConfigurationPour(compRésultat)
20     Retourner compRésultat
21 Fin
```

4.2.2.2 Génération du comportement des éléments atomiques

Les éléments atomiques constitutifs d'une architecture reconfigurable sont des primitives du langage Madeo-ADL. Ils sont dotés d'une sémantique comportementale implicite non définie dans le modèle car non nécessaire pour les outils applicatifs. Cependant, ces informations sont indispensables au MPCalc pour produire une architecture simulable.

La sémantique de chaque élément est définie dans les classes du MPCalc par des comportements paramétriques. Ces comportements sont spécialisés lors de l'instanciation en fonction

des paramètres issus du modèle Madeo-BET. La figure 4.6 montre le lien entre les éléments atomiques d'une tuile issu d'un modèle Madeo-BET et les comportements associés à chacun.

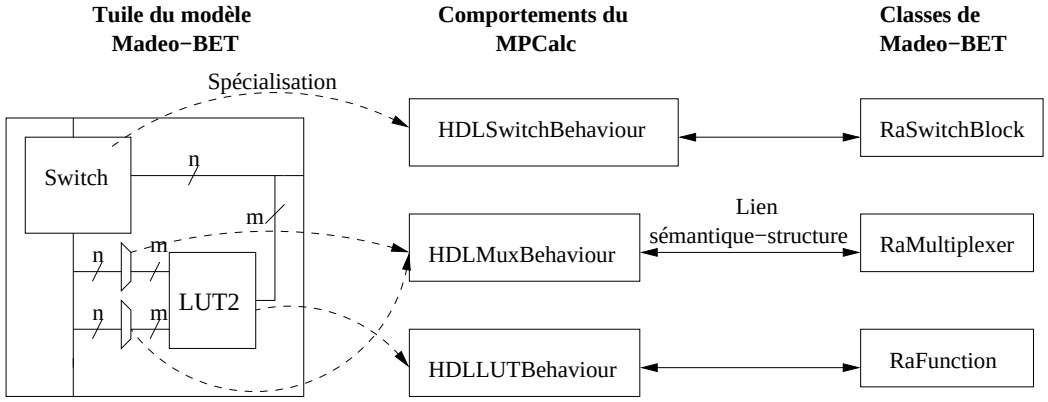


FIG. 4.6 – Association des éléments atomiques d'un modèle Madeo-BET avec leurs comportements.

Chaque élément du modèle Madeo-BET est produit par l'instanciation d'une classe. Pour déterminer la sémantique d'un élément, un lien est créé entre les classes de Madeo-BET, qui détermine la structure d'un élément, et le celles du MPCalc. Une fois qu'un comportement est associé à un élément, il est spécialisé et le code VHDL correspondant est généré. Ce code est ensuite associé à son composant dans le MPCalc.

Exemple de comportement paramétrique. Le code 4.1 montre la représentation du comportement paramétrique d'un multiplexeur du MPCalc. Conformément aux principes du langage VHDL, chaque composant est formé d'une déclaration d'entité et d'une architecture associée. A cela s'ajoute la déclaration du composant (similaire à celle de l'entité) qui n'est pas représentée ici. Les paramètres et portions de code entre accolades sont spécialisés et/ou répliqués en fonction des caractéristiques structurelles extraites d'un modèle Madeo-BET.

Le nombre d'entrées correspondra à la réplication de la ligne $\{in\_\{A\}$ *: IN STD_LOGIC_VECTOR(*$\{B\}$ *DOWNTO 0)*$\}$ *;* avec A l'identifiant unique de l'entrée et B sa largeur exprimée par un vecteur de bits. Dans un multiplexeur le nombre d'entrées conditionne le nombre de sélections possibles pour la sortie. La ligne $\{$*WHEN "*$\{E\}$*" => o <= in_*$\{A\}$ *;*$\}$ sera répliquée avec la mise à jour des noms des ports d'entrée et de la condition de sélection E.

Le code 4.2 représente le résultat de la génération du code VHDL d'un multiplexeur de type 2 entrées vers 1 sortie avec une largeur de port égale à 4 bits. Ces paramètres sont utilisés par le générateur de code pour spécialiser les éléments variables du multiplexeur donnés par le code 4.1. Dans le cas du multiplexeur, le port *sel* correspond à un port de configuration qui sera intégré au modèle de *bitstream* de sa tuile.

4.2.3 Méthodologie objet pour le reciblage du générateur de code

Pour préserver la généricité de la modélisation Madeo-BET, la logique de génération du MPCalc n'est pas intégrée ni figée dans le modèle. Cette dissociation permet le reciblage de la génération de code vers d'autres langages pour la simulation (e.g. System-C).

```
1  ENTITY mux{Nto1} IS
2          PORT(
3                  {in_{A} : IN STD_LOGIC_VECTOR({B} DOWNTO 0)};
4
5                  o : OUT STD_LOGIC_VECTOR({C} downto 0);
6                  sel : IN STD_LOGIC_VECTOR({D} downto 0));
7  END mux21;
8
9  ARCHITECTURE behaviouralmux{Nto1} OF mux{Nto1} IS
10 BEGIN
11         PROCESS({in_{A}}, sel)
12                 BEGIN
13                         CASE sel IS
14                                 {WHEN "{E}" => o <= in_{A};}
15
16                                 WHEN OTHERS => o <=  (OTHERS => 'Z');
17                         END CASE;
18         END PROCESS;
19 END behaviouralmux{Nto1};
```

Listing 4.1 – Exemple de comportement paramétrique d'un multiplexeur du MPCalc.

```
1  ENTITY mux21 IS
2          PORT(
3                  in0 : IN OUT STD_LOGIC_VECTOR (3 downto 0);
4                  in1 : IN OUT STD_LOGIC_VECTOR r(3 downto 0);
5                  o : OUT OUT STD_LOGIC_VECTOR (3 downto 0);
6                  sel : IN OUT STD_LOGIC_VECTOR (0 downto 0));
7  END mux21;
8
9  ARCHITECTURE behaviouralmux21 OF mux21 IS
10 BEGIN
11         PROCESS(in0, in1, sel)
12                 BEGIN
13                         CASE sel IS
14                                 WHEN "0" => o <= in0;
15                                 WHEN "1" => o <= in1;
16                                 WHEN OTHERS => o <= (OTHERS => 'Z');
17                         END CASE;
18         END PROCESS;
19 END behaviouralmux21;
```

Listing 4.2 – Résultat de la génération de code VHDL pour un multiplexeur de type 2 entrées vers 1 sortie avec une largeur de port égale à 4 bits.

Elle est obtenue par la conception du générateur suivant le schéma de résolution orienté objet appelé « Visiteur » [6]. Le principe est de centraliser dans une classe dédiée (qui correspond au Visiteur) l'ensemble des opérations de même nature à réaliser sur un modèle (e.g. génération de code, représentation, etc.). Pour chaque type d'opération un Visiteur est créé sans modification du modèle. Ainsi un traitement particulier peut être reciblé rapidement par la mise à disposition de hiérarchies de classes de visiteurs utilisables comme cadriciel métier. Ces hiérarchies factorisent l'ensemble des traitements génériques pour un domaine particulier : par exemple, les opérations d'entrées-sorties pour la génération de code, déterminer les interconnexions entres les éléments, etc.

La figure 4.7 montre le principe du Visiteur appliqué à la génération d'un MPCalc pour la production de code VHDL comportemental. Le visiteur et le modèle interagissent suivant un protocole d'appels de méthodes génériques.

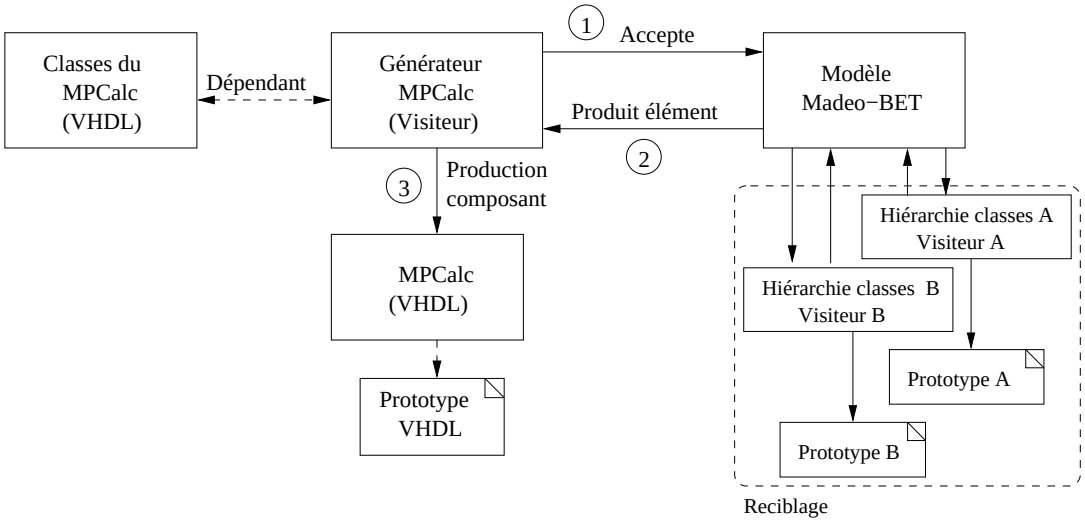


FIG. 4.7 – Génération du MPCalc par une approche fondée sur le schéma de résolution (*design pattern*) Visiteur.

La génération d'un élément du MPCalc est réalisée en trois étapes principales :

1. Le Visiteur demande à être accepté par le modèle sur lequel il opère. Cela se traduit par un appel de méthode sur le modèle avec comme passage de paramètre le visiteur lui-même.
2. L'élément appelle la méthode qui correspond à son traitement sur le Visiteur et se passe en paramètre. Cet appel est générique et l'élément ne connaît pas la nature du traitement à effectuer.
3. La dernière étape est la génération de l'élément par le Visiteur.

Dans le cas d'un élément hiérarchique le protocole d'appel est réalisé récursivement sur ses sous-éléments suivant le schéma de résolution Composite [6].

Le générateur de MPCalc est composé de deux éléments principaux qui sont la hiérarchie de classes du MPCalc et son visiteur associé. Le développement de générateurs pour d'autres langages de type HDL (e.g. Verilog, System-C) nécessite la définition de nouvelles hiérarchies

de classes (sous-classement de la classe *Component*, voir 4.3) et du Visiteur abstrait pour la génération de MPCalc.

La présence de cette infrastructure logicielle cadre et minimise l'effort de redéveloppement nécessaire pour cibler d'autres langages. Cet effort porte principalement sur les règles syntaxiques de production de code, si l'on se restreint à un paradigme composant utilisé par la plupart des langages de description matérielle.

4.3 Modélisation et génération du plan de configuration

Le MPConf organise en groupes les mémoires de configuration des tuiles du MPCalc. Les regroupements sont qualifiés de zones reconfigurables qui concatènent un nombre de mémoires de tuile inférieur ou égale au total des mémoires de configuration de l'architecture. La figure 4.8 montre le principe de regroupement en zones.

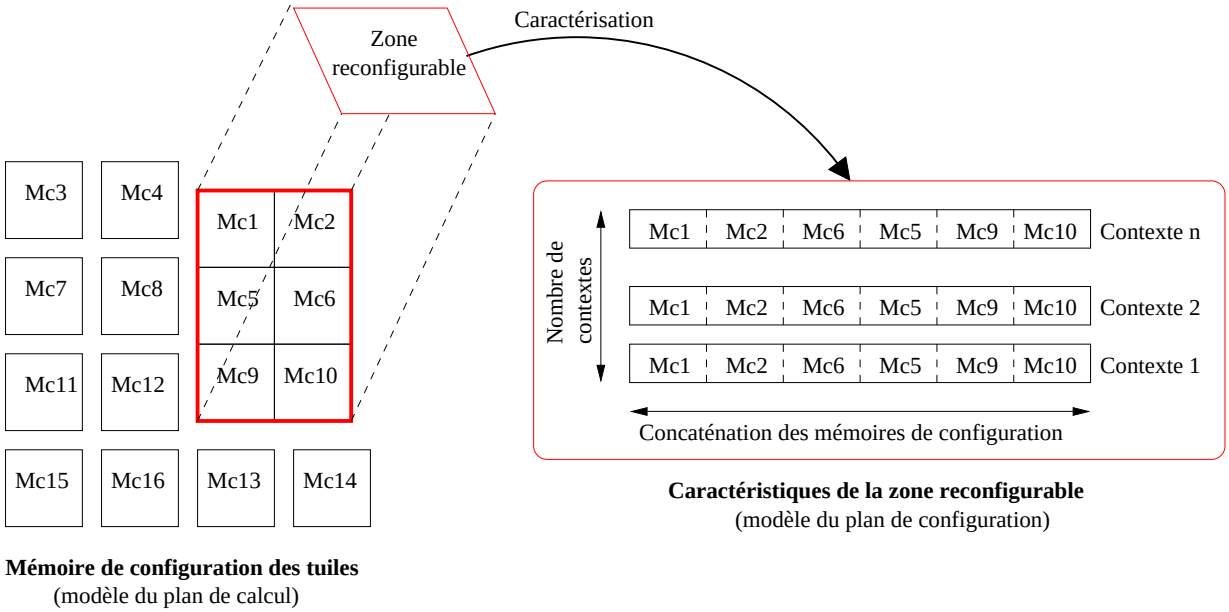


FIG. 4.8 – Structuration en zones reconfigurables des mémoires de configurations (Mc) des tuiles. Dans une même zone les mémoires sont concaténées et peuvent avoir plusieurs contextes. Une zone se comporte comme une mémoire de configuration unique.

Une zone est reconfigurable de manière indépendante permettant ainsi un mode de reconfiguration partielle du plan de configuration. Elle peut également supporter plusieurs contextes de configuration pour mettre en œuvre la reconfiguration multi-contextes [37, 138]. La taille des zones reconfigurables ainsi que leur nombre de contextes sont définis lors de la modélisation du plan de configuration. Ces deux paramètres délimitent un espace de conception du plan de configuration à deux dimensions illustré par la figure 4.9.

L'ensemble des zones reconfigurables forme le MPConf. Les contraintes définies par les zones dans le modèle sont utilisées pour d'une part la génération du code VHDL de l'architecture et d'autre part pour la génération du contrôleur de configuration microprogrammable.

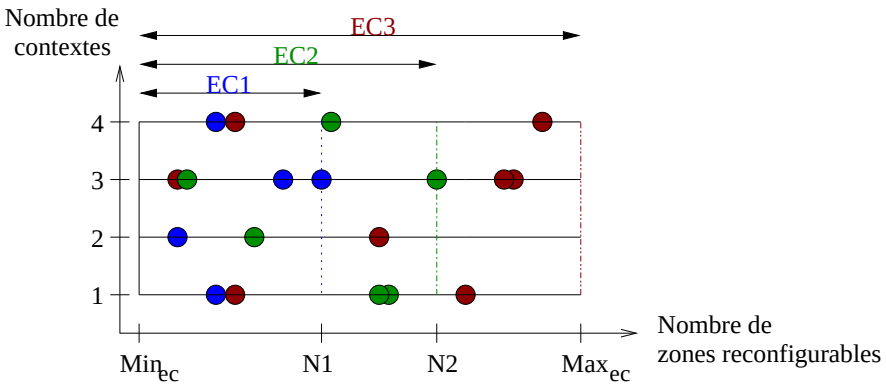


FIG. 4.9 – Exploration de l'espace de conception d'un plan de configuration en fonction du mode de reconfiguration de l'architecture. Le mode est conditionné par un nombre de zones (reconfiguration partielle) et un nombre de contextes de configuration par zone (reconfiguration multi-contextes). Ces deux paramètres déterminent les capacités de reconfiguration dynamique partielle et multi-contextes de l'architecture.

4.3.1 Modèle du plan de configuration

Définitions et propriétés. Un plan de configuration est défini par un ensemble de zones reconfigurables qui définissent la structure du plan. D'un point de vue structurel une zone reconfigurable est un rectangle dont l'unité de division est une mémoire de tuile configurable. Une zone peut inclure un ensemble de tuiles configurables possiblement hétérogènes. Au niveau comportemental, une zone est reconfigurable dynamiquement de manière indépendante et toutes ses tuiles sont configurées par un même *bitstream* maintenu dans un *contexte courant*. A ce contexte courant peuvent s'ajouter d'autres contextes pour stocker des configurations supplémentaires (voir section 4.3.2 pour les détails matériels).

Les zones reconfigurables ne peuvent pas se chevaucher et la somme des tuiles configurables qu'elles incluent est égale au nombre total de tuile de l'architecture modélisée. Cette propriété est caractérisée par la formule :

$$N_{ec} = \sum_{z=1}^{Z} nbTuiles(z) \tag{4.1}$$

N_{ec} le nombre total de tuiles de l'architecture, $Z \in [Min_{tuiles}, Max_{tuiles}]$ le nombre total de zones configurables et la fonction $nbTuiles(z)$ qui renvoie le nombre total de tuiles, $N_z \in [1, Max_{tuiles}]$, pour la zone z. Les entiers Min_{tuiles} et Max_{tuiles} représentent respectivement, le nombre minimal et maximal de tuiles pour une architecture donnée.

Le nombre N_z détermine la granularité de configuration d'une zone. Un plan de configuration avec $Z = Max_{tuiles}$ composé de zones avec $N_z = 1$ permet un accès aléatoire du contrôleur de configuration à chaque tuile de l'architecture. A l'inverse, un plan de configuration avec $Z = 1$ et $N_z = Max_{tuiles}$ implique une reconfiguration totale de l'architecture pour en changer la fonctionnalité. L'exploration du domaine de conception consiste à chercher un compromis idéal entre ces deux extrêmes

Une zone a par défaut un seul contexte de configuration (sans inclure le *contexte courant*, voir section 4.3.2). Théoriquement une zone z peut avoir un nombre (C_z) illimité de contexte

```
1 <ConfigZone named=''{Nom}'' origin=''{Coordonnées}'' corner=''{Coordonnées}''
2 context=''{Nombre de contextes}''\>
```

Listing 4.3 – Définition d'une zone reconfigurable dans une syntaxe XML.

et est donc borné dans l'intervalle $C_z \in [1, +\infty[$. En pratique, lors de la génération matérielle ce nombre est borné.

Dans la figure 4.9, les espaces EC1, EC2 et EC3 représentent les espaces de conception pour une même architecture avec un nombre de tuiles différent. Le nombre de contextes est ici borné à quatre.

Modélisation objet. Une zone reconfigurable est modélisée par une seule classe, *ConfigZone*, qui a pour attributs :

- **Domaine** : cet attribut est un objet de type rectangle dont la position est caractérisée par son point d'origine (coin inférieur gauche) et son point d'arrivée (coin supérieur droit). Il détermine la surface (en nombre de tuiles) occupée par la zone dans le plan de configuration de l'architecture.
- **Nom** : cet identifiant permet d'étiqueter les éléments du MPCalc et ainsi de connaître à quelle zone de configuration ils appartiennent.
- **TailleConfig** : cet attribut précise la taille du binaire de configuration pour cette zone. La taille globale est calculée par la somme des tailles des mots de configuration de chaque tuile d'une zone.
- **Contexte** : le nombre de contextes disponibles pour cette zone.
- **LargeurConfigIn** : cet attribut caractérise la taille du port d'entrée de la mémoire de configuration de la zone.

Description et outil graphique Les instances de *ConfigZone* sont produites à partir d'une description des zones dans une syntaxe XML (voir code 4.3). *Named* spécifie le nom de la zone reconfigurable définie. Le rectangle assigné à l'attribut domaine est construit à partir des deux coordonnées *origin* et *corner*.

La figure 4.10 montre notre outil graphique *Domain Viewer* de modélisation d'un plan de configuration. Les zones reconfigurables sont représentées dans la partie droite et ont chacune leur couleur.

Une fois que toutes les zones de configuration ont été spécifiées, le MPConf est produit. Ce modèle est ensuite utilisé à des fins prospectives ou pour la génération VHDL de l'architecture.

4.3.2 Structure matérielle du plan de configuration

A chaque tuile du plan de calcul est associée une mémoire de configuration. Lors de la génération du plan de configuration, ces mémoires sont fusionnées en fonction des zones reconfigurables définies par le concepteur. De ce fait, une zone est générée dans le matériel sous la forme d'une seule et unique mémoire de configuration.

4.3.2.1 Structure de base

La mémoire de configuration est composée de deux contextes de mémorisation. Les mémoires sont mises en œuvre par des registres de type bascule D avec un port de chargement

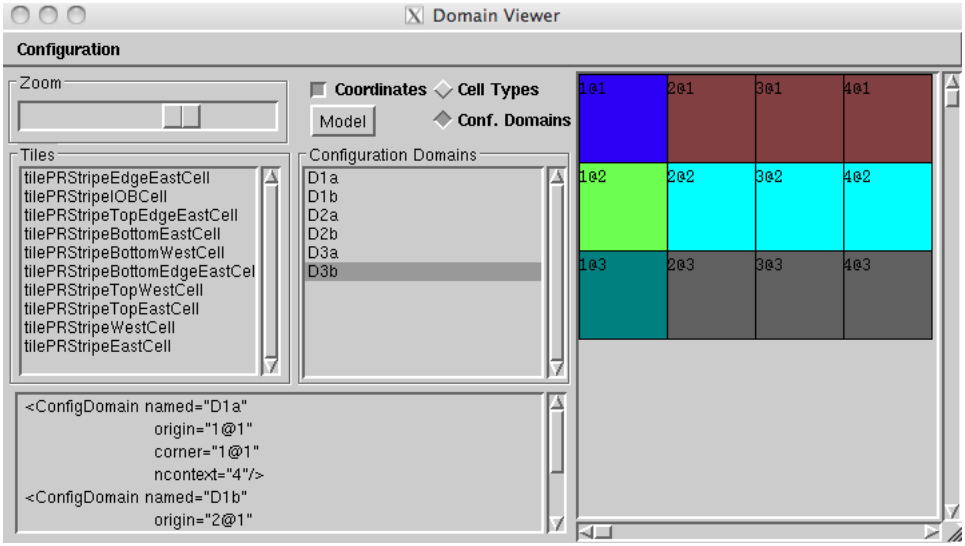


FIG. 4.10 – Définition et visualisation des zones reconfigurables dans notre outil *Domain Viewer*.

synchrone. Ainsi, le chargement d'un registre est synchronisé avec l'état du contrôleur de configuration qui envoie un mot de configuration sur le bus d'entrée de la mémoire. Par défaut, les registres représentés dans la figure 4.11(a) ont une capacité de mémorisation égale à la taille du bus de configuration de la tuile.

Le contexte courant est connecté aux éléments configurables de la tuile par son bus de configuration et mémorise la configuration active. Ce contexte est chargé en parallèle à partir d'un autre contexte (*contexte #1*) configuré par le contrôleur de configuration. Le port *configIn* reçoit la configuration et le port *load* charge le contexte. Le contexte courant est configuré par l'activation du port *enConfig*. Le couplage de ces deux contextes correspond à la technique classique du double-tamponnage et permet la reconfiguration dynamique des zones de l'architecture sans en perturber le fonctionnement. En effet, la reconfiguration en parallèle du *contexte courant* évite un état transitoire instable entre deux configurations et permet un multiplexage temporel des zones reconfigurables.

Dans un cas non contraint le temps de configuration minimal d'un contexte d'une zone est de 1 cycle. Ce temps est possible uniquement si la largeur du port *configIn* est égale à celle du bus de configuration (n). Cependant, en fonction du nombre de tuiles dans une zone, la taille du bus de configuration dépasse rapidement celle des bus mémoire alimentant le plan de configuration. Par conséquent, la taille du port *configIn* doit être ramené à une taille inférieure pour être utilisable.

Dans ce cas, il est nécessaire de sérialiser l'envoi du *bitstream* dans la mémoire par une division en mots de configuration. En fonction du découpage, la mémoire de configuration est générée par le dimensionnement, la réplication et l'aboutement du motif de base représenté par la figure 4.11(a). Une mémoire de configuration prenant en compte la sérialisation est illustrée par la figure 4.11(b). Ainsi, la taille du bus de configuration (n dans 4.11(a)) est réduite (n/r dans 4.11(b) avec r le nombre de réplications) et de ce fait la taille des registres. La réplication et l'aboutement reconditionnent le *contexte #1* en un registre à décalage commandé par le contrôleur de configuration. Le port *load* décale les mots de configuration dans la mémoire et

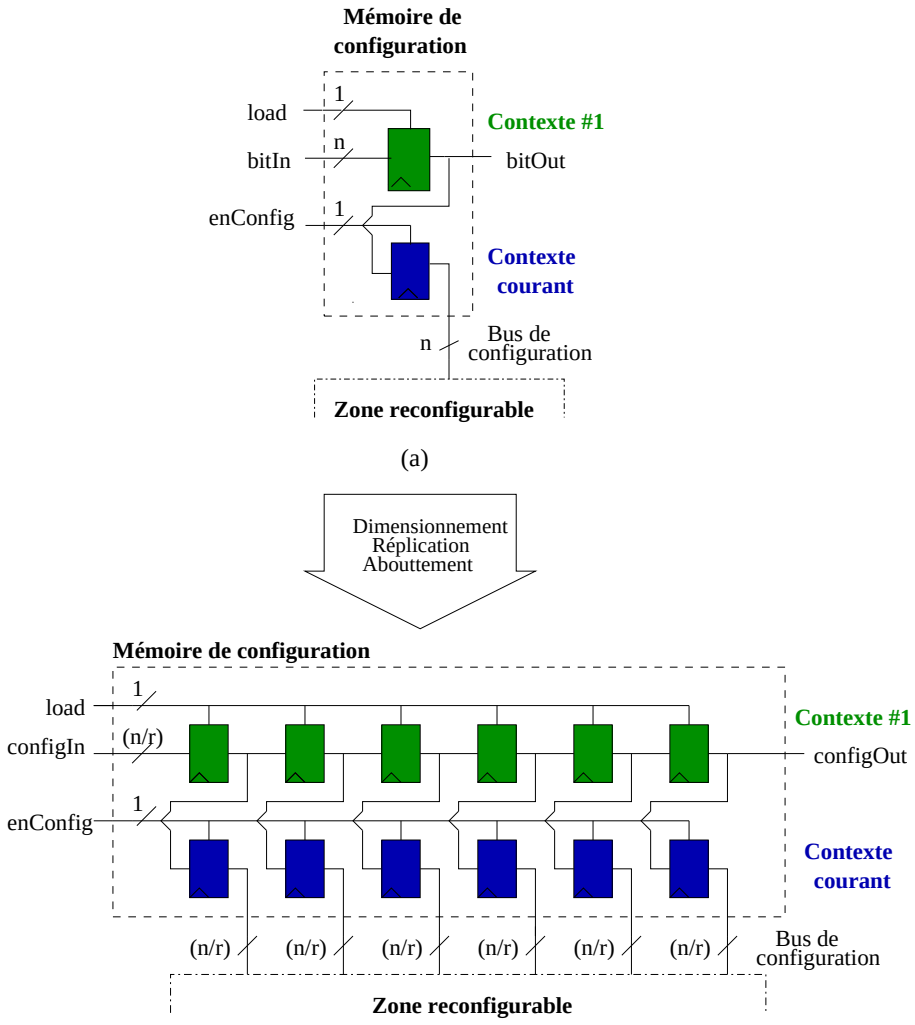


FIG. 4.11 – Réplication d'un motif de base pour la génération d'une mémoire de configuration reconfigurable dynamiquement.

le port *enConfig* active le chargement en parallèle du contexte courant.

Nombre de réplications pour zone. L'attribut *LargeurConfigIn* d'une zone conditionne le nombre de registres nécessaire pour la mémoire de configuration. Par conséquent, le nombre maximal de réplication R_z pour une zone z est obtenu par l'équation suivante :

$$R_z = \lceil \frac{TailleConfig_z}{LargeurConfigIn_z} \rceil \tag{4.2}$$

$LargeurConfigIn_z$ correspond à la largeur du port *configIn* de la zone i et $TailleConfig_z$ à la taille du binaire de configuration pour la zone z.

Si $TailleConfig_z$ et $LargeurConfigIn_z$ ne sont pas divisibles, le nombre de réplications est égale à l'arrondi supérieur ce qui implique un nombre de bits supplémentaire dans la mémoire. Ce *padding* est ajouté au *bitstream* sous forme de 0 et des registres supplémentaires sont ajoutés à la mémoire. La taille en nombre de bits du *padding*, P_z pour une zone z, est caractérisée par la formule suivante :

$$P_z = (R_z \cdot LargeurConfigIn_z) - TailleConfig_z \tag{4.3}$$

Temps de configuration d'une zone. R_z conditionne également le temps de configuration d'un contexte d'une zone puisqu'à chaque cycle, un registre mémorise une valeur (chargement synchrone). Le temps de configuration D_z d'une zone z est caractérisé par la formule :

$$D_z = R_z \cdot C \tag{4.4}$$

La constante C est le délai du contrôleur de configuration pour envoyer un mot de configuration dans la zone.

4.3.2.2 Mémoire de configuration multi-contextes.

Le nombre de contextes est dimensionnable suivant les mêmes principes que ceux énoncés précédemment. La figure 4.12(a) montre la structure de base d'une mémoire reconfigurable multi-contextes. Un port additionnel, *nContextIn*, prend en entrée le numéro du contexte qui sera configuré par la configuration prise en entrée dans *configIn*. Ce port spécifie également quel contexte sera chargé dans le contexte courant par l'activation de *enConfig*.

Le numéro du contexte est envoyé à chaque mémoire de configuration par le port *nContextIn*.

4.3.3 Contrôleur de configuration microprogrammable

Les modes de reconfiguration partielle et multi-contextes augmentent la complexité de séquencement des configurations pour réaliser un calcul. Une illustration de cette complexité est donnée par l'architecture *Piperench* [50] qui multiplexe ses ressources pour l'exécution des étages d'un DFG.

Notre approche réduit cette complexité par une spécification à haut-niveau du séquencement de l'envoi des configurations et de leurs activations (voir section 4.4.3). La description est compilée puis utilisée pour programmer un contrôleur de configuration microprogrammable qui se charge du séquencement des actions.

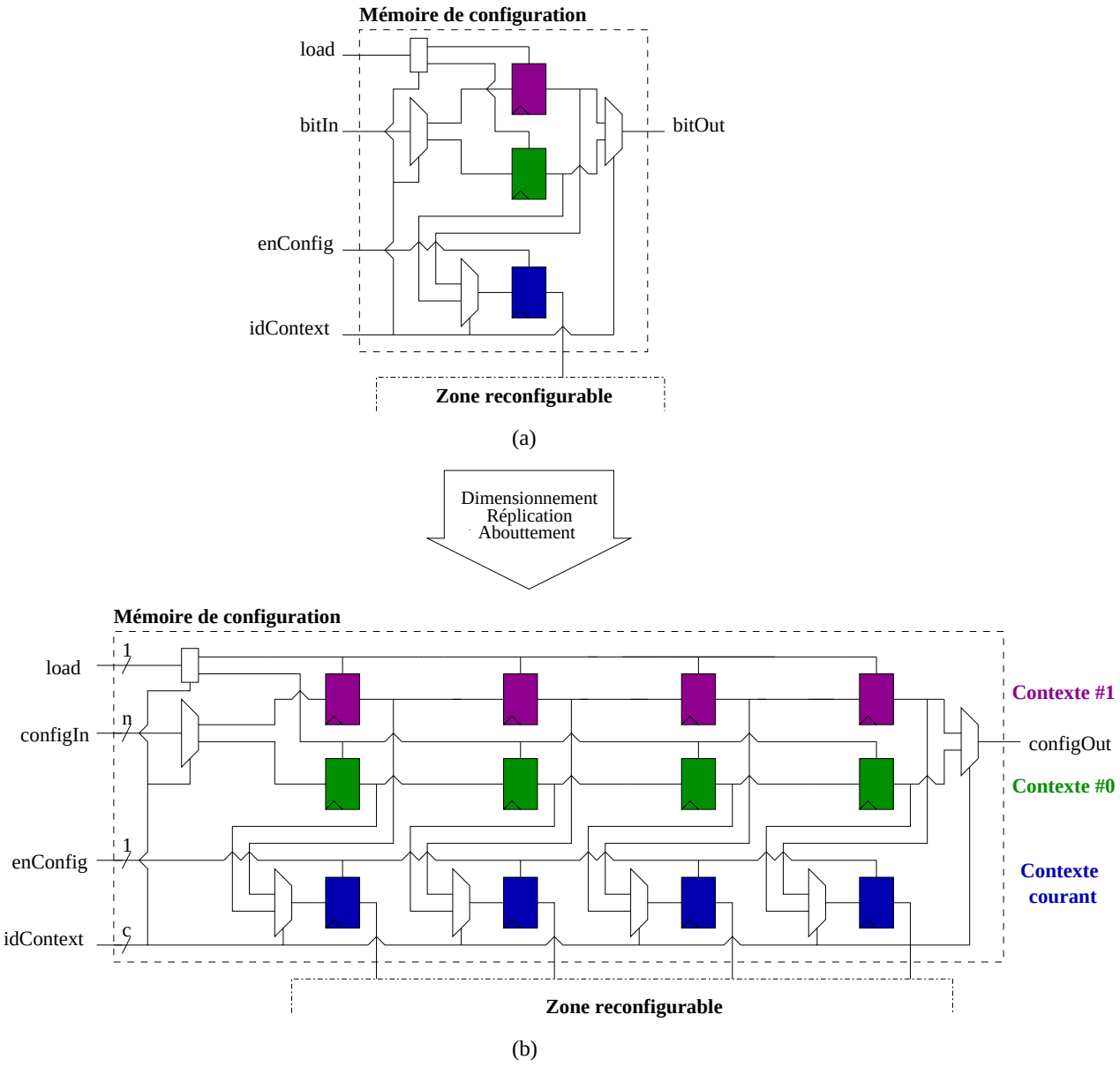


FIG. 4.12 – Architecture d'une mémoire de configuration multi-contextes. Le motif de base est répliqué en fonction du dimensionnement du plan de configuration.

4.3.3.1 Description du contrôleur

Le contrôleur de configuration est préexistant en tant que module. Il est spécialisé automatiquement en fonction du modèle de plan de configuration par le nombre de zones reconfigurables, la largeur de leur port *configIn* et leur nombre de contextes (voir figure 4.13).

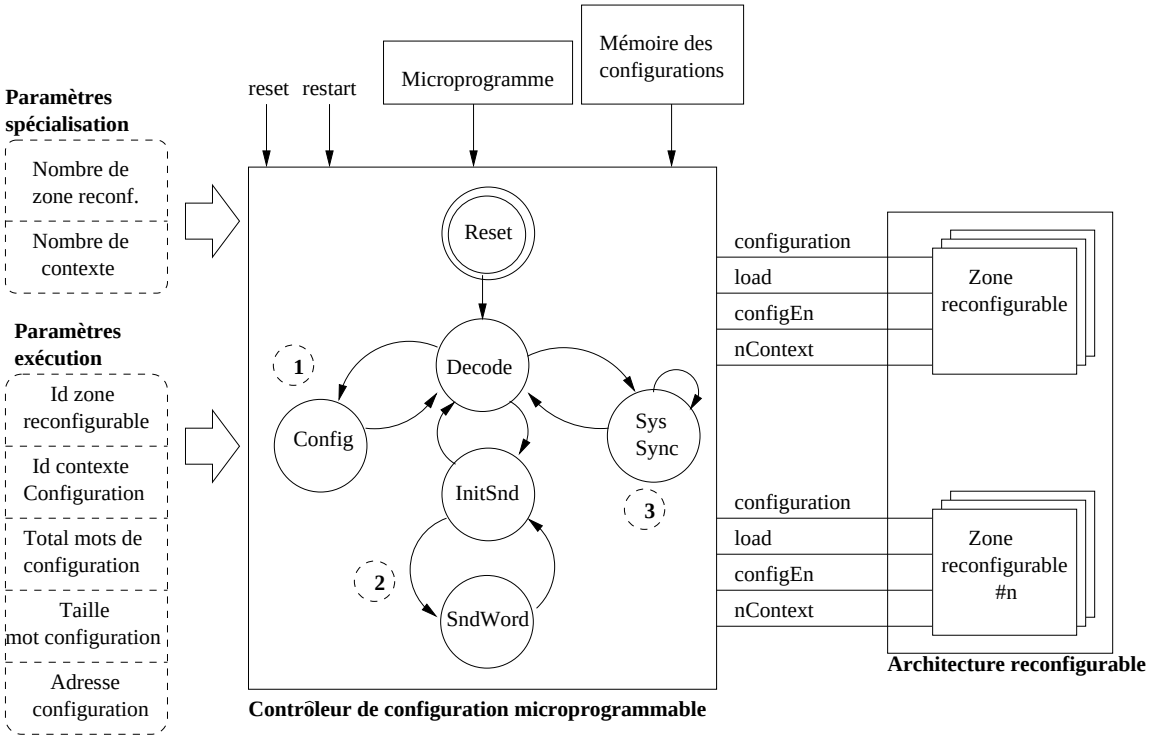


FIG. 4.13 – Le contrôleur de configuration est mis en œuvre par une machine à états finis.

L'envoi des configurations dans les zones reconfigurables de l'architecture est conditionné par les instructions d'un microprogramme. Les instructions de configuration sont codées par cinq mots qui spécifient le numéro de la zone reconfigurable, le numéro du contexte à configurer, la taille d'un mot de configuration en nombre de bits, le nombre total de mots et l'adresse de la configuration dans le cache de configuration. La structure d'une instruction de configuration est donnée par le tableau suivant :

[a-b]	[c-d]	[e-f]	[g-h]	[i-j]
Zone	Contexte	Taille mot	Total mots	Adresse de la configuration

Les tailles de chaque champ sont inférées en fonction des caractéristiques du MPConf (nombres de contextes, de zones, etc.). Elle sont utilisées pour la génération des instructions du programme du contrôleur (voir section4.4.3).

La figure 4.13 définit le comportement du contrôleur de configuration par une machine à états finis. Le décodage d'une instruction (état *Decode*) conditionne l'exécution en fonction de trois types d'instruction.

1. Le premier type correspond à l'état *Config*. Cette instruction configure le contexte courant d'une zone par l'activation de son port *configEn*. La zone et le contexte à configurer sont sélectionnés par les champs *Zone* et *Contexte* de l'instruction.

2. Le deuxième type correspond à l'envoi d'une configuration dans un contexte d'une zone reconfigurable. L'état *InitSnd* est un état de préparation à l'envoi du mot de configuration par la mise à 0 du signal de décalage des mots de configuration. L'arrêt de la configuration est également déterminé dans cet état. L'état *SndWord* correspond au chargement du mot de configuration dans la zone par l'activation du port *load*. Il faut donc 2 cycles au contrôleur pour envoyer un mot de configuration. Lorsque la configuration est complète l'instruction suivante est exécutée (*Decode*).
3. Une instruction spécifique permet la synchronisation du contrôleur avec le système dans lequel est intégrée l'architecture reconfigurable. L'objectif est de permettre la relecture des résultats produits par l'architecture configurée. Ainsi, un processeur hôte peut contrôler l'initialisation des entrées et la relecture des sorties de la matrice reconfigurable. Ce type de synchronisation est également utile pour les phases de débogage et de validation tardive de l'application. Ce comportement est similaire à celui du point d'arrêt tel qu'il est défini dans la programmation logicielle. L'exécution d'une instruction de synchronisation se traduit par la mise en attente du contrôleur dans l'état *Sys Sync*. L'exécution se poursuit une fois que le signal *restart* du contrôleur passe à la valeur 1. Ce signal provient d'une source externe, tel qu'un processeur hôte.

4.3.3.2 Simulation RTL du contrôleur microprogrammable

La figure 4.14 montre le chronogramme de simulation du contrôleur microprogrammable. L'ordre d'exécution est : configuration du contexte 0 de la zone 0, configuration du contexte courant de la zone 0 par le contexte 0 et attente système. Les pins de sortie prennent leurs valeurs de sortie après l'activation du signal *enConfig*.

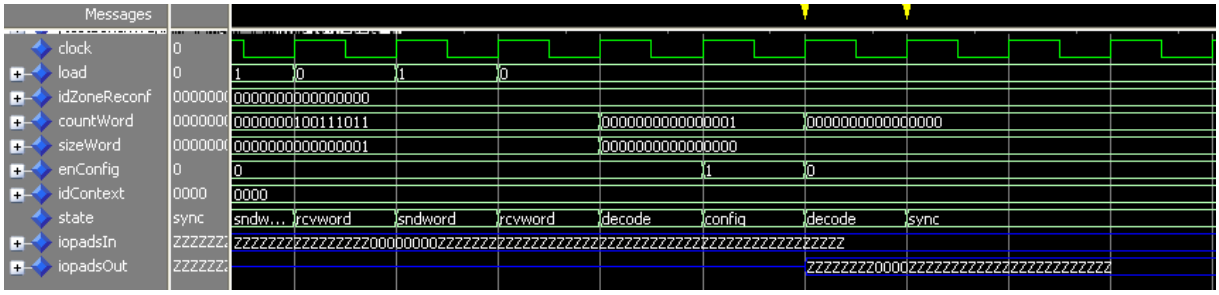


Fig. 4.14 – Simulation du contrôleur microprogrammable pour la configuration du contexte 0 de la zone 0.

4.4 Synthèse des binaires de configuration

Cette section décrit l'ajout aux outils de *back-end* de Madeo-BET d'un générateur de binaires de configuration pour les architectures générées en VHDL. Le générateur s'appuie sur les informations d'allocation de ressources du placement-routage et sur le modèle de *bitstream* généré pour un MPCalc donné (voir figure 4.2). Ce dernier étant généré conjointement au MPCalc, il couvre automatiquement les changements de paramètres de l'architecture modélisée.

4.4.1 Modélisation des configurations

La structuration du *bitstream* d'une tuile est inférée à partir de ses sous-éléments configurables. Le modèle de configuration est construit en même temps que la génération du MPCalc. Les mots de configuration des éléments atomiques sont ajoutés en premier au modèle puis leurs positions sont allouées récursivement dans les hiérarchies supérieures.

La figure 4.15 montre un exemple de structuration hiérarchique d'un modèle de *bitstream* pour une architecture de type FPGA (voir figure 4.5).

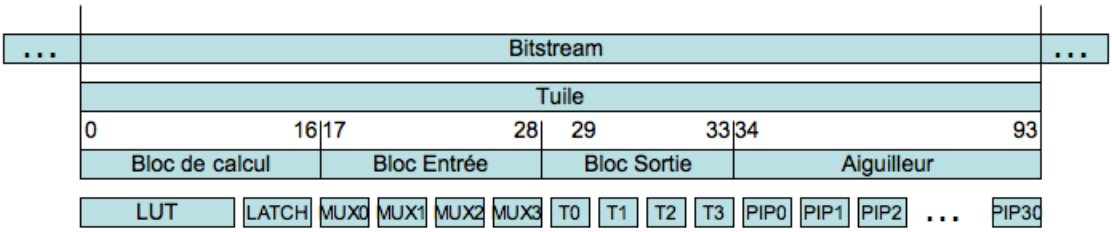


FIG. 4.15 – Le *bitstream* est modélisé comme une hiérarchie de mots de configuration.

Le modèle de *bitstream* final est équivalent à une table de symbole qui effectue le lien entre les éléments du MPCalc, ceux du modèle Madeo-BET et le *bitstream* généré. Ce lien permet également de relire un *bitstream* et de réallouer les ressources d'un modèle Madeo-BET. Cette fonctionnalité offre un moyen d'éditer un *bitstream* existant.

4.4.2 Extraction du bitstream

L'extraction du *bitstream* est réalisée en deux étapes principales par le générateur.

Configuration du MPCalc. La première étape génère pour chaque tuile d'un modèle Madeo-BET un mot de configuration pour sa tuile correspondante dans le MPCalc. La génération du mot de configuration de la tuile est effectuée par un parcours en profondeur des éléments configurables. Pour chaque élément, un mot est généré en fonction de la nature de l'élément (LUT, multiplexeur, etc) puis assigné dans le MPCalc en fonction du modèle de configuration.

Extraction du bitstream. Lorsque le MPCalc est entièrement configuré le binaire de configuration est extrait par un parcours de chaque zone reconfigurable définie dans le MPConf. Ce parcours suit l'interconnexion des mémoires de configuration des tuiles comme illustré par la figure 4.16. Les mots de configuration des tuiles sont concaténés suivant cet ordre pour former un *bitstream* pour chaque zone.

Les binaires de configuration de chaque zone sont ensuite ordonnancés pour une exécution sur l'architecture.

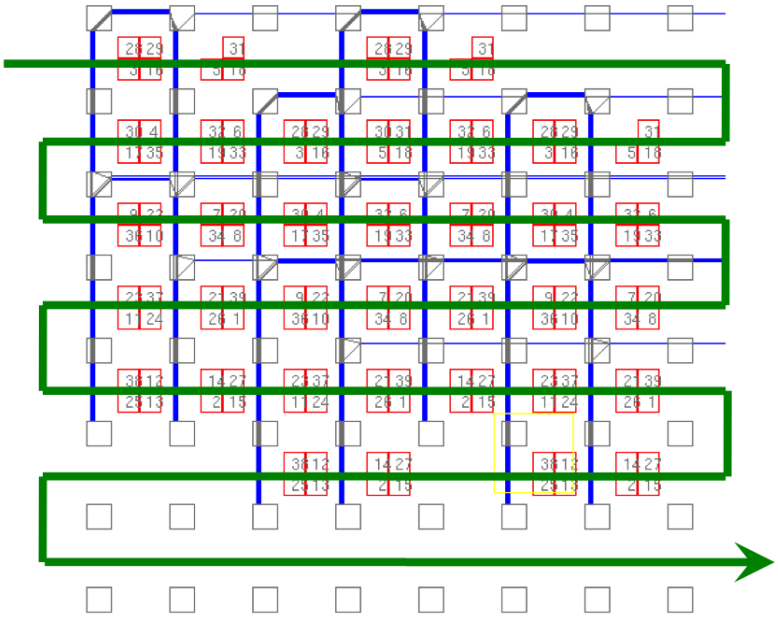


FIG. 4.16 – Extraction d'une configuration à partir du placement-routage sur un modèle Madeo-BET. Les carrés numérotés sont les LUT allouées et les lignes les ressources de routage.

4.4.3 Programmation du contrôleur de configuration microprogrammable

L'ordonnancement des pages de configuration est effectuée manuellement par le concepteur. Pour abaisser la complexité de la tâche, une interface graphique a été développée. Les pages de configuration et les instructions de configuration, activation et synchronisation sont ajoutées dans une liste qui définit l'ordre séquentiel des opérations à réaliser. La figure 4.17 montre un exemple de programme. Les pages sont situées dans la liste *Configuration pages* et les instructions dans *Scheduling*. Les instructions de configuration (qui correspondent au nom de la page à configurer dans la liste *Scheduling*) et d'activation (*activ* exécutée dans l'état *Config* du contrôleur) sont étiquetées avec le numéro de la zone et du contexte ciblé. Le contexte à configurer ou à activer est précisé par le concepteur lorsqu'une page est ordonnancée. Le bouton *Generate* déclenche la génération du microprogramme à partir des informations de la liste *Scheduling*.

4.5 Conclusion

La convergence rapide vers une solution architecturale passe par l'évaluation itérative de plusieurs variantes. Cette tâche nécessite la mise à disposition d'outils qui réduisent la complexité de l'exploration.

Ce chapitre a présenté l'outil DRAGE pour l'exploration et le prototypage d'unités reconfigurables embarquées. Il vient étendre le cadre de modélisation et de placement-routage Madeo-BET par les capacités suivantes :

– Modélisation de la reconfiguration partielle et multi-contextes.

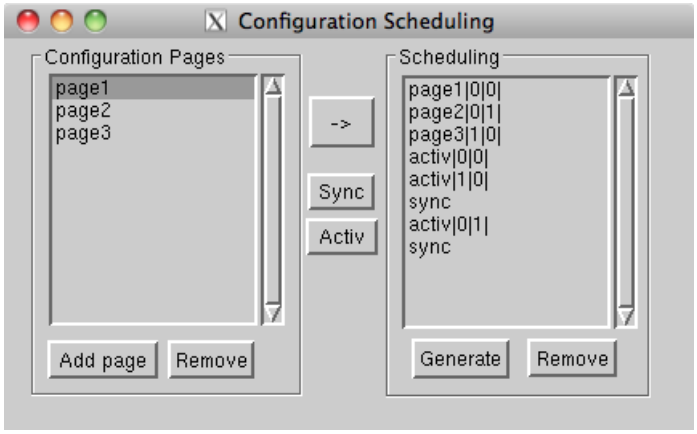


FIG. 4.17 – Les pages de configuration sont chargées dans l'interface graphique puis ajoutées manuellement dans la liste d'ordonnancement.

- Génération de l'unité reconfigurable et de son contrôleur de configuration microprogrammable en VHDL comportemental.
- Production de *bitstreams* pour l'architecture générée.

La modélisation d'une architecture est composée de deux parties principales qui sont, la description du plan de calcul et celle du plan de configuration. La première spécifie les ressources de l'architecture et leurs interconnexions dans le langage de description d'architectures Madeo-ADL. La seconde est un découpage du plan en zones reconfigurables dynamiquement et multi-contextes dans une syntaxe XML. La modélisation séparée des deux plans permet une exploration des types de ressources de calcul indépendamment des capacités de reconfiguration dynamique de l'architecture.

A partir des deux plans modélisés, l'architecture est générée en VHDL comportemental pour être simulée. Un contrôleur de configuration microprogrammable est spécialisé en fonction du plan de configuration. Il est programmé à haut-niveau via une interface graphique qui facilite l'ordonnancement des *bitstreams*. Ceux-ci sont produits à partir du placement-routage d'une application sur le modèle Madeo-BET de l'architecture et d'un modèle de *bitstream* conforme au plan de calcul.

Le bénéfice principal du flot est la garantie d'une forte productivité par la mise à disposition rapide de prototypes simulables d'unités reconfigurables. L'architecture est explorée à haut-niveau via les modèles des plans puis à un niveau matériel par la simulation.

L'exploration des performances d'une architecture passe par la simulation d'applications exécutées. Par conséquent, la facilité de programmation est également un facteur de productivité. Le chapitre 5 présente un flot de programmation haut-niveau qui cible les unités reconfigurables d'un RSoC.

Chapitre 5

Flot de programmation pour architectures reconfigurables embarquées

Le chapitre précédent a présenté un flot de prototypage d'architectures reconfigurables embarquées par une approche ADL. La modélisation à un haut-niveau d'abstraction permet une plus grande productivité des développeurs. En effet, la complexité matérielle ainsi que les développements fastidieux sont absorbés par la génération automatique du prototype ainsi que de ses environnements de programmation (partie basse) et de simulation.

L'évaluation des prototypes générés par DRAGE requiert la simulation de l'exécution d'applications. Par conséquent, la programmation des architectures ne doit pas être un obstacle à la productivité des concepteurs.

Dans ce chapitre, l'approche haut-niveau et orientée logiciel développée tout au long de ces travaux est appliquée à la programmation d'unités reconfigurables embarquées. Le flot de programmation ne cible pas uniquement les unités générées et s'interface également avec des chaînes d'outils préexistantes.

5.1 Introduction

La conception d'applications pour les architectures reconfigurables nécessite de la part du concepteur des connaissances en conception matérielle. De plus, dans un cadre SoC les parties logicielles résultantes d'un partitionnement sont également à prendre en considération. La multiplication des niveaux d'abstraction, le manque de modèle de calcul côté matériel et les différents formalismes à maîtriser, tout cela couplé à l'augmentation en complexité des cibles limitent la productivité des concepteurs. La complexité matérielle doit être masquée par la mise à disposition de formalismes de haut niveau fondés sur des modèles de calcul sur lesquels le concepteur peut raisonner. La compilation/synthèse de la description haut-niveau automatise la mise en œuvre de l'application sur un modèle d'exécution (une cible matérielle).

La mise en œuvre d'une application est décomposable en trois couches illustrées par la figure 5.1.

Le modèle d'exécution correspond à l'architecture ciblée par le flot de programmation. Le prototypage de cette couche en terme d'environnement de simulation et de partie basse

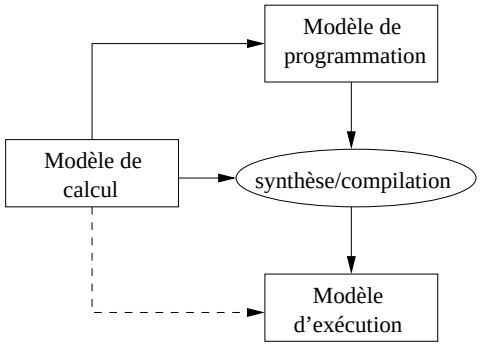


FIG. 5.1 – Décomposition de la mise en œuvre d'une application en trois couches.

de programmation est développé dans le chapitre 4. Le modèle de calcul définit de manière abstraite et formelle l'ensemble des règles pour la réalisation d'un calcul. La description d'une application, conforme au modèle de calcul, est réalisée par l'intermédiaire d'un modèle de programmation. Le flot de synthèse/compilation met en œuvre le modèle de programmation sur le modèle d'exécution. Le support des primitives du modèle de calcul au niveau de l'architecture est variable voire inexistant.

Dans ce chapitre nous présentons un flot de programmation d'unités reconfigurables embarquées. Ces travaux sont principalement fondés sur les activités de l'équipe LabSTICC/AS dans le cadre du projet européen MORPHEUS [124].

Les applications sont spécifiées sous la forme de réseaux de processus *Communicating Sequential Process* (CSP) [66] via un modèle de programmation fondé sur le langage C [122]. Les modèles d'exécution ciblés correspondent aux unités reconfigurables hétérogènes d'un RSoC. Nous abordons les différentes approches génériques pour la mise en œuvre d'un modèle de calcul commun sur les unités.

5.1.1 Modèle de calcul pour les applications

Il existe de nombreux modèles de calcul orientés vers le calcul spatial. L'un des plus répandus dans le domaine de l'embarqué est le modèle de réseaux de processus de Kahn [74]. Les calculs sont structurés en processus séquentiels communiquant de manière asynchrone par des canaux FIFO unidirectionnels. D'autres modèles sont fondés sur le concept d'acteur [4] tels que les graphes de flot de données synchrones [94]. Ils sont un cas particulier de graphes flot de données avec une spécification *a priori* du nombre de données produites et consommées par chaque nœud. Le modèle de calcul synchrone [101] est principalement utilisé dans le contexte des systèmes temps-réel. Les échanges de données sont synchronisés sur une horloge globale afin de garantir le déterminisme du système.

Pour tirer profit du parallélisme spatial intrinsèque des architectures reconfigurables, nous utilisons le modèle de calcul CSP [66]. Ce choix est justifié par une longue série de travaux déjà présents dans le laboratoire et initiés par le projet Armen [22].

Les applications sont décrites sous forme d'un réseau de processus qui communique de manière synchrone (sémantique de *rendez-vous*) par des canaux. Ce modèle présente l'avantage d'être adapté à la modélisation de systèmes distribués et de supporter leur vérification formelle. Par exemple, la détection des « verrous » dans le réseau de processus.

Le parallélisme temporel et spatial est exploité à plusieurs niveaux de granularité. Au

niveau réseau les accès mémoire et les calculs sont chacun réalisés par des processus spécifiques indépendants. Ainsi ces deux activités se recouvrent temporellement. Par la suite nous nommerons « processus de communication » les processus en charge des accès en mémoire et « processus de calcul » les processus en charge de réaliser des opérations sur les données.

Les calculs sur les données lues en mémoire sont décrits par des graphes de flot de données. Ainsi, il est possible d'activer en parallèle des opérateurs indépendants et d'obtenir un recouvrement temporel par la mise en place d'un *pipeline*.

La figure 5.2(a) illustre l'organisation d'un réseau de processus. Les processus sont interconnectés par des canaux de communication CSP, c'est-à-dire que la sémantique de synchronisation est de type *rendez-vous*. Le processus central est un processus de calcul alimenté par deux processus de communication. Ce schéma est extensible avec l'ajout de plusieurs processus de différents types. Un processus réalise des calculs mis en œuvre par un graphe d'opérateurs. Les arcs du graphe représentent les dépendances de données.

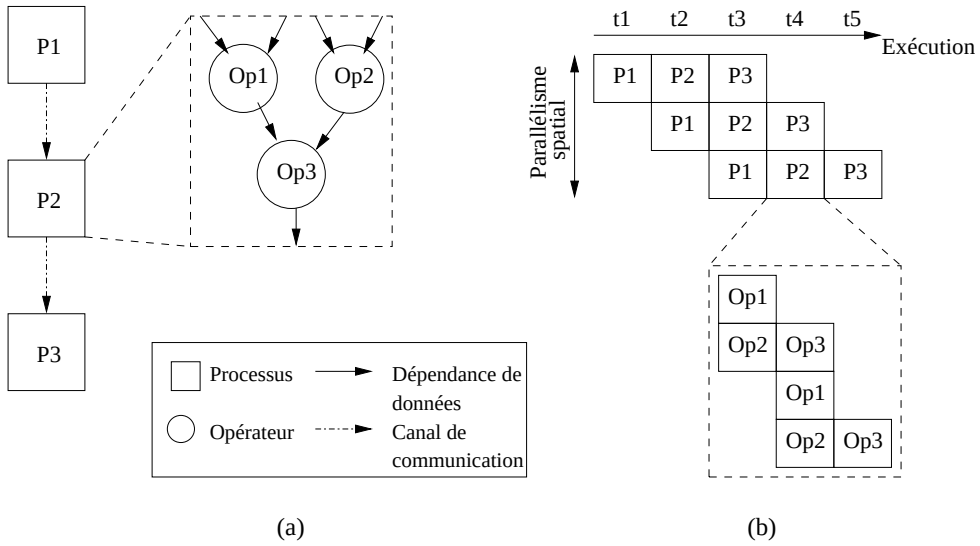


FIG. 5.2 – Organisation et exécution d'un réseau de processus.

Le réseau de processus exploite à plusieurs niveaux le parallélisme spatial et temporel. L'exécution des processus et des opérateurs est illustrée par la figure figure 5.2(b). Les trois processus s'exécutent en parallèle au temps *t3*. De la même manière, les opérateurs d'un processus s'exécutent en parallèle lorsqu'ils n'ont pas de dépendance de données (parallélisme spatial) ou alors par recouvrement temporel.

Le modèle de programmation développé pour la description de ces réseaux de processus est détaillé dans la section 5.2.

5.1.2 Modèle d'exécution dans un SoC

Le modèle d'exécution des architectures reconfigurables embarquées est défini par une intégration système dans le SoC. Nous considérons un modèle d'exécution fondé sur des travaux récents sur la conception d'un RSoC et de son environnement de programmation [124]. La figure 5.3 illustre un exemple d'intégration de deux unités reconfigurables dans un SoC.

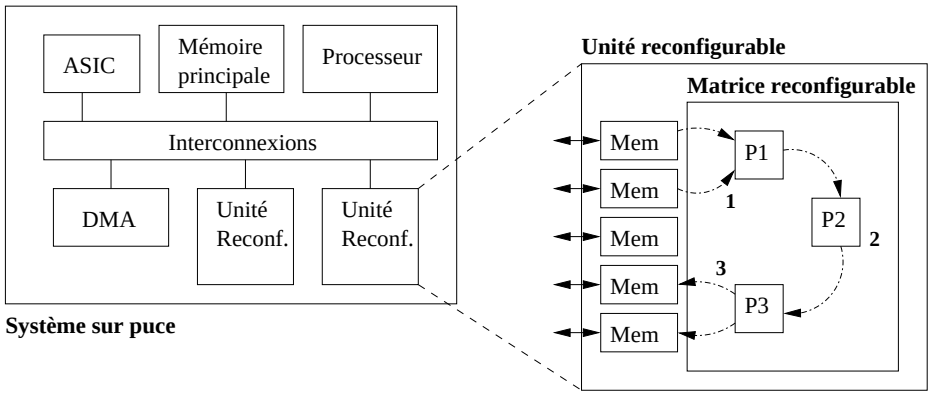


FIG. 5.3 – Détail d'une unité reconfigurable dans un système sur puce.

La matrice reconfigurable est interfacée avec le système par un ensemble de mémoires locales pour former une unité reconfigurable. Ces mémoires contiennent les données prises en entrée des calculs et également les résultats produits. Au niveau du SoC ces mémoires sont lues et écrites par un contrôleur d'accès mémoire (DMA) qui transfert les données depuis/vers la mémoire principale.

Au niveau de l'unité reconfigurable, l'exécution d'un réseau de processus est décomposable en trois étapes principales données par la figure 5.3. Ces trois étapes sont :

1. Lecture des données dans les mémoires locales par le processus de communication P1. Elles sont prises en entrée par le processus de calcul P2.

2. Le processus de calcul P2 opère sur les données et produit un résultat communiqué au processus P3.

3. Le processus P3 écrit les résultats dans les mémoires locales.

Recouvrement temporel des communications. Pour minimiser les pénalités du goulot d'étranglement mémoire [145] dans le SoC, les accès du contrôleur DMA aux mémoires locales de l'unité reconfigurable sont masqués par la technique des *doubles mémoires*. A chaque processus de communication est associé un couple de mémoires sur lesquelles sont alternées les écritures et les lectures par le contrôleur DMA et le processus. Ainsi les fonctions accélérées accèdent aux données calculées dans le cycle précédent sans risque de violation de flot de données. Cette technique permet d'obtenir un recouvrement temporel, d'une part au niveau système entre l'exécution accélérée et les transferts DMA, et d'autre part au niveau unité reconfigurable entre les tâches accélérées qui communiquent via les mémoires locales. Cette optimisation est indépendante du modèle de programmation et est appliquée automatiquement par la partie basse du flot.

Ce modèle d'exécution fondé sur les mémoires locales est commun à l'ensemble des unités reconfigurables du SoC. En revanche, la mise en œuvre des deux types de processus est dépendante de la nature de la matrice reconfigurable et de son modèle d'exécution. Cet aspect est pris en charge par la partie basse du flot de programmation.

5.1.3 Flot de programmation

La partie haute du flot de programmation prend en entrée une description de l'application sous forme d'un réseau de processus (cf. figure 5.4). Le formalisme développé et utilisé est une variante syntaxique du langage de composition Avel [122]. Il a été initialement conçu comme une branche connexe du langage NetGen qui est actuellement à la base de travaux pour la modélisation de réseaux de capteurs et les calculs massivement parallèles [70, 76]. Les différences de Avel par rapport à NetGen sont l'intégration de comportements hétérogènes associés aux processus et une structuration hiérarchique des réseaux. Dans le cadre des travaux présentés dans ce manuscrit, Avel est utilisé pour la capture d'applications du traitement du signal.

Un réseau est composé de deux types de processus : calcul et communication. Les processus de calcul sont décrits dans un sous-ensemble du langage C, nommé AvelC, spécialisé pour la description de graphes de flot de données. Les processus de communication sont décrits en XML et spécifient les motifs d'adressage des mémoires locales. Enfin, l'ensemble des processus sont interconnectés par une description XML qui décrit d'une part les mémoires connectées aux processus de communication et d'autre part les canaux reliant les processus entre eux (voir section 5.2).

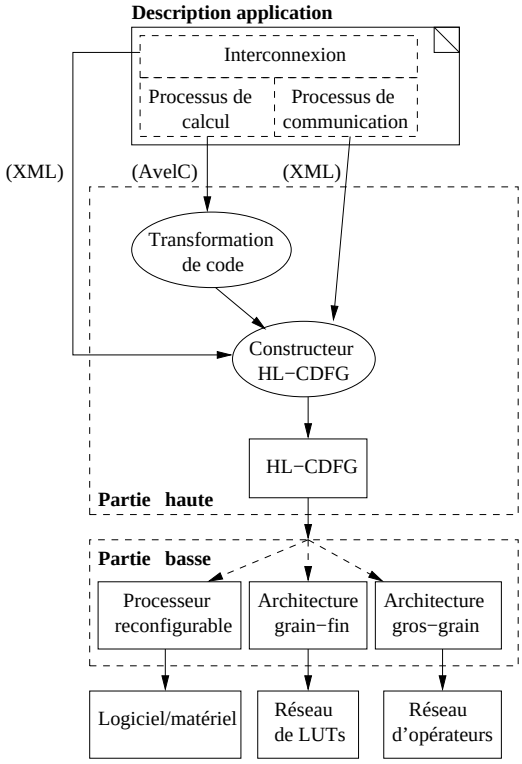


FIG. 5.4 – Flot du frontal de haut niveau de Madeo+.

Le flot de programmation utilise deux langages, C et XML, standardisés et très répandus dans le monde logiciel. Ainsi le concepteur d'application n'a pas à apprendre de nouveaux langages et la description est réalisée à un haut-niveau d'abstraction. De plus, ces langages bénéficient de nombreux supports en termes de librairie de programmation permettant la

```
process [nom du processus](chanin [type] [nom du canal], [...],
                           chanout [type] [nom du canal]){
 [Déclarations des variables locales]

[Comportement du processus]
}
```

Listing 5.1 – Déclaration d'un processus.

génération du comportement des processus par des outils tiers. Ainsi, la partie haute du flot de programmation peut être interfacée avec d'autres outils de description d'applications (e.g. saisie graphique du réseau de processus).

Les arbres de syntaxe abstraite des trois descriptions sont utilisés pour produire la représentation intermédiaire de l'application prise en entrée de la partie basse. Avant cette étape, des transformations de code sont appliquées sur la partie calcul pour obtenir la description d'un DFG (voir section 5.3.3). La représentation utilisée dans ces travaux est une variante de graphe de flot de contrôle et de données (CDFG) adaptée pour la représentation de réseaux de processus CSP (voir section 5.3). Le CDFG produit à partir de la description du réseau de processus sera ensuite spécialisé ou transformé dans la partie basse pour cibler des modèles d'exécution.

La partie basse du flot de programmation cible trois types majeurs d'architectures reconfigurables intégrées comme unités dans un SoC. Le CDFG est spécialisé par transformation de modèle pour cibler les différents modèles d'exécution. La section 5.4 détaille la mise en œuvre des réseaux de processus pour les trois modèles d'exécution : logiciel/matériel, réseau de LUT et réseau d'opérateurs à gros-grain.

5.2 Formalisme de description en réseau de processus

Une application est décrite par le concepteur sous la forme d'un réseau de processus de calcul et de communication. Le formalisme Avel permet la déclaration, l'interconnexion et la composition hiérarchique des processus.

5.2.1 Les processus de calcul

Les fonctions accélérées correspondent à des tâches exécutées intensivement (aussi qualifiées de « points chauds ») et extraites d'applications du traitement du signal.

Dans le but de conserver le lien entre l'aspect langage et le modèle de calcul CSP mis en œuvre dans le CDFG, la syntaxe de déclaration des processus de calcul s'appuie sur le modèle de programmation du langage OCCAM [35]. Le code 5.1 donne la structure générale de déclaration d'un processus.

Les canaux d'entrée et de sortie sont déclarés dans l'en-tête de la fonction. La direction d'un canal est déterminée par les mots-clés *chanin* pour une entrée et *chanout* pour une sortie. Les calculs effectués par les tâches sont orientés flot de donnée, ils sont donc spécifiés dans un sous-ensemble de C proche du langage Griffy-C [114]. Ce langage restreint l'utilisation des opérations de contrôle et de certaines opérations arithmétiques telles que la multiplication et la division. En revanche, le système de type utilisé est identique au langage C et peut être affiné par un typage au bit près. Cependant, l'usage des pointeurs est proscrit car inutile dans ce cadre d'utilisation. En effet, le modèle de calcul CSP impose la localité des données,

par conséquent une fonction ne peut avoir d'effet de bord. L'ensemble des restrictions a pour objectif de rendre possible la synthèse de la spécification comportementale.

Pour permettre la prise en compte des opérations de communications par canaux entre processus il est nécessaire d'enrichir le langage. Ces opérations sont effectuées par l'ajout des opérateurs de réception « ? » et d'envoi « ! » identiques à ceux des langages OCCAM et Handel-C [25]. Les mots-clés de séquencement tels que « seq » et « par » ne sont pas utilisés dans le langage. En effet, dans le but de rester le plus proche possible d'une syntaxe standard C et à cause du cadre applicatif (DSP) traité, nous avons choisi de garder un paradigme séquentiel pour la programmation de la fonction accélérée. L'exploitation du parallélisme d'instruction est réalisée par l'outil de synthèse lors de la phase d'ordonnancement du DFG.

Pour plus d'informations concernant les restrictions du langage, le lecteur peut se référer au document de référence de Griffy-C [113]. Pour la syntaxe OCCAM, le lecteur peut se référer à [97] et également au manuel Handel-C pour une syntaxe CSP orientée langage C [25].

Exemple. Le code 5.2 donne un exemple de processus de calcul qui effectue une somme de différences absolues définie par la formule :

$$SAD = \sum_{x=0,y=0}^{N_x,N_y} |a_{x,y} - b_{x,y}| \tag{5.1}$$

Les variables a et b représentent les pixels de deux matrices de taille $N_x \times N_y$. Cette opération est utilisée pour le calcul de distance entre deux blocs de pixels dans l'étape de *motion estimation* pour l'encodage vidéo MPEG [139].

Les opérations élémentaires de différence absolue puis la somme des différences sont incluses dans une structure *repeat*. Le code est répliqué 4 fois pour traiter les deux matrices. Cette solution a pour avantage de compacter le code. En revanche, l'exploitation du parallélisme spatial est limité par la dépendance de donnée créée par l'accumulation. En dépit d'un modèle de programmation séquentielle, l'exploitation du parallélisme spatial par une gestion des dépendances de données entre les opérateurs est laissée à la charge du développeur. Une autre solution consiste à ne pas utiliser d'accumulation pour paralléliser les opérations de différences et de sommes dans le processus de calcul. Le parallélisme spatial est également exploitable à un niveau hiérarchique supérieur par un réseau de processus *sad4*.

La corps du processus ne contient aucune information sur son nombre d'activations. Ce paramètre est dépendant du nombre de données traitées à chaque activation et est donc calculé en fonction de la description des processus de communication. Le corps du processus est intégré dans une boucle uniquement au niveau de la représentation intermédiaire.

5.2.2 Les processus de communication

Les processus de communication sont des tâches répétitives qui accèdent aux données en mémoire par des motifs d'adressage multi-dimensionnels. A titre d'exemple, nos travaux se rapprochent du langage Array-OL [40, 48] et de son environnement de programmation visuelle GASPARD [23]. Cependant, Avel considère comme atomiques les données qui transitent sur les canaux CSP contrairement à Array-OL qui utilise des vecteurs.

Les accès mémoire multi-dimensionnels sont décrits dans le langage XML. Le listing 5.3 montre trois processus de communication qui seront connectés à la fonction du code 5.2. Les

```
1  process sad4(chanin uchar in1, chanin uchar in2, chanout uchar out){
2    unsigned char <8> sub1a, sub1b, sub1;
3    short int <8> add, p1, p2;
4
5    add = 0;
6
7    repeat(4){
8      in1 ? p1;
9      in2 ? p2;
10
11     sub1a = p1 - p2;
12     sub1b = p2 - p1;
13     sub1 = (sub1a < 0) ? sub1b : sub1a;
14
15     add = add + sub1;
16   }
17
18   out ! add;
19 }
```

Listing 5.2 – Exemple d'une somme de différences absolues entre deux matrices 2x2 de pixels. L'opération de base est répliquée 4 fois.

processus sont déclarés dans la balise « AGS » et sont définis par les attributs de « AG » :

- *name* : qui donne le nom du processus ;
- *input* ou *output* : qui détermine si le processus lit (données prises en entrée) ou écrit (sortie de résultats) dans sa mémoire locale ;
- *datatype* : qui qualifie le type des données lues ou écrites.

Les motifs d'accès aux données sont spécifiés dans la balise « CONFIG » par un pas de lecture/écriture (attribut *step*) et le nombre total de données (*count*).

Dans l'exemple du code 5.3, les deux processus d'entrée effectuent des accès à deux dimensions sur les données alors que le processus de sortie écrit les résultats linéairement en mémoire.

Impact de la description sur la synthèse. Le nombre de dimensions pour les accès mémoire n'est pas restreint et permet ainsi d'exprimer tous les motifs possibles. Cependant ce nombre a un impact sur les caractéristiques surface et performance de l'application en fonction de la cible choisie. Par exemple, dans le cas d'une architecture FPGA, les processus seront directement mis en œuvre dans le matériel ce qui limitera les pertes de performance. En revanche, un nombre important de dimensions nécessite de la logique de contrôle ce qui augmentera la surface consommée. Dans le cas d'une architecture bénéficiant de générateurs d'adresses matériels, telle que le DREAM, les processus seront générés sous forme de configurations pour les générateurs d'adresse avec la prise en compte de la limite en dimension. Les générateurs d'adresses du DREAM sont limités à deux dimensions ce qui implique la mise en œuvre des dimensions supplémentaires sur le processeur hôte. De ce fait le nombre d'exécutions autonomes sur l'accélérateur, c'est à dire sans les retours de contrôle au processeur hôte, est diminué, ce qui augmente les pénalités d'activation de la fonction accélérée (voir section 5.4.1).

```
<AGS>
<AG name='input0' output='p1' datatype='uchar'>
        <CONFIG init='0' step='1' count='4'>
          <CONFIG step='1' count='256'>
          </CONFIG>
        </CONFIG>
</AG>

<AG name='input1' output='p2' datatype='uchar'>
        <CONFIG init='0' step='2' count='4'>
          <CONFIG step='3' count='256'>
          </CONFIG>
        </CONFIG>
</AG>

<AG name='output0' input='chanOut' datatype='uchar'>
        <CONFIG init='0' step='1' count='1024'>
        </CONFIG>
</AG>
</AGS>
```

Listing 5.3 – Description des schémas d'accès aux données en XML.

5.2.3 Coordination globale des processus

L'application forme un réseau de processus structuré en étages à un niveau hiérarchique supérieur. Un étage est composé d'une ou plusieurs fonctions accélérées et de leurs contrôleurs d'accès mémoire. Ces étages sont interconnectés par des mémoires locales comme le montre la figure 5.5.

Les interconnexions entre les étages sont décrites par le langage de coordination Avel adapté pour une syntaxe XML [122]. L'exemple du code 5.4 montre la définition de trois étages (balise « STAGE ») dont les deux premiers appliquent un premier traitement (ici un filtre numérique pour l'exemple) puis envoient les résultats à un autre étage qui effectue une somme de différences absolues.

Un étage prend en paramètres les mémoires d'entrée et de sortie (attributs *inputDEBs* et *outputDEBs*) qui sont utilisées par ses processus de communication. Chaque processus (balise « AG », « FUNCTION ») d'un étage prend en paramètres la connexion de sa sortie (*connections*), le nom de son comportement pour effectuer le lien lors de la génération du CDFG (le comportement est inséré dans le corps du nœud hiérarchique *ProcessNode*) et le nom de sa mémoire pour lire/écrire les données. Les connexions (attribut *connections*) sont spécifiées par le nom du processus de destination et le port de destination.

La balise « MAIN » définit la topologie globale du réseau avec les déclarations des étages et leurs interconnexions. A chaque étage on donne en paramètre les noms des mémoires locales pour réaliser les communications entre étages.

5.3 Représentation intermédiaire pour la synthèse/compilation

Pour permettre la génération de code bas niveau ou la synthèse matérielle à partir de spécifications de haut niveau, les outils développés dans nos travaux s'organisent autour d'un modèle intermédiaire qui est un CDFG. Ce modèle représente les algorithmes dans un format

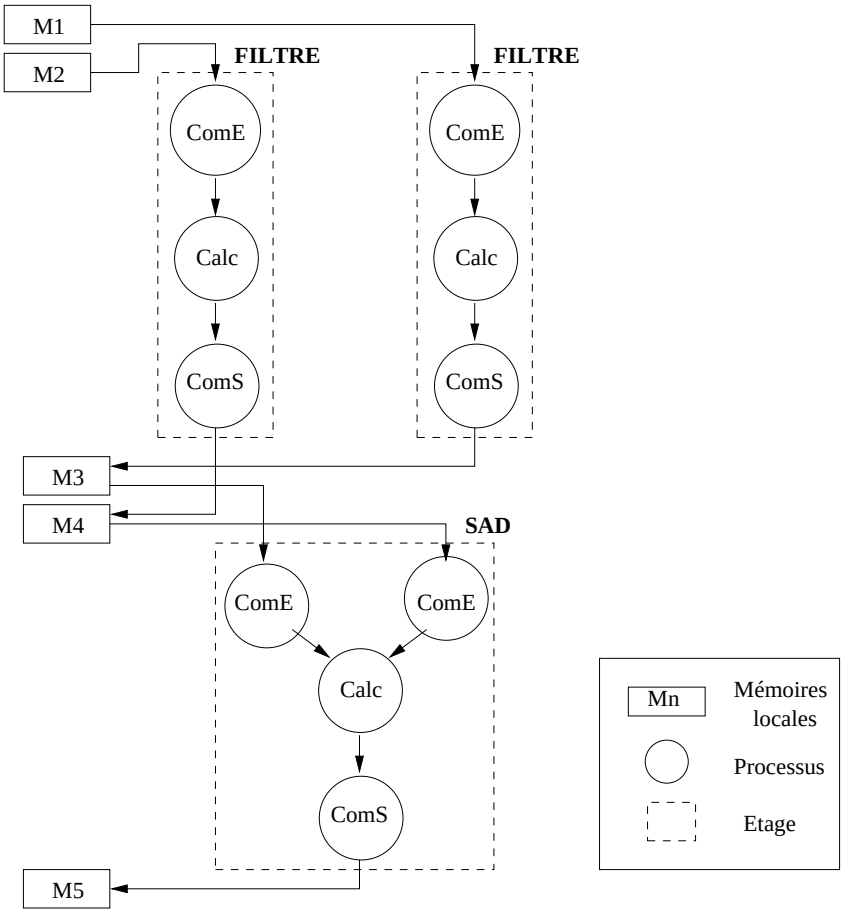


FIG. 5.5 – Vision hiérarchique d'une application spécifiée sous forme d'un réseau de processus.

```
1  <NETWORK>
2  <STAGE name='SAD' inputDEBs='D0, D1' outputDEBs='D2'>
3    <AG name='CtrlIn0'
4        connections='Fsad4.in1'
5        behaviour='input0'
6        memory='D0'></AG>
7    <AG name='CtrlIn1'
8        connections='Fsad4.in2'
9        behaviour='input1'
10       memory='D1'></AG>
11   <FUNCTION name='Fsad4'
12             connections='CtrlOut.chanOut'
13             behaviour='sad4'></FUNCTION>
14   <AG name='CtrlOut'
15       connections=''
16       behaviour='output0'
17       memory='D2'></AG>
18 </STAGE>
19
20 <STAGE name='FILTRE' inputDEBs='D0' outputDEBs='D1'>
21   <AG name='CtrlIn'
22       connections='Ffiltre.in1'
23       behaviour='input0'
24       memory='D0'></AG>
25   <FUNCTION name='Ffiltre'
26             connections='CtrlOut.chanOut'
27             behaviour='seuil'></FUNCTION>
28   <AG name='CtrlOut'
29       connections=''
30       behaviour='output0'
31       memory='D1'></AG>
32 </STAGE>
33
34 <MAIN>
35   <STAGE name='FILTRE' inputDEBs='M1' outputDEBs='M3'></STAGE>
36   <STAGE name='FILTRE' inputDEBs='M2' outputDEBs='M4'></STAGE>
37   <STAGE name='SAD' inputDEBs='M3, M4' outputDEBs='M5'></STAGE>
38   </MAIN>
39 </NETWORK>
```

Listing 5.4 – Coordination des groupes de processus pour former un réseau complet.

indépendant des outils et des langages. Il sert de support pour l'application de transformations telles que l'ordonnancement des opérations.

Le modèle de CDFG présenté dans ce chapitre est développé par l'équipe LabSTICC/AS dans le cadre du projet MORPHEUS [86, 124]. Il capture les informations suivantes :

- les dépendances de flot de données entre les opérations ;
- les modifications du flot de contrôle par les opérations de type itérations et conditionnelles ;
- l'ordonnancement des opérations ;
- les informations de type concernant les données consommées et produites par les opérations.

5.3.1 Graphes de flot de contrôle et de données hiérarchiques

Cette section présente la modélisation des dépendances de données puis des instructions de contrôle. Le modèle objet du CDFG hiérarchique est ensuite détaillé.

5.3.1.1 Modélisation des dépendances de données

Les dépendances de données au sein d'un algorithme définissent des contraintes sur l'ordre d'exécution des opérations. Pour réaliser des transformations *légales* il est nécessaire de respecter ces contraintes [2].

Il existe une dépendance de données entre deux opérations Op_i et Op_j lorsqu'une variable est partagée entre ces deux opérations. On distingue trois types de dépendances :

- les dépendances de flot : Op_j lit une variable écrite par Op_i ;
- les anti-dépendances : Op_j écrit dans une variable après sa lecture par Op_i ;
- les dépendances de sortie : Op_i et Op_j écrivent dans la même variable.

Les anti-dépendances et les dépendances de sortie sont éliminées par renommage des variables. En revanche les dépendances de flot sont conservées et déterminent l'ordre d'exécution des opérations. Si une dépendance de flot de données existe entre Op_i et Op_j alors ces deux opérations ne peuvent pas être exécutées en parallèle. Op_j ne peut s'exécuter qu'à condition que Op_i soit terminée.

Les dépendances de flot de données entre les opérations d'un algorithme sont modélisées par un DFG. Un graphe flot de données est défini comme suit :

Définition 5.3.1. *Un graphe flot de données est un graphe orienté acyclique noté* $G_{DFG}(Ops, E_d)$*, où l'ensemble des nœuds* $Ops = \{op_i; i = 1, ...n_{ops}\}$ *est l'ensemble des* n_{ops} *opérateurs de l'algorithme d'entrée et l'ensemble des arcs* $E_d = \{(op_i, op_j); i, j = 1, ..., n_{ops}\}$ *représente les dépendances de flot. Un arc dirigé* $e_{ij} = (op_i, op_j)$ *existe dans* E_d *si une donnée produite par l'opérateur* op_i *est consommée par* op_j *[54].*

5.3.1.2 Modélisation hiérarchique des instructions de contrôle

Traditionnellement les informations de contrôle d'un algorithme sont représentées par un graphe dont les nœuds sont des *blocs de base* connectés par des arcs qui représentent le flot de contrôle [2]. Un bloc de base est une séquence d'instructions consécutives dans lequel le flot de contrôle n'est pas modifié par les instructions. La modélisation du contrôle utilisée dans ces travaux est différente et n'intègre pas les notions de blocs de base et d'arcs de flot de contrôle.

Les instructions qui modifient le flot de contrôle telles que les boucles ou les instructions conditionnelles sont modélisées par des nœuds hiérarchiques. Ils permettent de conserver les informations concernant la structure de l'algorithme. Ces nœuds peuvent encapsuler d'autres hiérarchies ou contenir uniquement des nœuds atomiques tels que les opérateurs. Au niveau d'un nœud hiérarchique, ses sous-opérateurs (hiérarchiques ou atomiques) sont vus comme des opérateurs atomiques.

Du point de vue de la conception objet, le modèle des nœuds hiérarchiques et atomiques est conforme au schéma de résolution *Composite* [6]. Cette vision permet d'appliquer de manière homogène sur tous les nœuds des algorithmes de transformation comme l'ordonnancement.

Les hiérarchies sont représentées dans les graphes par des boîtes qui contiennent des sous-graphes (voir figure 5.8(d)).

5.3.1.3 Composition du flot de données et du flot de contrôle

Les CDFG associent dans une même structure le flot de contrôle et les dépendances de données de type *flot* [116] [106] [54]. Le modèle de CDFG utilisé dans ces travaux se compose des nœuds hiérarchiques qui représentent les instructions de contrôle, ainsi que la structure du programme, et des nœuds atomiques qui représentent les opérateurs. Les nœuds du graphe sont connectés entre-eux par des arcs de flot de données qui seront appelés par la suite *arcs de flot*.

Cette section présente le modèle objet du CDFG.

Les nœuds hiérarchiques. Ces nœuds correspondent aux instructions de contrôle et à la structure du programme. La figure 5.6 montre partiellement l'arbre d'héritage des nœuds hiérarchiques du modèle objet du CDFG.

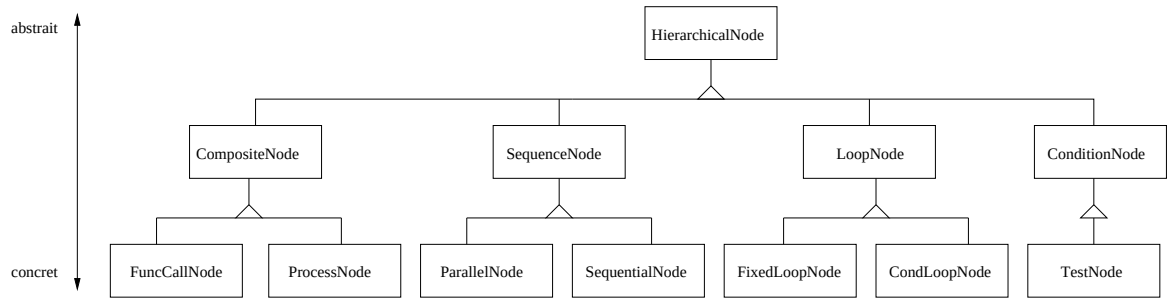


FIG. 5.6 – Hiérarchie de classes du modèle de nœud hiérarchique du CDFG.

La racine de l'arbre de la figure 5.6 représente une hiérarchie abstraite dont la spécialisation est la plus forte au niveau des feuilles. Ce type de nœud contient des sous-graphes qui sont alimentés en données par les entrées de leur hiérarchie. En effet, une donnée ne peut pas être consommée directement par un nœud de calcul si elle n'est pas locale, c'est-à-dire produite par un nœud de calcul contenu dans la même hiérarchie. Par conséquent, pour permettre les re-directions vers les données réelles à partir des hiérarchies, des *alias* sont utilisés. Un nœud hiérarchique définit un ensemble d'entrées et de sorties qui sont des collections d'*alias*.

Les quatre nœuds de la partie centrale de l'arbre d'héritage sont détaillés comme suit :

- **Nœuds de composition/structuration (*CompositeNode*).** Ces nœuds contiennent les informations de structuration du programme comme les appels de fonctions et les processus.
- **Nœuds de séquencement (*SequenceNode*).** Ils contiennent l'information sur le séquencement des opérateurs/hiérarchies dans le graphe. L'exécution est soit parallèle (*ParallelNode*) ou séquentielle (*SequentialNode*). Ces nœuds sont ajoutés lors de l'ordonnancement du CDFG.
- **Nœuds d'itération (*LoopNode*).** Ils représentent les boucles. Ce nœud est spécialisé en boucle dont le nombre d'itérations est fixé par un indice (*FixedLoopNode*) ou est déterminé par une condition d'arrêt (*CondLoopNode*). Ce type de nœud encapsule une hiérarchie qui correspond au corps de la boucle.
- **Nœuds de condition (*ConditionNode*).** Ils représentent les instructions conditionnelles de type *if-then-else* (*TestNode*). Les différentes branches sélectionnables sont représentées par des hiérarchies (e.g. on a deux hiérarchies, *true* et *false*, pour une instruction *if-then-else*).

Les nœuds atomiques Ces nœuds correspondent à des opérateurs spécifiques et constituent l'unité d'exécution dans le CDFG. Une partie de l'arbre d'héritage des nœuds atomiques du modèle objet du CDFG est illustrée dans la figure 5.7.

D'un point de vue structurel, au niveau le plus abstrait, un nœud atomique possède des entrées-sorties. Il applique à ses entrées l'opération définie par sa sémantique et produit éventuellement une sortie.

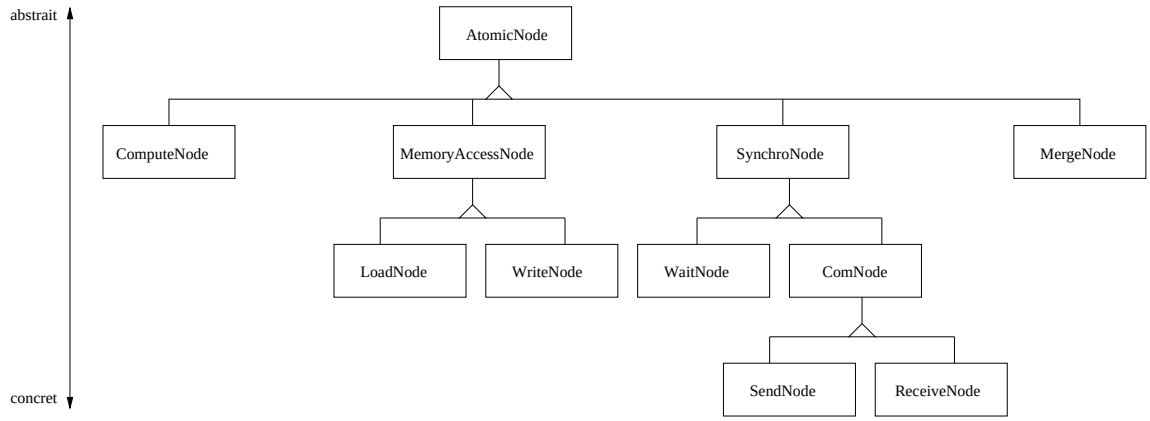


FIG. 5.7 – Hiérarchie de classes du modèle de nœud atomique du CDFG.

Les principaux nœuds atomiques sont :

- **ComputeNode.** Les opérateurs du graphe (e.g opérateurs arithmétiques) sont représentés par ce nœud. Il implémente n'importe quelle opération.
- **LoadNode et WriteNode.** Les écritures/lectures dans des tableaux sont représentées par ces nœuds qui prennent en entrée la référence au tableau et un indice. À un niveau plus proche du matériel ces nœuds correspondent à des lectures dans des mémoires.
- **SendNode et ReceiveNode.** Ces nœuds représentent les communications entre les processus. Ils utilisent des canaux dont la sémantique peut être bloquante ou non-bloquante.

– **MergeNode.** Ce nœud est utilisé pour sélectionner une donnée parmi les sorties de plusieurs hiérarchies, par exemple entre les branches *true* et *false* d'un *TestNode*.

5.3.2 Modélisation de réseaux de processus

Le modèle de CDFG permet la composition de systèmes concurrents grâce à la représentation des processus par des nœuds hiérarchiques et des communications par canaux par des nœuds de communications/synchronisations. Le modèle de calcul utilisé pour ces travaux est CSP de Hoare [66]. Ainsi il permet de représenter les réseaux spécifiés en Avel.

La figure 5.10 montre un exemple de CDFG simplifié qui représente un système de processus communicants. Les trois processus sont représentés par les nœuds hiérarchiques englobants. Les nœuds de communications sont nommés *Snd* pour l'envoi et *Rcv* pour la réception des données. Ils sont reliés par des canaux de communication.

5.3.3 Normalisation en flot de données des calculs

Le comportement d'un processus de calcul du réseau est spécifié sous forme d'un DFG. Un tel graphe prend N données en entrée et produit un seul résultat. Pour permettre la génération de la partie DFG dans le CDFG global, le code AvelC est normalisé par le compilateur. La figure 5.8 donne les différentes étapes de transformation appliquées à une opération de dilatation sur un voisinage de 3 pixels.

La figure 5.8(a) correspond au code AvelC initial spécifié par le programmeur. Les comparaisons des valeurs de pixel sont réalisées deux fois dans la structure itérative *repeat*.

La première transformation (figure 5.8(b)) appliquée est le déroulage du *repeat* par la duplication du corps de l'instruction. Conjointement à cette réécriture le code est normalisé sous forme *3-adresses* [2]. Une instruction *3-adresses* définie au maximum deux adresses d'opérandes en entrée et une adresse pour le résultat de sortie. Cette transformation implique l'insertion de nouvelles variables pour stocker les résultats des opérations intermédiaires (*t_0_1, t_0_2*). Dans le cas d'opérations booléennes de comparaison, les variables sont typées automatiquement, e.g. *t_0_1* a une taille de 1 bit). Dans le cas des autres opérations, les types définis par le programmeur sont propagés.

L'étape suivante (figure 5.8(c)) correspond à la réécriture finale du code pour la génération du DFG. Le code est mis sous une forme à assignation unique (*Static Single Assignment* (SSA)) qui permet d'identifier les dépendances de données [115]. Cette étape applique également : l'élimination du code mort, la propagation des constantes et la résolutions des expressions neutres (e.g. $x \cdot 1$) ou nulles (e.g. $x \cdot 0$) [2].

5.4 Mise en œuvre sur des architectures reconfigurables embarquées

Les sections précédentes ont présenté la partie haute du flot de programmation des unités reconfigurable. Cette section a pour objectif de décrire la partie basse du flot qui cible trois types majeurs d'architectures reconfigurables intégrées dans des unités reconfigurables. Le CDFG est spécialisé pour chaque unité en fonction de la nature de leur architecture reconfigurable. En effet, chaque architecture définit un modèle d'exécution qui contraint la mise en œuvre des réseaux de processus sur les unités. Ces trois modèle d'exécution sont :

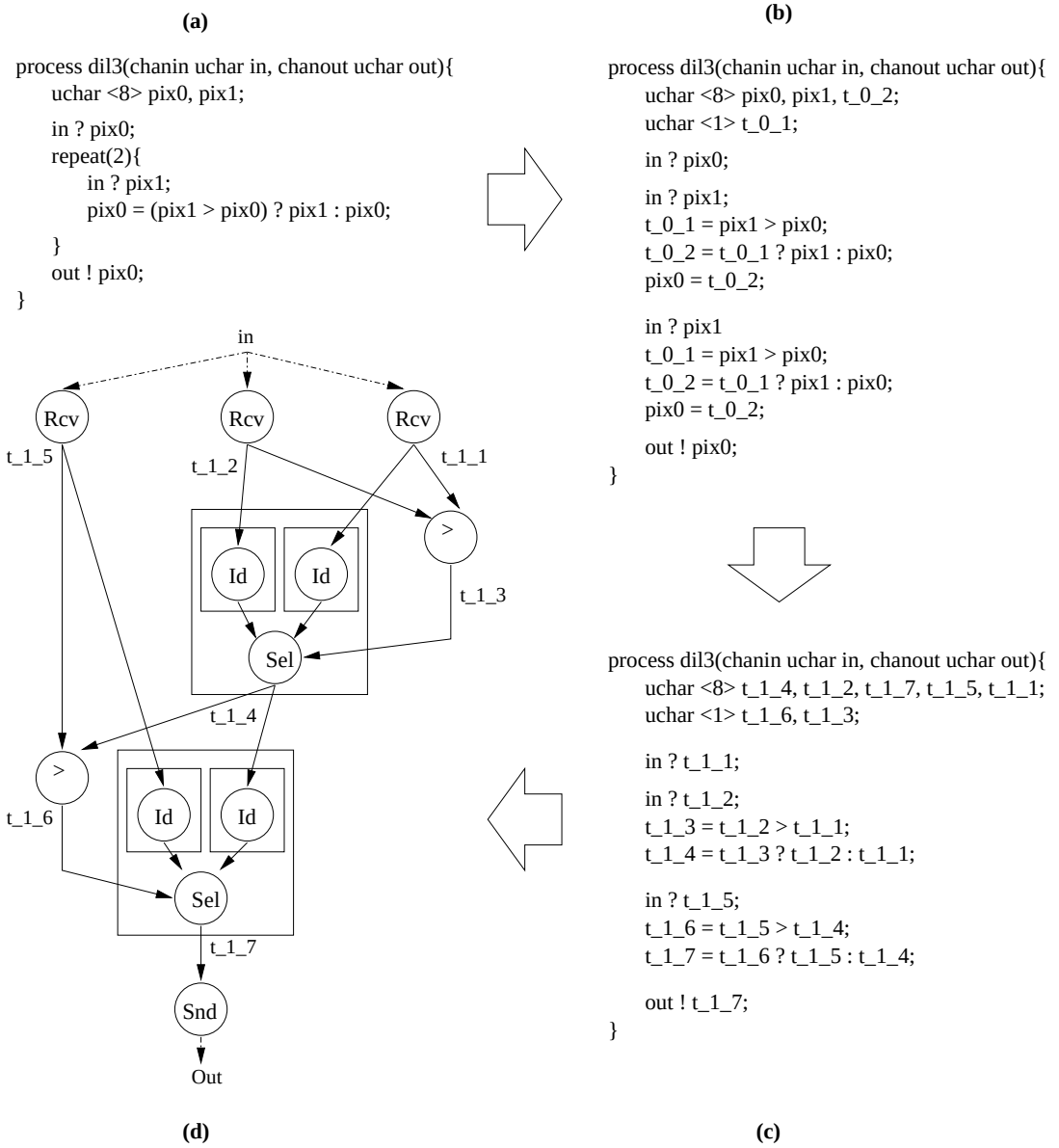


FIG. 5.8 – Transformation du code AvelC des processus de calcul pour la génération d'un DFG.

Logiciel/matériel. Ce modèle d'exécution est principalement utilisée dans le cas d'un couplage processeur-matrice reconfigurable (e.g. Molen [140]). Le processeur gère l'activation des tâches accélérées sur la matrice reconfigurable et configure les accès des tâches aux mémoire locales de l'unité. Ce modèle implique un partitionnement logiciel/matériel de l'application. Notre outil gère la génération de code pour chaque partie et les appels du matériel à partir du logiciel. Comme cas pratique, nous ciblons le processeur reconfigurable DREAM [27].

Réseau de LUT. L'unité ne contient pas de processeur et l'ensemble de l'application est mise en œuvre sur une matrice reconfigurable à grain-fin de type FPGA. Les applications sont synthétisées en logique puis en un réseau de LUT placé et routé sur l'architecture. Notre outil produit un réseau de LUT non-liées à une librairie particulière qui peut cibler soit des architectures modélisées avec Madeo-BET ou des cibles tierces. Dans le premier cas le réseau de LUT est placé/routé par les outils de Madeo spécialisés pour l'architecture modélisée. Puis, un binaire de configuration peut être produit pour le prototype matériel généré.

Réseau d'opérateur à gros-grain. La démarche est identique pour les architectures à gros-grain, excepté que l'application est compilée sous forme d'un graphe d'opérateurs (e.g. additionneurs). Les nœuds du graphe sont directement liés aux opérateurs de l'architecture. Il n'y a donc pas d'étape de synthèse logique. L'architecture que nous considérons est une architecture organisée en étages d'opérateurs à gros-grain qui forment un *pipeline*. Cette organisation est similaire à celle des architectures PiCoGA [98] et Piperench [50].

5.4.1 Compilation pour processeur reconfigurable

Cette section détaille les étapes de mise en œuvre d'un réseau de processus Avel sur un processeur reconfigurable. Nous considérons comme cas d'étude une unité dont l'architecture reconfigurable est le DREAM.

5.4.1.1 Modèle de programmation

L'architecture DREAM [27] est un processeur reconfigurable orienté vers le domaine applicatif du traitement du signal. Il est composé d'un processeur embarqué RISC couplé à une matrice reconfigurable, le PiCoGA-III, qui joue le rôle d'un coprocesseur.

Une matrice d'interconnexion interface les mémoires locales de l'unité reconfigurable avec un ensemble de *Générateur d'Adresses* (GA) qui alimentent la matrice PiCoGA. Chaque GA est dédié à une mémoire locale et permet de réaliser des motifs d'adressage à plusieurs dimensions. La figure 5.9 donne une vue simplifiée de l'architecture DREAM.

Le RISC est programmé par du C standard. Le lien entre le processeur et le PiCoGA est effectué par un ensemble de fonctions (*Application Programming Interface* (API)) qui permettent le chargement et l'activation des traitements sur le PiCoGA, ainsi que la configuration des GA et de la matrice d'interconnexion. Les fonctions accélérées sont programmées dans un sous-ensemble de C, appelé Griffy-C, spécialisé pour la description de graphes de flot de données [114].

Notre flot de programmation prend en entrée un *High-Level CDFG* (HL-CDFG) produit depuis une description Avel. La spécialisation du CDFG pour cibler le DREAM est réalisée

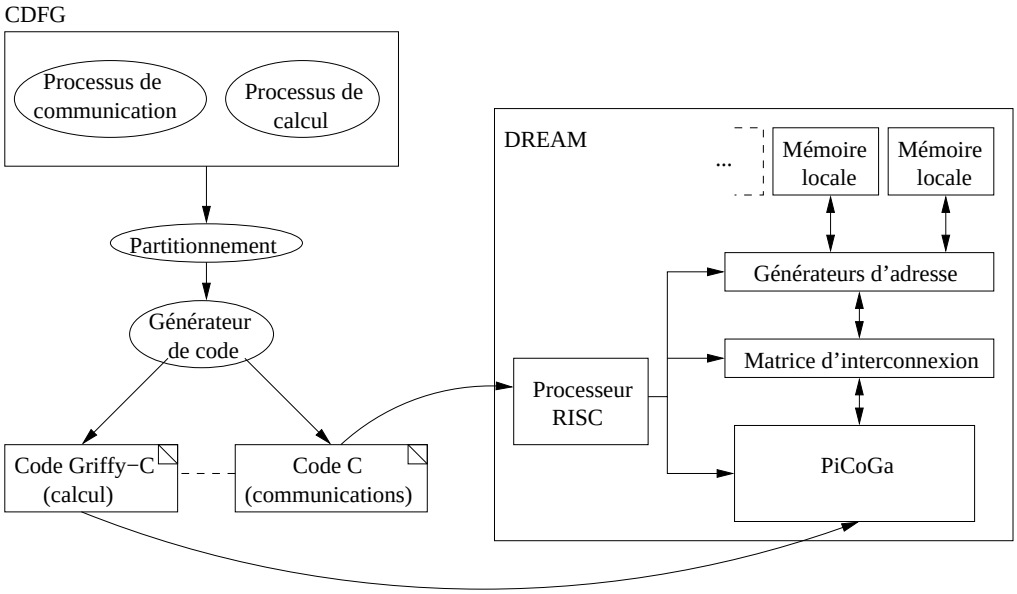


FIG. 5.9 – Synthèse et compilation pour la plate-forme DREAM.

par un partitionnement des processus de communication et de calcul entre le RISC et le PiCoGA. Les processus de communication sont implémentés sur le RISC sous la forme de nids de boucles décrits en C. Leur rôle est de calculer les configurations des GA puis d'activer les fonctions accélérées. A cela s'ajoute la configuration de la matrice d'interconnexion pour sélectionner les mémoires locales connectées aux entrées-sorties de la fonction accélérée. Cette configuration est utilisée pour alterner les accès aux doubles-mémoires.

Les graphes de flot de données des processus de calcul sont générées en Griffy-C pour cibler le PiCoGA. L'ensemble du code généré de l'application est ensuite pris en entrée de la chaîne d'outil du DREAM.

5.4.1.2 Modèle d'exécution

La figure 5.10 donne le flot d'exécution d'une application composée de deux processus de communication et d'un processus central de calcul. Conformément au modèle CSP ces trois processus s'exécutent en parallèle et se synchronisent par les canaux de communication. Cependant, le modèle d'exécution du processeur RISC est de par nature séquentiel. Par conséquent, le parallélisme exprimé dans le CDFG ne peut être supporté complètement.

Le code C destiné au processeur est généré suivant une trame prédéfinie qui décrit un scénario général d'exécution sur le DREAM.

1. Côté système : le contrôleur DMA écrit les données d'entrée dans les mémoires locales.
2. Le processeur RISC commence son exécution lorsque les données ont été écrites.
3. Les fonctions accélérées sont allouées sur le PiCoGA.
4. Initialisation des configurations de la matrice d'interconnexion.
5. Boucle principale d'un processus de communication : Configuration GA d'entrée et de sortie → Activation de la fonction accélérée et production de résultats → barrière de synchronisation.

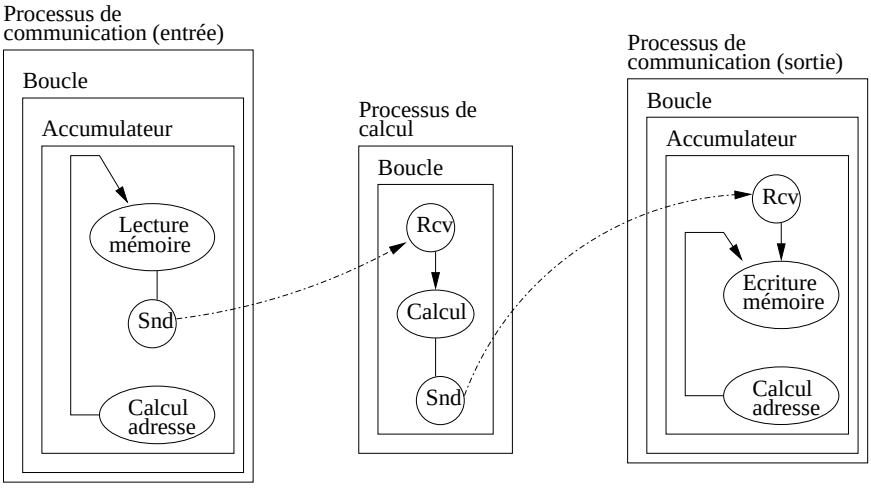


FIG. 5.10 – Exemple de structuration d'une application en processus communicants.

6. Désallocation des fonctions accélérées.
7. Côté système : lecture des résultats finaux par le DMA.

Le flot de génération de code applique un certain nombre de transformations sur les processus de communication. Ces transformations sont de type fusion de boucles entre les processus pour obtenir une exécution séquentielle sur le processeur. L'étape 5 correspond à l'exécution des deux processus fusionnés qui lisent et écrivent les données dans les mémoires locales.

5.4.1.3 Mise en œuvre des processus de communication

Les processus de communications sont mis en œuvre via l'API du DREAM. Cette API se compose de deux fonctions principales qui sont :

Activation de la fonction accélérée. *pga_op(idf, idm, iter)* avec *idf* l'identifiant de la fonction accélérée ; *idm* l'identifiant de la configuration de la matrice d'interconnexion pour la gestion des doubles mémoires et *iter* le nombre d'activation de la fonction.

Configuration des GA. *set_DF(idGA, adr, count, stride, step, le)* avec *idGA* l'identifiant du générateur d'adresses ; *adr* l'adresse de départ pour les lectures/écritures ; *count* le nombre de mots de données par paquet ; *stride* le nombre de mots de données entre deux paquets ; *step* le pas d'accès aux données et *le* le mode lecture ou écriture pour le GA.

La figure 5.11(a) donne un exemple de comportement d'un processus de communication qui alimente les entrées d'un processus de calcul.

La figure 5.11(b) montre le code C généré pour le processus de communication. Chaque niveau de boucle est mis en correspondance avec un paramètre de la fonction *set_DF*. Le calcul d'adresse dans la boucle de niveau 2 correspond au calcul du paramètre *step*. Au niveau 1, le nombre de données entre deux paquets, *stride*, est calculé. Ces deux paramètres permettent de réaliser l'adressage à deux dimensions. L'adresse initiale, *adr*, est définie par l'initialisation de la variable *addr1* avant la boucle de niveau 1.

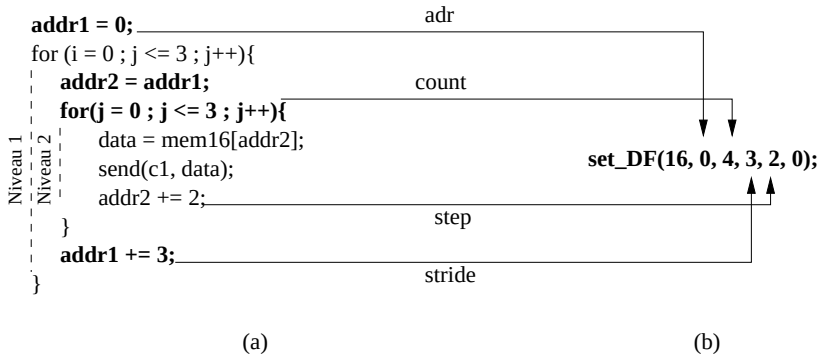


FIG. 5.11 – (a) Comportement d'un processus de communication de type entrée en pseudo-C (b) Mise en œuvre du processus de communication avec l'API DREAM.

Fusion des processus de communication. Dans un programme complet, un processus de calcul prend en entrée des données puis produit des résultats. Ces deux flots de données, en entrée et en sortie, sont gérés par deux processus de communication. Le parallélisme n'étant pas disponible sur le processeur RISC ces deux processus doivent être fusionnés dans le programme C. La figure 5.12 illustre la fusion de deux processus de communication d'entrée et de sortie.

Les configurations de GA résultantes de la fusion correspondent aux niveaux 2 et 3 du processus d'entrée et au niveau 2 du processus de sortie. Les GA du DREAM ne gèrent pas les accès aux mémoires sur plus de deux dimensions (paramètres *stride* et *step*. Le niveau 1 du processus d'entrée est mis en œuvre dans le code C par la boucle englobante.

La mise en œuvre d'un seul niveau de boucle pour le GA de sortie est imposée par les réceptions séquentielles sur un même canal par le processus de calcul. Un résultat est produit lorsque toutes les données ont été reçues. Or, les adresses d'accès, calculées par les GA, sont mises à jour à chaque activation de la fonction accélérée. Par conséquent, une fonction avec R réceptions séquentielles ne produira un résultat que toutes les R activations. Le nombre d'activations, $N_{activation}$, d'une fonction accélérée est donné par la formule :

$$N_{activation} = I_{innerLoop} \times R \tag{5.2}$$

Avec $I_{innerLoop}$ le nombre d'itérations de la boucle interne du processus de sortie. Il correspond également au nombre de données dans un paquet au niveau hiérarchique le plus bas. Puis R le nombre de réceptions séquentielles du processus de calcul. Sur $N_{activation}$ un résultat est produit toutes les R activations. Pour permettre la sélection des résultats produits, le paramètre *step* est mis à zéro et sa valeur correcte est placée au niveau du paramètre *stride*. Le paramètre *count* prend la valeur de R. Ainsi, les résultats intermédiaires produits par les activations pour la réception sont écrasés.

Contraintes sur le schéma de génération. Pour permettre la génération de code, notre schéma de fusion de processus de communication est contraint par plusieurs hypothèses. Ces hypothèses concernent la structure des nids de boucles des processus.

– Le processus d'entrée a N niveaux de boucle et le processus de sortie a $N-1$ niveaux avec $N \in [2, \infty[$. Ce décalage est dû à la contrainte posée par la mise en œuvre d'une

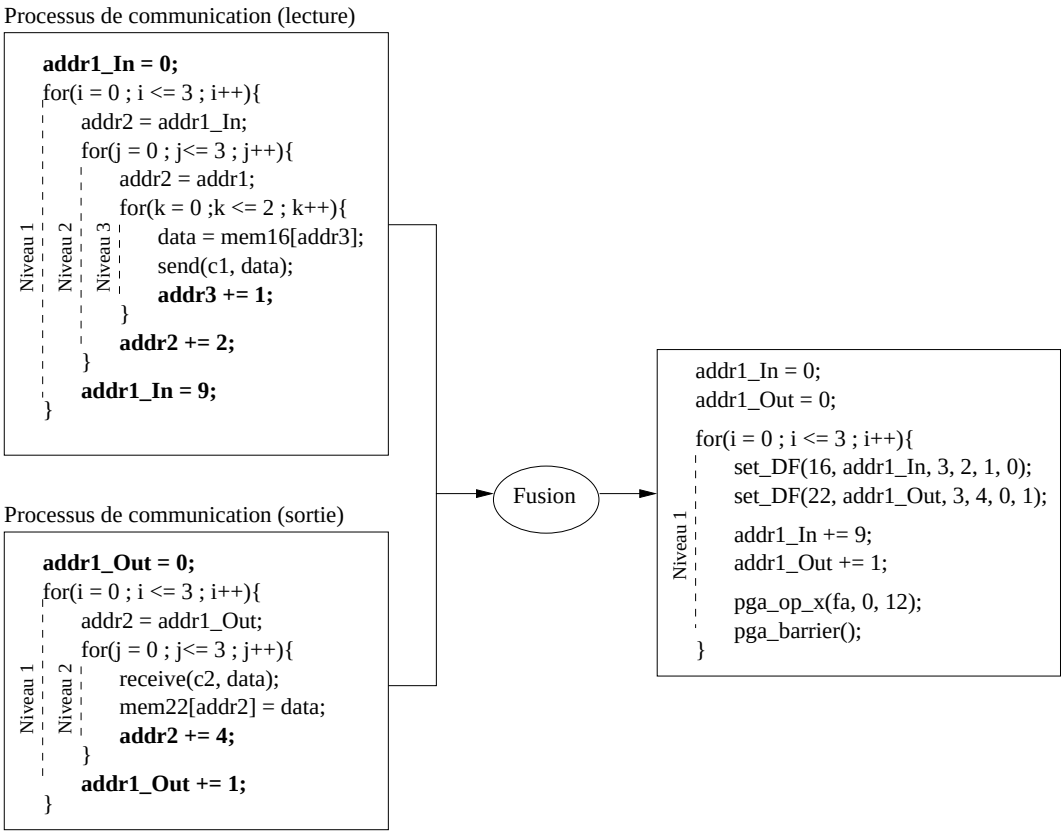


FIG. 5.12 – Fusion du comportement de deux processus de communication pour alimenter les entrées-sorties d'un processus de calcul.

seule dimension par le GA de sortie. Si $N = 1$, les deux processus ont chacun un seul niveau de boucle.
- Les boucles (des deux types de processus) non-absorbées dans les configurations des GA doivent avoir un nombre d'itérations identique.
- Dans le cas où il y aurait plusieurs processus d'entrée, ceux-ci doivent avoir la même structure de nid de boucle pour pouvoir être fusionnés.

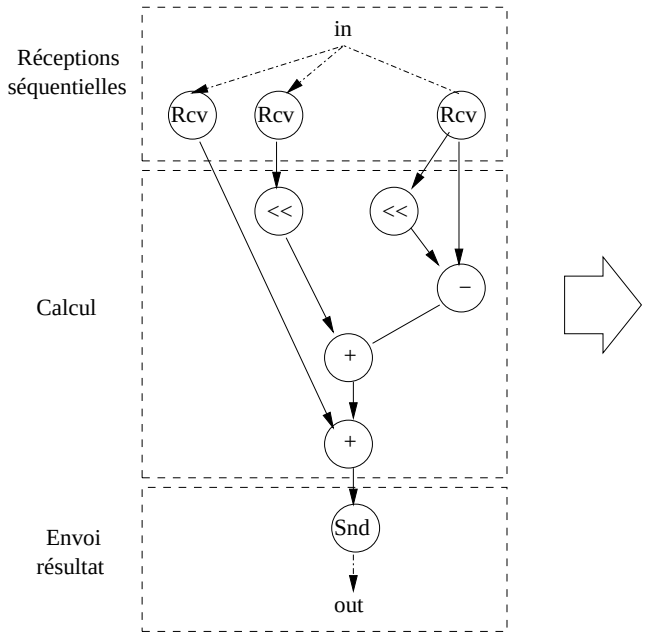
5.4.1.4 Mise en œuvre des processus de calcul

Les fonctions accélérées sont exécutées sur le PiCoGA. Le code Griffy-C est généré à partir du CDFG du processus de calcul. Les restrictions de Griffy-C concernent principalement les structures de contrôle de type boucle, les opération arithmétiques et une forme de code de type *assignation unique* (SSA).

La figure 5.13 donne le DFG et le code Griffy-C généré du corps d'un processus de calcul qui est un filtre *Finite Impulse Response* (FIR) défini par la formule :

$$y(n) = \sum_{i=0}^{N} h(i)x(n-i) = h(0)x(n) + h(1)x(n-1) + ... + h(N)x(n-N) \qquad (5.3)$$

Les variables x et y correspondent respectivement aux signaux d'entrée et de sortie. La variable b correspond aux coefficients du filtre. Le DFG de la figure 5.13 représente l'application du filtre à 3 échantillons pour les coefficients $\{1, 2, 7\}$. Il reçoit trois données depuis son processus de communication d'entrée puis produit un résultat envoyé à son processus de communication de sortie.




```
#pragma picoga processnode 1 1 out in (
    static int t1, t2, t3;
    int temp_3845,
        temp_2236,
        temp_2534,
        temp_3247,
        temp_4149;
    //Réceptions séquentielles
    t1 = t2;
    t2 = t3;
    t3 = in;

    //Calcul
    temp_3845 = t2 << 1;
    temp_2236 = t3 << 3;
    temp_2534 = temp_2236 − t3;
    temp_3247 = temp_2534 + temp_3845;
    temp_4149 = t1 + temp_3247;

    //Envoi résultat
    out = temp_4149;
}
#pragma end
```

FIG. 5.13 – Graphe flot de données du corps d'un processus de calcul (filtre FIR-3) et le code Griffy-C généré pour le PiCoGA.

Les données reçues séquentiellement sont maintenues dans des variables statiques qui conservent leurs valeurs entre les activations. La figure 5.14 illustre la réception de trois données par trois activations. Les valeurs des variables statiques sont propagées à chaque activation. Lorsque le nombre d'activations est égal au nombre de réceptions, les variables ont une valeur correcte.

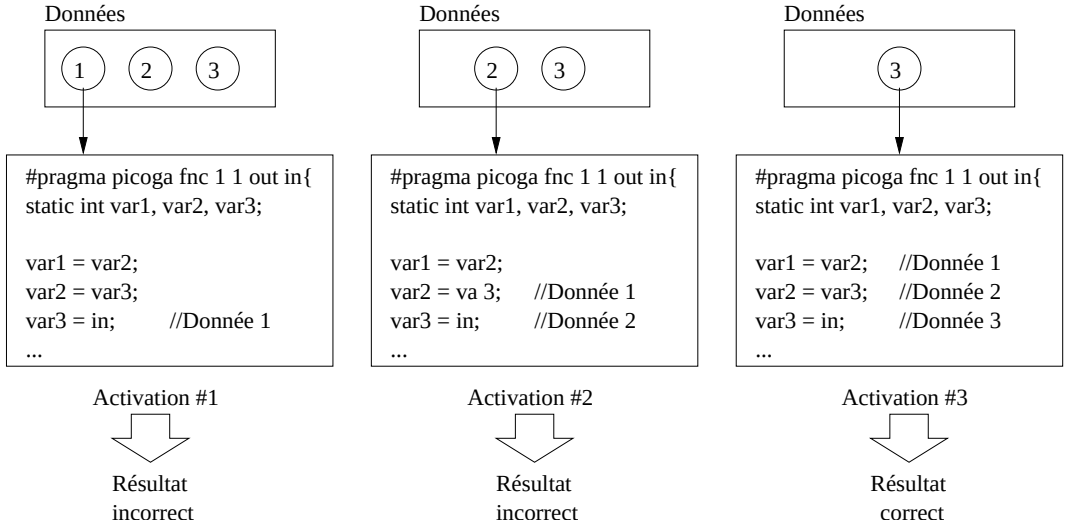


FIG. 5.14 – Réception séquentielle des données par le processus de calcul généré en Griffy-C.

5.4.2 Synthèse pour une architecture à gros-grain pipelinée

La section précédente a présenté le principe de réécriture d'un code Avel pour cibler un modèle d'exécution de type logiciel/matériel. Le code généré est pris en entrée d'une chaîne d'outils tiers dédiés à une architecture (dans notre cas d'étude le DREAM). Dans cette section nous nous intéressons au ciblage d'une matrice reconfigurable à gros-grain prototypée dans Madeo-BET. Cette matrice peut être intégrée dans un processeur reconfigurable tel que le DREAM (voir section 5.4.1) ou être indépendante.

Pour ce cas d'étude nous supposerons que les processus de communication seront pris en charge par des contrôleurs spécifiques indépendants de la matrice et similaires à ceux du DREAM. Par conséquent le principe de calcul des configurations pour les contrôleur est de même nature que celui détaillé en section 5.4.1.

5.4.2.1 Modèle d'exécution

Structure de la matrice. Nous prenons comme cas d'étude une architecture structurée en étage d'opérateurs à gros-grain pipelinés et que nous appellerons CGRA. Cette architecture a une structuration similaire aux matrice PiCoGA [98] et PipeRench [50]. La figure 5.15 détaille le principe de composition de l'architecture.

L'opérateur de calcul de CGRA est une UAL dont les opérations possibles sont définies dans la description Madeo-ADL. L'UAL prend en entrée deux opérandes qui proviennent soit d'un *Input Output Block* (IOB) ou de l'étage supérieur adjacent. Les entrées d'une UAL sont sélectionnées par deux multiplexeurs et sa sortie est connectée à un aiguilleur.

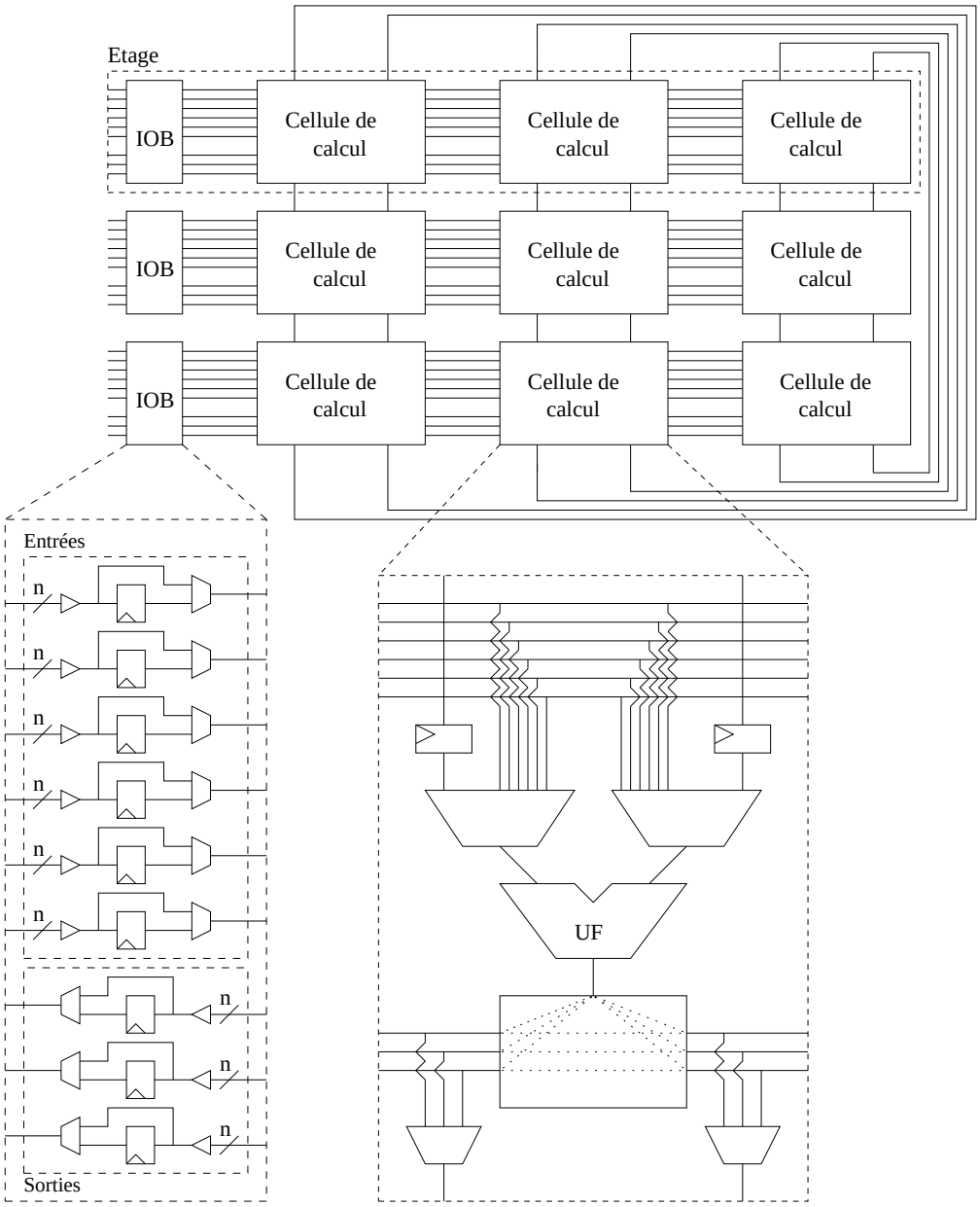


FIG. 5.15 – Structure de l'architecture CGRA. Les cellules de calcul sont aboutées pour former des étages. Les transferts de données entre étages sont tamponnés dans des registres pour obtenir un recouvrement temporel à l'exécution.

Un étage est formé d'un IOB et de plusieurs cellules de calcul aboutées. Un IOB est composé d'un bloc d'entrée et d'un bloc de sortie. Le routage des données vers les cellules de calcul d'un étage est contrôlé par des tampons à 3 états. Les transferts de données entre les étages sont tamponnés dans des registres pour obtenir un recouvrement temporel entre l'exécution de chaque étage.

Le dernier étage de la matrice est connecté au premier pour former une connexion de type circulaire. Ce type d'interconnexion est utilisée dans le cas d'une virtualisation des étages de calcul (voir ci-dessous).

Le modèle d'exécution de CGRA est spécialisé pour la mise en œuvre de DFG dont les nœuds sont directement mis en correspondance avec les UAL.

5.4.2.2 Compilation du graphe flot de données

L'application est un DFG qui correspond au corps d'un processus de calcul. Sa mise en œuvre ne nécessite pas d'étape de synthèse logique, les nœuds du DFG sont directement mis en correspondance avec les opérateurs de l'architecture. Cependant des transformations doivent être appliquées au graphe afin qu'il soit structurellement conforme pour l'étape de placement-routage. La figure 5.16 illustre le flot de transformation du DFG d'un processus de calcul.

La première étape ordonnance les nœuds du DFG (figure 5.16(a) et (b)). Contrairement à PiCoGA, CGRA ne gère pas la mémorisation des données entre plusieurs activations. Par conséquent, les trois données en entrée sont reçues sur trois canaux différents et donc en parallèle.

Les emplacements possibles d'un nœud dans le DFG sont définis par son élasticité [106]. L'élasticité d'un opérateur est bornée par l'intervalle $[e1, e2]$ avec $e1$ l'étage pour une exécution au plus tôt et $e2$ pour une exécution au plus tard. Ces deux bornes sont calculées par les ordonnancements *As Soon As Possible* (ASAP) (figure 5.16(a)) et *As Late As Possible* (ALAP) 5.16(b) du DFG. L'ordonnancement d'un opérateur dans cet intervalle garantit la non-violation des dépendances de donnée. Les opérateurs sont placés en fonction du nombre de ressources de calcul disponibles dans les étages.

Les données lues et produites par les opérateurs sont tamponnées d'étage en étage pour permettre un recouvrement temporel. Par conséquent, les transferts directs entre étages non-adjacents entraîneraient des violations de dépendance de donnée. La figure 5.16(a) met en évidence dans le DFG des dépendances de données qui traversent plusieurs étages. Dans une première étape, un ordonnancement ALAP permet de réduire le nombre de ces dépendances (figure 5.16(b)). En effet, l'exécution au plus tard d'un opérateur le rapproche des consommateurs de sa sortie. Cependant, l'ordonnancement ALAP n'est pas suffisant dans le cas de plusieurs consommateurs placés à différents étages. De plus, il ne s'applique que sous réserve de ressources disponibles aux étages concernés.

Pour permettre le transfert d'une donnée dans un étage, des opérations *identité* sont ajoutées au DFG pour connecter les étages (figure 5.16(c)). Cet opérateur effectue une identité entre sa première entrée et sa sortie. L'UAL met en œuvre cette opération par une connexion directe entre sa première entrée et sa sortie.

L'algorithme 2 détaille l'insertion des opérateurs *identité* dans les étages du DFG.

La fonction *insertionOpId* prend en paramètres les étages d'un DFG ordonnancé. Les étages sont traités de haut en bas pour l'insertion des opérateurs *identité*. Pour chaque étage,

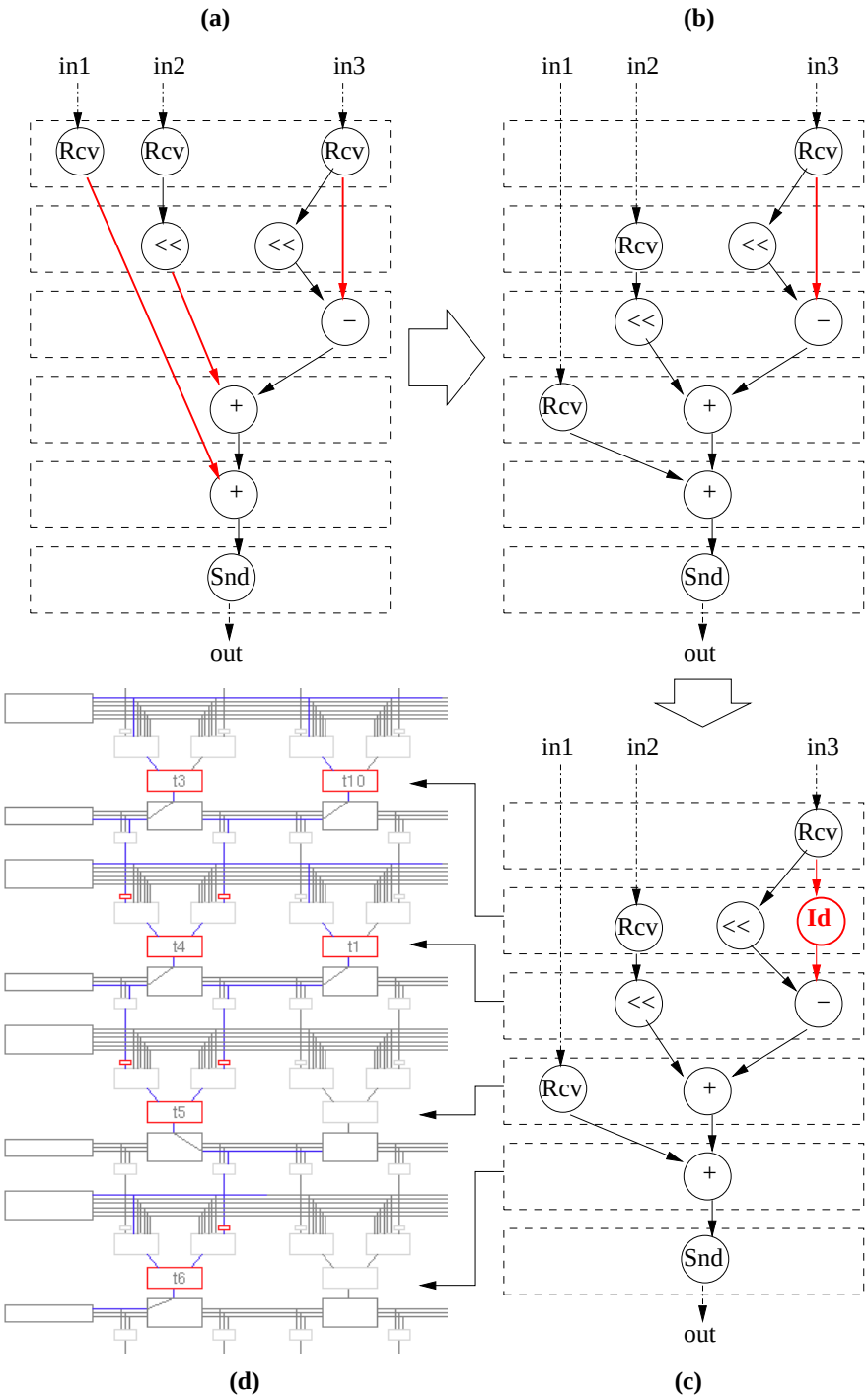


FIG. 5.16 – Transformation du DFG pour sa mise en œuvre sur la matrice CGRA.

Algorithme 2 Algorithme pour l'ajout des opérateurs *identité* dans les étages d'un DFG ordonnancé. Ces opérateurs interconnectent les étages pour le passage de données aux opérateurs placés dans différents étages.

```
1  Fonction insertionOpId( étages : Etages ordonnancés d'un DFG ) :
2  Variable
3      cheminEtages, étagesPrécedents : Pile
4      stop : Booléen
5      index : Entier
6  Début
7      étagesPrécédents empiler étages[0]
8      Supprimer étages[0] de étages
9      PourChaque étage De étages Faire
10         PourChaque opérateur De étage Faire
11             PourChaque entrée De opérateur Faire
12                 cheminEtages ←nouvelle Pile
13                 index ←0
14                 TantQue étagesPrécédents[index] ≠ source de entrée Faire
15                     empiler étagesPrécédents[index] dans cheminEtages
16                     index ← index + 1
17                 Si cheminEtages = ∅ Alors
18                     entréeCourante ← entrée
19                     PourChaque cheminEtage De cheminEtages Faire
20                         opId ← création opérateur identité
21                         Mise à jour entrée de opId avec entréeCourante
22                         Ajout de opId à cheminEtage
23                         entréeCourante ← sortie de opId
24                     Mise à jour de l'entrée de opCourant avec entréeCourante
25         empiler étage dans étagesPrécedents
26 Fin
```

la source de chaque entrée des opérateurs est recherchée dans les étages supérieurs. La pile *étagesPrécédents* est initialisée avec le premier étage qui est alimenté par les entrées du DFG. Si la source se trouve dans un étage adjacent alors il n'est pas nécessaire d'insérer un opérateur. Sinon chaque étage traversé est empilé dans *cheminEtages* jusqu'à ce que l'étage source soit atteint. L'étape suivante insère dans chaque étage de *cheminEtages* un opérateur *identité* et met à jour les connexions. Lorsqu'un étage a fini d'être traité, il est ajouté à la pile *étagesPrécédents* pour l'analyse de l'étage suivant.

Optimisation. Une utilisation directe de l'algorithme 2 peut entraîner une occupation sous-optimale des ressources de calcul et de routage. Pour chaque donnée à transférer entre les étages, des opérateurs *identité* sont alloués. L'allocation est également réalisée pour des données issues d'un même opérateur ce qui entraîne une redondance inutile. La figure 5.17(a) illustre une exemple d'occupation sous-optimale des ressources.

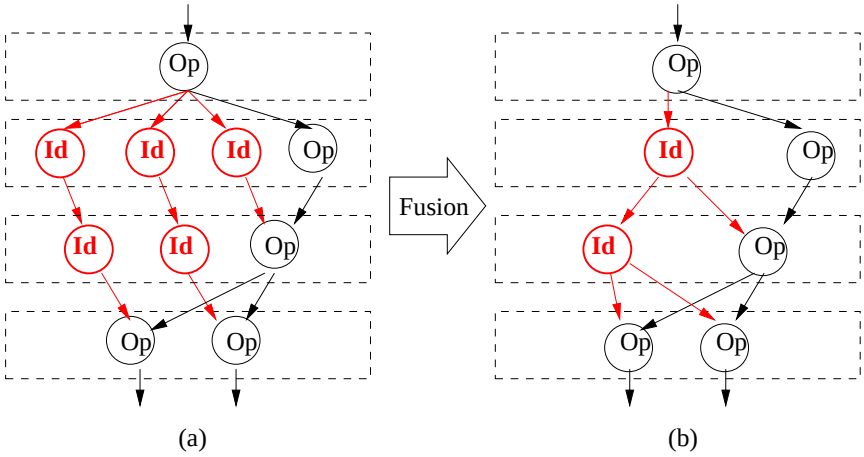


FIG. 5.17 – (a) Pour chaque donnée à transférer entre les étages un opérateur *identité* est alloué. (b) L'occupation des ressources par les opérateurs *identité* est optimisée par fusion des opérateurs identiques.

Le coût engendré est réductible par une mutualisation des cellules pour des données identiques. Lorsqu'une même donnée doit être transférée à plusieurs étages les opérateurs *identité* placés dans un même étage et dont les entrées sont identiques sont fusionnés. La figure 5.17(b) montre la fusion des opérateurs *identité*. Cette opération est réalisée sur le DFG produit par l'algorithme 2.

5.4.2.3 Placement-routage des opérateurs

Le placement-routage du DFG sur la matrice CGRA est réalisé par une réutilisation objet des outils de Madeo. Le placeur-routeur de Madeo est orienté vers les architectures à grain-fin. Pour le placement il utilise une métaheuristique de type *recuit simulé*. Le routeur est un routeur point-à-point avec la prise en compte des conflits de ressource. Ce dernier est directement utilisable pour l'architecture CGRA et ne requiert aucune modification. Cependant, l'algorithme de placement n'est pas adapté à la structure régulière en étages de CGRA.

Les algorithmes de placement-routage sont définis dans une classe, *MyPlacerRouter*, et utilisés à travers l'interface de la classe. Pour cibler CGRA, *MyPlacerRouter* est sous-classée pour redéfinir l'algorithme de placement. Cette méthode est typique d'une approche cadricielle qui prône l'extension d'un ensemble initial de classes pour une adaptation à des besoins spécifiques. Le concepteur gagne en productivité par la réutilisation de méthodes préexistantes. Ainsi, nous exploitons la méthodologie objet sur laquelle s'appuie Madeo pour répondre aux besoins de notre architecture.

5.4.3 Synthèse pour les architectures de type FPGA

Dans cette section nous considérons la mise en œuvre du réseau de processus sur une architecture de type FPGA. La partie basse cible soit une architecture prototypée avec Madeo-BET ou une architecture tierce. Dans le premier cas le binaire de configuration est produit par le flot décrit dans le chapitre 4. Dans le deuxième cas une *netlist* Verilog liée à la libraire de l'architecture est produite. La génération du binaire de configuration est effectuée par les outils liés à l'architecture tiers. Nous prenons comme cas d'étude le FPGA embarqué M2000 [3].

5.4.3.1 Flot global

Une architecture de type FPGA est constituée d'un réseau de cellules de base qui contiennent chacune comme élément de calcul une ou plusieurs mémoires (LUT). Le contenu d'une LUT est la table de vérité d'une équation logique. Ces tables de vérités sont le résultat de la synthèse logique des opérateurs du HL-CDFG par Madeo+. La figure 5.18 détail le flot de mise en œuvre.

La synthèse produit un CDFG qualifié de bas-niveau (*Low-Level CDFG* (LL-CDFG)) qui se situe à un niveau transfert de registre. Les opérateurs du HL-CDFG sont mis en œuvre sous la forme d'un réseau de tables de vérité au format BLIF produites par le synthétiseur SIS [128]. A ce stade le LL-CDFG prend en compte les spécificités de l'architecture ciblée (type de LUT, de bascule, etc.) et est utilisé comme format intermédiaire pour la mise en œuvre de l'application.

La spécialisation du LL-CDFG produit un réseau de cellules de base d'une architecture à grain-fin. La cellule considérée est celle du FPGA M2000 qui est volontairement simple pour cibler un cadre embarqué. Elle est composée d'une LUT-4 dont la sortie est soit combinatoire ou *latchée* par une bascule-D. Cette cellule est représentative des FPGA et peut être considérée comme un sous-ensemble d'architectures plus complexes (e.g Xiling Virtex).

La première étape de spécialisation est une visite (au sens objet) du LL-CDFG afin de produire l'ensemble des LUT du circuit. Cette étape calcule les mots de configuration pour chaque LUT et détermine leurs entrées/sorties. La seconde étape est la fusion des couples LUT-bascule pour produire les cellules de base. L'étape suivante est l'instanciation des signaux et enfin la génération de la *netlist* Verilog ou bien le placement routage sur une architecture Madeo-BET. Dans le cas d'une génération de *netlist* Verilog, les cellules de base sont mises en correspondance avec leurs instances de librairie.

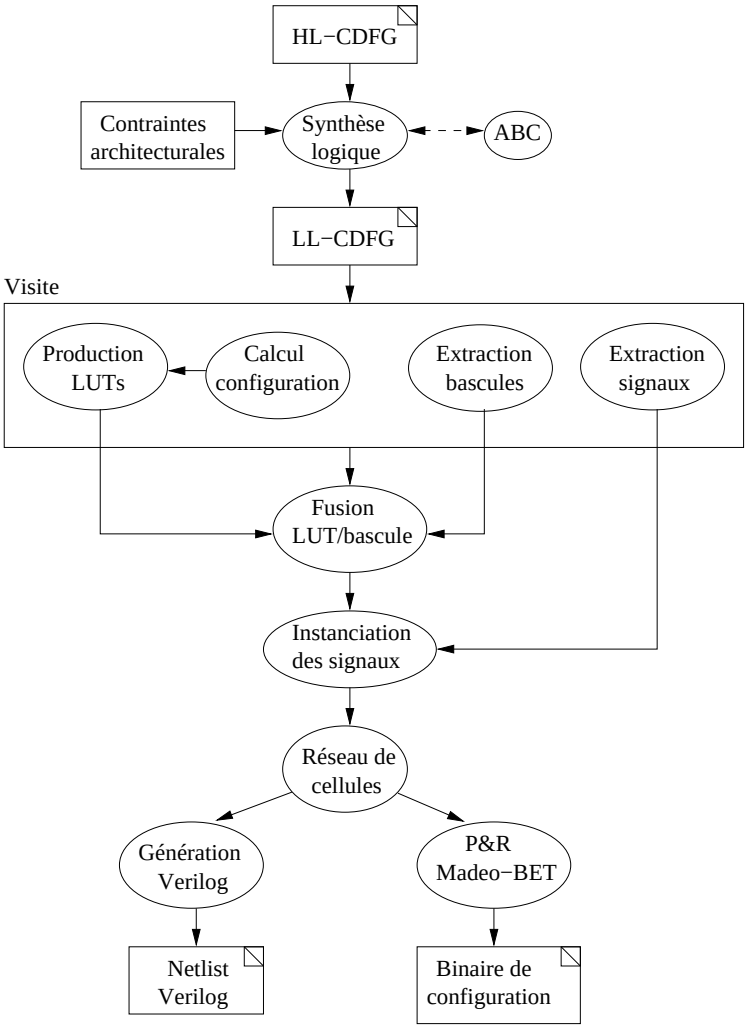


FIG. 5.18 – Flot Madeo+ pour la génération d'un réseau de cellules configurables à grain-fin.

5.4.3.2 Production du réseau de cellules de base

Les nœuds d'un LL-CDFG se divisent en plusieurs catégories : les constantes prisent en entrée d'opérateurs ; les *custom operators* qui correspondent à un groupement de plusieurs fonctions logiques mises en œuvre chacune par une LUT ; les bascules qui sont de plusieurs types ; les LUT avec un nombre différent d'entrées et spécifiées par leur équation logique e.g. `I1*I2`. Ces différents nœuds sont mis en œuvre par des cellules interconnectées lors de la phase d'instanciation des signaux.

Production des cellules. Les cellules de base du circuit sont modélisées dans l'outil par une classe *LUTCell* qui définit le nom de la LUT, son type d'instance dans la librairie M2000, ses entrées/sorties, ses ports optionnels qui sont `clock`, `enable`, `reset` et enfin son mot de configuration. Ainsi une *netlist* est modélisée par un réseau d'instance de la classe *LUTCell*. Chaque instance est spécialisée en fonction de ces différents paramètres qui déterminent sa correspondance dans la librairie M2000. Les paragraphes suivants détaillent la mise en œuvre des cellules en fonction des contraintes de l'architecture.

Les constantes. Les constantes sont mises en œuvre en fonction de leur valeur : 0 ou 1. La constante 0 est absorbée dans l'entrée de sa destination. La constante 1 n'est pas absorbée dans l'entrée et nécessite la création d'une LUT. Celle-ci a pour caractéristique d'avoir ses entrées bloquées à 0 et d'avoir le bit de poids faible (position 0) de son mot de configuration à la valeur 1.

Les custom operators. Un *custom operator* est un graphe de fonctions logiques encodées dans le format BLIF. La mise en œuvre de ce type d'opérateur consiste à produire une LUT par fonction BLIF. Les mots de configurations de chaque LUT son calculés par une évaluation complète de la table de vérité minimisée. Les connexions sont déterminées à partir des entrées/sorties des BLIF. Après une opération de *retiming* d'un *custom operators* des bascules sont insérées dans le graphe de fonctions. Ces bascules seront fusionnées, si possible, avec leur LUT connectée en entrée.

Les bascules. Dans certains cas, une bascule-D n'est pas fusionnable avec une LUT e.g. une chaîne de plusieurs bascules. Par conséquent, l'instanciation est réalisée par une cellule qui spécifie le type de la bascule-D et dont la LUT n'est pas utilisée. La connexion est effectuée uniquement avec la première entrée, les autres étant forcées à 0. La configuration de la LUT a pour valeur `0000000000000010` (encodage en *little endian*) qui réalise une identité entre sa première entrée et sa sortie.

Les LUT avec équation logique. Les LUT représentées dans le LL-CDFG sont caractérisées par leur nombre d'entrées et leur équation logique spécifiée textuellement. Dans le cas d'une LUT avec un nombre d'entrées inférieur à 4, les entrées non utilisées sont forcées à 0. Pour générer le mot de configuration d'une LUT il est nécessaire d'évaluer son équation logique. Pour cela un compilateur génère une représentation exécutable de l'équation (un bloc Smalltalk dans notre cas). Ensuite, l'équation est évaluée avec toutes les combinaisons possibles appliquées aux entrées pour produire le mot de configuration.

Instanciation des signaux L'instanciation des signaux est réalisée à partir d'une table des symboles qui associe à chaque entrée et sortie des LUT du circuit un signal. Ces signaux sont alloués lors de la visite du LL-CDFG. Le résultat de cette instanciation est un réseau

de cellules qui correspond à la *netlist* de l'application. Après optimisation ce réseau peut être généré en Verilog ou placé-routé sur une architecture Madeo-BET.

5.4.3.3 Techniques d'optimisation

Il existe de nombreux algorithmes d'optimisation du *mapping* technologique sur des LUT[1] pour les architectures de type grain-fin. Ils sont généralement fondés sur des heuristiques et adressent, séparément ou conjointement, la minimisation du nombre de LUT, la réduction des délais, l'amélioration de la routabilité et la réduction de la consommation énergétique [46, 34, 134, 72, 30].

Dans cette section nous détaillons des optimisations pour réduire le nombre de cellules nécessaires. La définition de nouveaux algorithmes sort du cadre de ces travaux et nous considérons ces optimisations comme des primitives de bases réutilisables.

Suppression des LUT NOP. En fonction du contexte de la synthèse (paramètres du circuit), certaines LUT peuvent devenir inutiles car effectuant des opérations nulles. Nous qualifions de LUT *No OPeration* (NOP) une LUT combinatoire qui effectue une opération d'identité entre une seule de ses entrés (les autres étant forcées à 0) et sa sortie. Par conséquent, ces LUT peuvent être supprimées et remplacées par une connexion sans affecter le bon fonctionnement du circuit. Les étapes de suppression sont détaillées par l'algorithme 3.

Algorithme 3 Algorithme pour la suppression des LUT NOP dans le réseau de cellules.

```
1  Fonction suppressionLUTNOP(cellules : Collection de LUTCell) :
2  Variable
3      colNOP : Collection de LUTCell
4      colDest : Collection de LUTCell
5  Début
6      PourChaque celluleCourante De cellules Faire
7          Si (configurationDe(celluleCourante) = '1010101010101010') et
   (entréesDe(celluleCourante)[1] ≠ '0') et (typeDe(celluleCourante) =
   'LUT combinatoire') et (nombreEntréesDe(celluleCourante) = 1) Alors
8              Ajouter celluleCourante à colNOP
9      PourChaque celluleNOP De colNOP Faire
10         colDest ←Récupération des destinations de la sortie de celluleNOP
11         PourChaque celluleDest De colDest Faire
12             Connexion de l'entrée de celluleNOP à l'entrée de colDest
13         Suppression de celluleNOP dans le réseau de cellules
14 Fin
```

Fusion des LUT et des bascules. L'étape de fusion consiste à former des couples LUT-bascule connectées entre elles. En fonction du type de la bascule, une cellule correspondante

[1]Dans le cas de Madeo+ cette étape est réalisée dans le premier étage du flot de la figure 5.18.

est instanciée. Elle a pour sortie celle de la bascule et pour entrées celles de la LUT. La fusion est impossible dans deux cas :

1. L'entrée de la bascule n'est pas connectée à la sortie d'une LUT, par exemple elle est connectée à une entrée du module top ou à une autre bascule.
2. La sortie d'une LUT est connectée à l'entrée d'une bascule mais également à d'autres cellules. Dans ce cas la sortie de la LUT ne peut pas être absorbée.

Dans ces deux cas la bascule est instanciée par une cellule individuelle.

Fusion de LUT à 2 entrées. Cette opération consiste à fusionner deux LUT à deux entrées qui forment un chemin combinatoire. Le résultat de cette optimisation est le remplacement des deux LUT par une seule LUT à trois entrées au comportement équivalent. Le principe de la fusion est illustré par la figure 5.19(a) et détaillé dans l'algorithme 4.

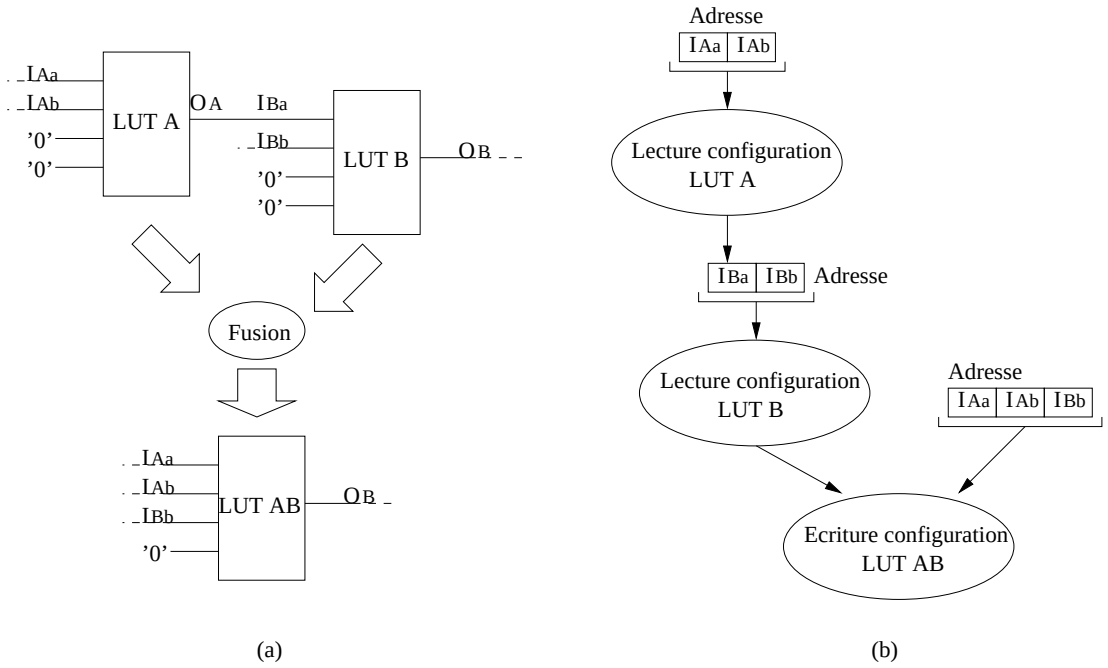


FIG. 5.19 – (a) Fusion de deux LUT-2 remplacées par une LUT-3. (b) Calcul de la configuration de la LUT-3 ayant un comportement équivalent aux deux LUT-2

La première étape de l'algorithme est la sélection des LUT-2 combinatoires. La seconde étape est de vérifier pour chaque LUT-2 que toutes les destinations sont compatibles pour une fusion. Les deux critères principaux sont que la LUT de destination ait deux entrées et qu'une de ses entrées soit connectée à la sortie de la LUT source. La fusion est impossible si la sortie de la LUT source commande les ports de chargement ou d'initialisation d'une bascule-D située dans la cellule de destination. Si une seule destination n'est pas compatible pour la fusion alors celle-ci n'est pas effectuée car la LUT source ne peut être supprimée sans affecter le bon fonctionnement du circuit.

Une fois les LUT-2 sélectionnées pour la fusion, il est nécessaire de recalculer une configuration pour chaque LUT-3 qui remplacera les couples (source, destination). Pour cela, les mots

de configuration des deux LUT sont fusionnés par la combinaison des deux entrées de la LUT source avec celle de la LUT de destination (voir figure 5.19(b)). L'algorithme revient à évaluer le couple de LUT pour toutes les entrées possibles. On obtient ainsi le mot de configuration de la LUT résultante de la fusion.

Algorithme 4 Algorithme pour la fusion des LUT-2.

```
1  Fonction suppressionLUTNOP(cellules : Collection de LUTCell) :
2  Variable
3      colSource, colDest : Collection de LUTCell
4      msbCol, lsbCol : Collection de mots binaires
5      nouveauMotConfig : Mot binaire
6      resSrc, resNouv : Bit
7  Début
8      PourChaque celluleCourante De cellules Faire
9          Si (typeDe(celluleCourante) = 'LUT combinatoire') et
   (nombreEntréesDe(celluleCourante) = 2) Alors
10             Ajouter celluleCourante à colSource
11     PourChaque celluleSource De colSource Faire
12         Ajout des destinations sortie celluleSource dans colDest
13         Si Connexion sortie celluleSource est compatible avec tout colDest
   Alors
14             PourChaque celluleDest De colDest Faire
15                 msbCol ← ('00' '00' '01' '01' '10' '10' '11' '11')
16                 lsbCol ← ('0' '1' '0' '1' '0' '1' '0' '1')
17                 nouveauMotConfig ← '0000000000000000'
18                 PourChaque msb, lsb De msbCol, lsbCol Faire
19                     resSrc ← configuration celluleSource[msb]
20                     resNouv ← configuration celluleDest[concat(resSrc, lsb)]
21                     nouveauMotConfig[concat(msb, lsb)] ← resNouv
22                     Connexion entrées celluleSource avec entrées celluleDest
23                     configuration celluleDest ← nouveauMotConfig
24                 Suppression de celluleSource dans le réseau de cellules
25 Fin
```

5.4.3.4 Evaluations des optimisations

Les évaluations se fondent sur les résultats donnés par Madeo+ et les outils M2000 pour la génération de netlists Verilog. L'outil M2000 prend en entrée une *netlist* au format EDIF et produit un équivalent Verilog prêt pour la synthèse. Madeo+ produit directement la *netlist* Verilog à partir du LL-CDFG. Nous prenons comme exemple de référence la synthèse d'un

incrémenteur.

Par défaut il n'est pas possible de contrôler les optimisations appliquées par l'outil M2000, en revanche les optimisations de Madeo+ peuvent être activées ou désactivées individuellement. De ce fait chaque optimisation de Madeo+ peut être comparée au meilleur résultat de l'outil M2000. L'outil M2000 ne donne pas d'information sur la nature des transformations appliquées à la *netlist* mais une transformation évidente est la fusion des LUT et des bascules.

Les résultats en nombre de *mfc* (*multi-function cell* qui est la cellule de base du M2000) allouées pour la synthèse d'un incrémenteur 32-bits par les outils M2000 et Madeo+ sont données dans le tableau 5.1. Les optimisations sont évaluées séparément et combinées pour obtenir le résultat final.

Outils	Sans opti.	Suppr. NOP	Merge LUT/Latch	Merge LUT2	Toutes opti.
M2000	-	-	-	-	394
Madeo+	564	551	408	453	354

TAB. 5.1 – Résultats de synthèse des outils M2000 et Madeo+ pour un incrémenteur 32-bits.

La figure 5.20 donne le surplus et les gains obtenus par le synthétiseur Madeo+ pour l'incrémenteur. Les résultats sont normalisés par rapport au nombre de cellules *mfc* allouées par l'outil M2000 (394 *mfc*). Les résultats positifs indiquent le nombre de cellules additionnelles et les résultats négatifs indiquent le gain par rapport à la synthèse M2000.

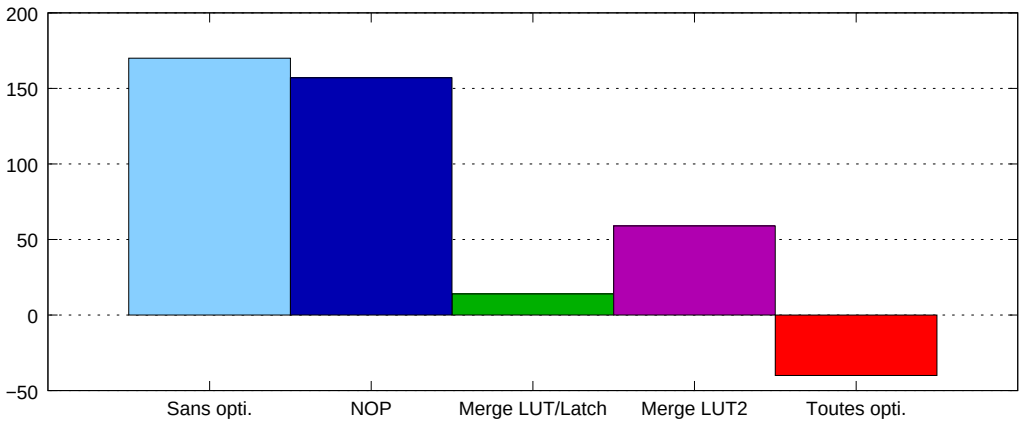


FIG. 5.20 – Résultats normalisés donnant le gain en nombre de LUT par rapport à l'incrémenteur.

La figure 5.20 montre l'impact de chaque optimisation sur la *netlist*. On constate que sans l'application d'optimisations nous obtenons un surplus de 170 *mfc* par rapport à l'incrémenteur. Cela est principalement dû à l'instanciation séparée des LUT et des bascules ce qui a pour conséquence d'augmenter le nombre de ressources utilisées. La fusion des LUT et des bascules permet de réduire jusqu'à 92% le nombre de *mfc* supplémentaires du cas sans optimisations et donne un surplus de 3,5% par rapport au résultat de référence (394 *mfc*). Les suppressions de LUT NOP ne concernent qu'un nombre restreint de LUT ce qui limite les gains. En revanche la fusion des LUT-2 permet de réduire jusqu'à 65% le nombre de *mfc* du cas sans optimisations avec un surplus de 15%. Enfin la combinaison de toutes les optimisa-

tions donne un nombres de *mfc* allouées inférieur à l'outil M2000. Le gain total est de 40 *mfc* ce qui représente une amélioration de 10%.

5.5 Conclusion

Pour explorer les performances des prototypes générés par DRAGE le concepteur doit pouvoir programmer rapidement les unités. Le niveau d'abstraction considéré est décisif pour favoriser les variations et la réutilisation des développements.

Ce chapitre a introduit un flot de programmation haut-niveau pour les unités reconfigurables d'un SoC. Le modèle d'exécution considéré est un accélérateur reconfigurable alimenté en données par des mémoires locales.

L'application est spécifiée dans le langage de composition Avel sous forme d'un réseau de processus communicants fondé sur le modèle de calcul CSP [66]. Une partie des processus est dédiée aux communications et l'autre aux calculs sur les données. Les processus de communication sont assimilables à des générateurs d'adresses et accèdent aux mémoires locales de l'unité. Les processus de calcul sont spécialisés pour le flot de données et sont spécifiés dans un sous-ensemble de C. L'ensemble des processus sont interconnectés et composés hiérarchiquement pour former le réseau.

La description complète est prise en entrée du synthétiseur Madeo+ pour cibler des architectures de différentes natures (granularité, couplage, etc.), préexistantes ou prototypées avec Madeo/DRAGE. Ainsi, trois architectures sont considérées pour montrer la portabilité d'une description Avel : processeur reconfigurable, unité à grain-fin et à gros-grain.

Ce chapitre et le précédent ont détaillé un flot outillé d'exploration rapide (prototypage et programmation) centré sur les unités reconfigurables. Cependant, ces unités ont pour vocation à être embarquées dans un SoC ce qui implique la prise en compte de l'impact de leur intégration système sur l'application. Pour obtenir une vision globale de l'exécution, il est impératif d'offrir au concepteur la capacité de visualiser les activités externes à l'unité (gestion de mémoires locales, transferts de données, etc.). Pour cela, le chapitre 6 aborde la modélisation système d'un SoC et une méthodologie de validation fonctionnelle des applications exécutées sur les unités reconfigurables.

Chapitre 6

Validation fonctionnelle multi-niveaux d'applications sur RSoC

Les chapitres 4 et 5 ont détaillé le flot d'exploration (prototypage et programmation) d'unités reconfigurables DRAGE. Les unités prototypées ont pour vocation à être embarquées dans un SoC ce qui implique la prise en compte de l'impact de leur intégration système sur l'application. Pour obtenir une vision globale de l'exécution, il est impératif d'offrir au concepteur la capacité de visualiser les activités externes à l'unité (gestion de mémoires locales, transferts de données, etc.). Ainsi, la validation fonctionnelle des applications est réalisée à un niveau local et également système.

Ce chapitre présente une méthodologie de validation fonctionnelle d'une application exécutée sur les unités reconfigurables d'un RSoC. Elle est fondée sur l'utilisation d'un flot d'outils pour la modélisation, la simulation et le débogage de l'application. Il s'appuie sur un langage dynamique orienté objet et sur des cycles de validation itératifs courts.

6.1 Introduction

6.1.1 Modélisation et simulation

La validation fonctionnelle d'une application exécutée sur un SoC est réalisée par la simulation d'un modèle de la plate-forme et de l'application. Cette dernière est partitionnée entre les composants du système. Cela implique une décomposition en plusieurs activités concurrentes qui incluent les transferts mémoire, l'activation des accélérateurs, le contrôle du processeur, etc. Par exemple, l'exécution d'une application sur une unité reconfigurable est dépendante de l'ordonnancement des transferts de données depuis la mémoire principale vers ses mémoires locales. L'impact de ces activités sur les performances et sur les résultats attendus doit être pris en compte dans la simulation. Ainsi, l'hétérogénéité de la plate-forme et sa nature concurrente complexifient la tâche de validation.

Pour réduire le niveau de complexité, les concepteurs modélisent la plate-forme et l'application à un haut-niveau d'abstraction [14]. Les langages VHDL [68] et Verilog [135] sont les plus répandus et autorisent des descriptions à un niveau fonctionnel. Cependant ils restent orientés vers la description matérielle et ne sont pas adaptés pour des modélisations rapides

à très haut-niveau. Une solution est donnée par l'extension de langages de programmation tels que C/C++ [69, 47] ou encore Java [63]. Le langage SystemC est le plus répandu pour la modélisation et la simulation de systèmes. Une description est réalisée par l'utilisation d'un cadriciel de classes C++ [55]. La particularité de SystemC est qu'il apporte les avantages de la conception objet pour le développement de systèmes matériels. Cependant, l'utilisation de C/C++ impose des limitations en comparaison des langages objet dynamiques disponibles dans le monde logiciel. Ces langages fournissent une plus grande capacité d'exploration de l'application à l'exécution avec le remplacement dynamique de code ou la modification de l'état des objets. De plus, malgré une orientation logicielle de l'approche SystemC, l'utilisation de méthodes de génie logiciel, qui favorisent la productivité, reste peu répandue.

Les sections 6.2, 6.3 et 6.4 présentent un cadriciel de modélisation et de simulation multi-niveaux d'une application exécutée sur un RSoC. Il s'appuie sur le langage dynamique Smalltalk-80 [49] et sur l'environnement Cincom VisualWorks [141].

6.1.2 Débogage d'une application

La simulation d'une application permet la mise en place de tests pour valider les résultats produits. Lorsque les tests échouent il est nécessaire de déboguer l'application. L'activité de débogage consiste à rechercher la cause d'une défaillance. L'efficacité de la recherche est conditionnée par les degrés d'observabilité et de contrôlabilité de la méthode utilisée. L'observabilité correspond à la capacité d'analyser l'état de l'application (e.g. valeurs des variables) tout au long de l'exécution. La contrôlabilité caractérise la possibilité de contrôler le flot d'exécution (e.g. arrêt de l'exécution)

Dans le domaine de la conception du logiciel, le débogage bénéficie d'outils sans réel équivalence dans celui de la conception de circuits. Par exemple, l'analyse de la pile de contextes d'exécution, le retour en arrière dans le flot d'exécution ou le remplacement dynamique de portions de code. La simulation logicielle d'un circuit (e.g. dans ModelSim) fournit un degré élevé d'observabilité et de contrôlabilité. Cependant, dans le cas de système complexes, les temps de simulation augmentent très fortement. La validation nécessitant plusieurs itérations à travers des cycles hypothèse-simulation-analyse, cette solution est incompatible avec une forte productivité.

La performance peut être obtenue par une exécution de l'application directement dans le matériel. L'analyse est réalisée par l'utilisation d'analyseurs logiques embarqués [152, 7], par une instrumentation du *bitstream* [52, 136] ou par les capacités de relecture de certains FPGA commerciaux [151, 100, 119]. Cependant, ces solutions ne sont pas orientées vers la contrôlabilité et restreignent le débogage à un niveau d'abstraction faible.

La section 6.5 présente une méthodologie de débogage qui combine les avantages de la simulation logicielle à l'exécution dans le matériel. Pour cela, des sondes sont insérées via une interface graphique dans la description comportementale de l'application (en AvelC) qui sera ensuite synthétisée. L'exécution est contrôlée par un contrôleur de débogage inclus dans la *netlist* et les signaux tracés sont visualisés par une reconstitution des variables de l'application. L'objectif est d'obtenir un degré élevé d'observabilité et de contrôlabilité tout en gardant une simulation rapide.

6.1.3 Méthodologie et outils pour la validation

Le flot de validation fonctionnelle est détaillé par la figure 6.1.

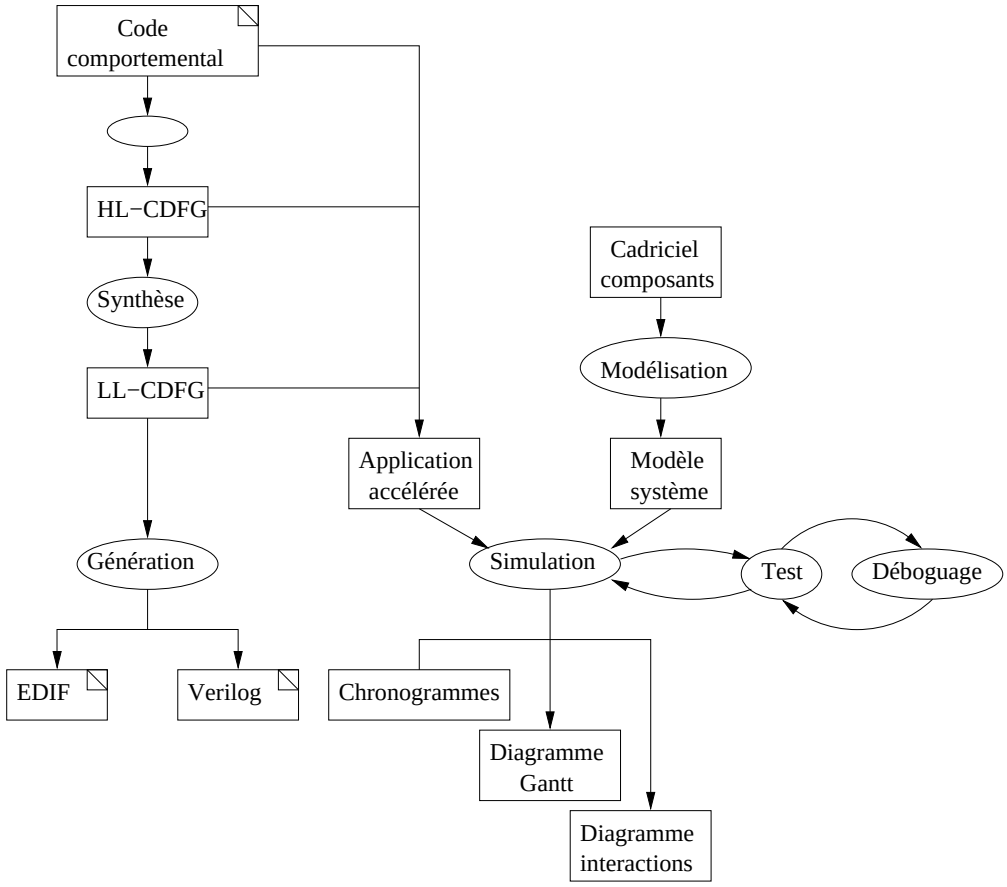


FIG. 6.1 – Flot de validation par la simulation multi-niveaux d'une application [121].

La plate-forme d'exécution est modélisée à un niveau système par l'assemblage de composants issus d'un cadriciel objet. L'application placée sur une unité reconfigurable est spécifiée à différents niveaux d'abstraction et intégrée dans un composant du modèle système. L'ensemble du cadriciel de modélisation est défini dans le langage dynamique Smalltalk-80.

La simulation multi-niveaux prend en entrée la spécification de l'application à accélérer et le modèle de la plate-forme d'exécution. Au plus haut niveau d'abstraction l'application est spécifiée par du code comportemental. La simulation à ce niveau est effectuée par une exécution logicielle de l'application. Un second point d'entrée est la représentation intermédiaire (HL-CDFG) de l'application ce qui permet d'élargir la spécification à d'autres langages. Cependant, l'utilisation d'un autre langage nécessite soit une réécriture vers celui du cicatriciel ou la disponibilité d'un générateur de CDFG pour le langage choisi.

Le HL-CDFG est synthétisé par Madeo+ pour produire un LL-CDFG qui représente l'application à un niveau RTL. Le LL-CDFG est utilisé pour la simulation ou bien pour la production de *netlists* (voir section 5.4).

Le simulateur produit des diagrammes de Gantt et des diagrammes d'interactions pour visualiser les activités au niveau système. Dans le cas d'une simulation de l'application à un niveau RTL, des chronogrammes tracent les valeurs des signaux.

Ces résultats sont pris en entrée d'un flot itératif de validation à travers des cycles courts incrémentaux de test et de débogage. Cette méthodologie de validation correspond à l'application aux SoC des concepts *agiles* issus du génie logiciel. Notre approche s'inspire principalement du cycle de développement *eXtreme Programming* (XP) orienté vers la flexibilité et l'efficacité des développements [156].

6.2 Modélisation système orientée objet

Cette section détaille le cadriciel objet *SmallSystem* pour la modélisation de systèmes logiciels et matériels.

6.2.1 Cadriciel de composants pour la modélisation

SmallSystem est un cadriciel orienté objet qui définit un ensemble d'entités abstraites pour la modélisation de systèmes concurrents simulables. Les modèles sont structurés par des composants interconnectés par des ports de communication. Les concepts mis en œuvre dans *SmallSystem* sont indépendants d'un langage particulier et peuvent être reciblés vers d'autres environnements. Cependant, notre approche bénéficie des caractéristiques inhérentes de l'environnement Cincom VisualWorks [141], à savoir un ensemble d'outils qui supportent la réflexivité (i.e. une capacité d'introspection avec l'accès aux états internes du programme exécuté et une capacité d'intercession avec la possibilité de modifier l'état d'exécution) [45], le typage dynamique, la gestion automatique de la mémoire objet (*garbage collector*) et un respect strict des règles de conception objet. Il intègre un débogueur symbolique tirant profit de la réflexivité du langage et permet l'exploration du contexte d'exécution, la modification du code et de son flot d'exécution dynamiquement.

Notion de cadriciel. La notion de cadriciel orienté objet peut être définie comme un ensemble de classes abstraites conçues pour répondre à une famille de problèmes [73]. Les avantages principaux d'un cadriciel en comparaison d'une librairie conventionnelle réside dans sa

capacité d'extensibilité et la granularité de réutilisation. Ces deux caractéristiques sont supportées par trois caractéristiques majeures de la programmation objet qui sont : l'abstraction, le polymorphisme et l'héritage.

Pour chaque élément majeur du cadriciel est définie une classe abstraite ainsi que ses interactions possibles avec les autres éléments. Ces classes sont réutilisables et spécialisables pour adresser un problème particulier. La spécialisation est réalisée par sous-classement de la hiérarchie de classes abstraites du cadriciel. De ce fait, le ciblage d'un nouveau domaine applicatif apparaît comme une extension par héritage de la hiérarchie de classes. De ce fait, la réutilisation se situe au niveau d'un ensemble de classes et non pas au niveau d'une classe individuelle.

Principes de modélisation SmallSystem. SmallSystem met à disposition des composants et des liens de communication abstraits. Un composant abstrait est défini par un comportement et une interface de communication qui contient les ports d'entrées-sorties.

Le comportement d'un composant est défini par un ensemble d'activités mises en œuvre par des méthodes d'instance. Ces activités internes sont des processus ordonnancés lors de l'initialisation du composant. Les communications entre processus sont réalisées de plusieurs manières, soit par des variables d'instance partagées, soit par des canaux de communications point-à-point utilisés également pour connecter les composants entre eux. Dans le premier cas le concepteur doit assurer la cohérence des accès par l'utilisation de sémaphores. Dans le deuxième cas les accès concurrents sont automatiquement gérés par la sémantique du canal.

Les ports sont interconnectés par des canaux de communication dont les sémantiques et les capacités sont définies par le concepteur. Conformément à la conception objet, un composant encapsule son comportement et son état. Autrement dit, un changement d'état provoqué par un autre composant n'est possible qu'à travers un envoi sur un port.

L'intégration d'un composant dans un système ne requiert que la connaissance de sa fonctionnalité et de son interface. Cette approche IP permet de définir des ensembles de composants indépendants d'un cadre spécifique de modélisation. Les composants sont hiérarchiques et peuvent définir un système complet réutilisable comme élément atomique à travers l'interface du niveau hiérarchique englobant.

La figure illustre un exemple de modèle hiérarchique. La hiérarchie englobante *Main* encapsule deux composants *Unit1* et *Unit2*. Le comportement de *Unit1* est défini par trois processus. Son interface est composée de trois ports avec deux processus qui communiquent avec l'extérieur. Les communications internes entre les processus sont assurées par des déclarations locales de canaux de communication.

6.2.2 Structure du modèle système

Hiérarchie de classes. La figure 6.3 décrit la hiérarchie de classe du cadriciel *SmallSystem*. Un modèle est créé par le sous-classement de deux classes principales qui sont : *Component* et *Connection*. Les classes *User Class (#1 et #n)* correspondent à une extension utilisateur pour cibler un domaine particulier.

Les modèles décrits avec *SmallSystem* sont orientés vers la simulation. Les composants et les connexions interagissent avec un simulateur à événements discrets à travers les méthodes héritées de la classe *SimulationObject*. Le simulateur est défini par la classe *Simulator* qui est le point d'entrée de la simulation d'un modèle. Cette classe est sous-classée par *SmallSystem*

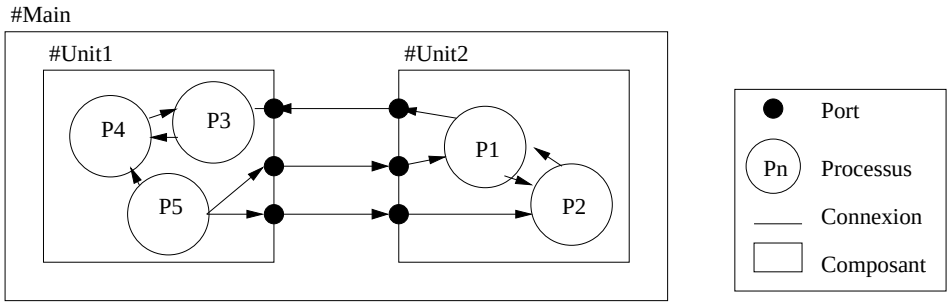


FIG. 6.2 – Exemple d'un modèle à trois composants. La hiérarchie englobante est le composant *Main* qui encapsule deux sous-composants *Unit #1* et *Unit #2*. Les activités internes de chaque sous-composant communiquent à travers les ports d'entrées-sorties. En revanche, dans un même composant les activités communiquent par le biais de canaux.

pour fournir une interface de création de journal. L'utilisateur peut définir son propre point d'entrée pour effectuer des initialisations spécifiques. Le simulateur est détaillé plus amplement dans la section 6.3.

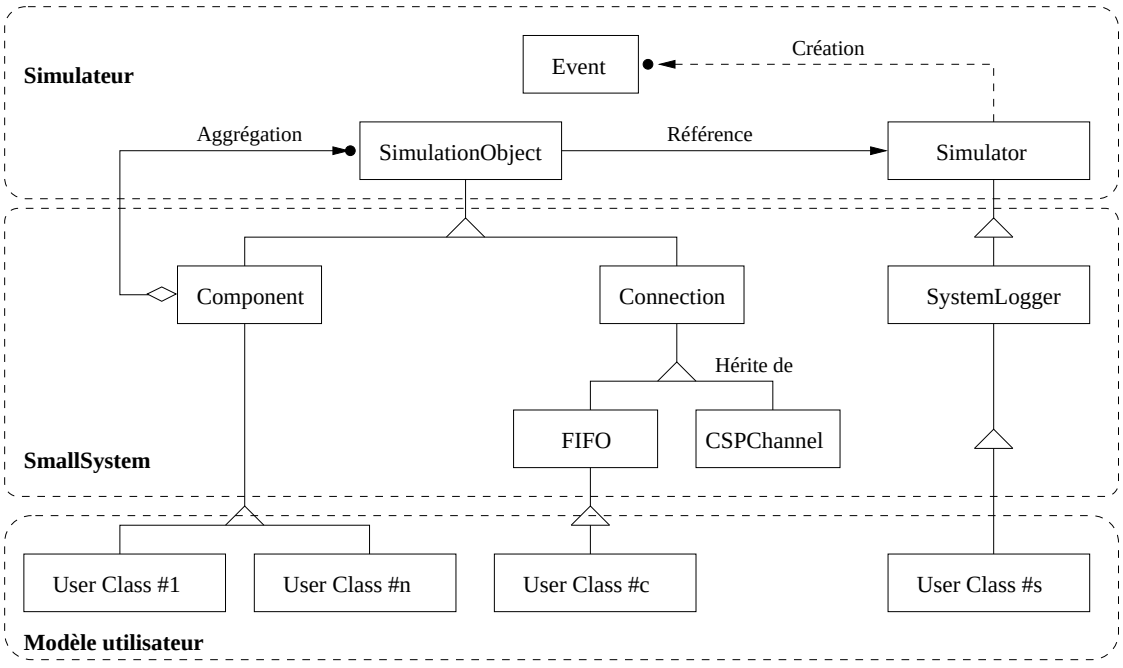


FIG. 6.3 – Hiérarchie de classes du cadriciel *SmallSystem*. Les fonctionnalités de simulation sont rendues disponibles par la relation d'héritage entre SmallSystem et le simulateur.

Composant. Un composant est l'unité de structuration d'un modèle décrit avec *SmallSystem*. A un niveau abstrait il définit l'ensemble des variables d'instance et des méthodes pour la création du composant. Les méthodes suivantes sont héritées et décrivent les caractéristiques principales d'un modèle :

defineComponents. Cette méthode est utilisée par les composants de type hiérarchique afin de déclarer leurs sous-composants. Pour un niveau hiérarchique N, les sous-composants de la hiérarchie inférieure directe $N-1$ seront déclarés. Un composant est déclaré par le message *addComponent : classeComposant named : nomComposant* qui prend en paramètres la classe à instancier et le nom associé au composant. Ce nom sera ensuite utilisé pour accéder a ces ports ou lors d'une exploration du modèle.

definePorts. Cette méthode est utilisée pour définir l'ensemble des entrées-sorties du composant. Les messages *defineOutputPort : nomPort type : typeConnexion* et *defineInputPort : nomPort type : typeConnexion* prennent en paramètre un identifiant et le type de connexion associée au port. En fonction de sa direction le nom du port est ajouté aux dictionnaires *inputs* ou *outputs* pour être utilisé lors de la connexion de deux ports.

defineConnections. Un composant hiérarchique déclare ses sous-composants et définis également leurs interconnexions. L'ensemble des connexions entre les ports des sous-composants de sa hiérarchie $N-1$ sont définis dans cette méthode. Le message *connect : nomComposant1 atOutputPort : nomPort to : nomComposant 2 atInputPort : nomPort* prend en paramètre le nom des deux composants à connecter et les noms des ports d'entrée et de sortie. La connexion est réalisée uniquement si les types des ports sont compatibles (direction et type de connexion).

defineChannels. Les activités internes d'un composant communiquent via des canaux de communications. Ces canaux sont locaux à un composant et sont définis dans cette méthode par le message *defineChannel : nomCanal type : typeCanal*. Ce message prend en paramètre le nom du canal qui permet aux activités d'y accéder et son type. Les connexions point-à-point sont garanties par l'envoi du message *allocate* au canal par les processus qui l'utilisent. Ce message incrémente un compteur lié au canal qui ne peut dépasser la valeur 2.

defineProcess. Les activités d'un composant correspondent à des processus ordonnancés lors de son initialisation. L'ordonnancement est réalisé par l'intermédiaire du simulateur qui permet de déterminer l'activation d'une activité à une date donnée.

Connection. A un niveau abstrait une connexion est définie par trois variables d'instances :

- **Source et destination** : la source et la destination de la connexion sont définies par deux éléments qui sont le nom du composant et le nom du port. Ces deux variables d'instances sont utilisées dans le cas d'une connexion directionnelle entre deux composants.
- **Content** : cette variable correspond au contenu du canal de communication qui établit la connexion. Le type du contenu varie en fonction de la nature du canal de communication. Par exemple, dans le cas d'une FIFO, elle est une collection d'objets et sa taille est contrainte par l'utilisateur. Smalltalk étant non typé le type d'objet assigné à *content* n'est pas contraint. Cependant, les contraintes sur les données peuvent être raffinées dans les sous-classes par la mise en place de tests.

6.2.3 Ecritures et lectures dans les canaux de communication

Une connexion abstraite ne définit pas de méthodes d'accès en lecture et écriture. Ces méthodes sont dépendantes de la sémantique du canal et sont redéfinies lors du sous-classement

de *Connection*. Deux types de canaux sont fournis par *SmallSystem* qui sont les FIFO et les canaux bloquants CSP.

Une FIFO est caractérisée par sa taille et par deux sémaphores qui contrôlent les opérations de lectures et d'écritures. Ces deux sémaphores contiennent un nombre de jetons d'autorisation d'accès. Le sémaphore d'écriture est initialisé avec un nombre de jetons égal à la taille de la FIFO. Le sémaphore de lecture est progressivement incrémenté après écriture des données. Par défaut, une lecture est bloquante si aucune donnée n'est disponible et une écriture est bloquante si la FIFO est pleine sinon elle est asynchrone. Les opérations de lecture/écriture sur une FIFO sont décrites par les algorithmes 5 et 6.

Algorithme 5 Algorithme pour la lecture sur une FIFO. L'opération est bloquante si la FIFO est vide. Le nombre de jeton dans le sémaphore détermine le nombre de données disponibles.

```
1 Fonction lectureFIFO( ) : Paquet de donnée
2 Variable
3     donnée : Paquet de donnée
4 Début
5     Attente sur le sémaphore de lecture
6     donnée ←lecture du premier élément du tableau content
7     Ajout d'un jeton dans le sémaphore d'écriture
8     Retourner donnée
9 Fin
```

Algorithme 6 Algorithme pour l'écriture asynchrone sur une FIFO. L'opération est bloquante si la FIFO est pleine.

```
1 Fonction écritureFIFO(donnée : Paquet de données ) :
2 Variable
3     donnée : Paquet de données
4 Début
5     Attente sur le sémaphore d'écriture
6     Ajouter donnée comme dernier élément du tableau content
7     Ajouter un jeton au sémaphore de lecture
8 Fin
```

Les canaux CSP ont une sémantique bloquante de type « rendez-vous » conformément au modèle de calcul défini par Hoare [66]. Une lecture (resp. écriture) est bloquante jusqu'à ce que la donnée à lire (resp. écrite) soit disponible (resp. lue). Contrairement aux FIFO le canal de communication ne contient qu'une seule donnée à la fois. Un canal CSP est une spécialisation d'une FIFO de taille 1. L'algorithme de lecture est identique à l'algorithme 5 en revanche contrairement à la FIFO l'écriture est synchrone et doit attendre l'opération de lecture (voir algorithme 7).

Algorithme 7 Algorithme pour l'écriture synchrone dans un canal de type CSP.

```
1 Fonction écritureFIFO(donnée : Paquet de données) :
2 Variable
3     donnée : Paquet de données
4 Début
5     Ajouter donnée comme dernier élément du tableau content
6     Ajouter un jeton au sémaphore de lecture
7     Attente sur le sémaphore d'écriture
8 Fin
```

6.3 Simulation système et multi-niveaux

Une simulation à de multiples niveaux d'abstraction gère la cohabitation de plusieurs modèles de représentation de l'application. A chaque représentation est associé un modèle de simulation qui interagit avec la plate-forme d'exécution modélisée dans *SmallSystem*. Cette section détaille les modèles de simulation utilisés à un niveau système et à un niveau RTL pour une application accélérée.

6.3.1 Définition des niveaux de simulation

Un modèle de simulation définit dans le temps l'évolution des états d'un système simulé. Le modèle utilisé pour la simulation d'un système modélisé avec SmallSystem est à événements discrets et dirigé par les événements, tandis que la simulation RTL d'une application accélérée est effectuée par un simulateur de type cycle près et bit près (*Cycle Accurate and Bit Accurate* (CABA)). Ce dernier est un simulateur à événements discrets dirigé par le temps. Il a été développé par L. Lagadec et intégré à notre cadriciel SmallSystem.

Dans un simulateur à événements discrets, un événement correspond à un changement d'état du système simulé causé par des données entrantes ou par un processus interne. Un état du système est défini par un ensemble de variables discrètes. Les événements sont produits à des instants appelés *dates*. Les variables continues du système sont prises en compte uniquement aux dates des événements.

Les simulations à événements discrets sont classées en fonction du du type de séquencement du temps par l'horloge de simulation.

Simulation dirigée par les événements. Le modèle dirigé par les événements n'utilise pas une horloge globale (incrémentée à intervalle régulier par un pas constant). Le temps de simulation est mis à jour par les dates associées aux évènements insérés dans la file d'attente du simulateur. Ainsi, le séquencement du temps de simulation correspond à un ordonnancement croissant des événements dans la file en fonction de leur date. La boucle principale de l'ordonnanceur d'événements consiste à lire le premier élément de la file, exécuter l'action associée et mettre à jour l'horloge. Ce type de modèle de simulation est adapté à des systèmes dont les événements ne sont pas contigus dans le temps e.g. la simulation d'un SoC prend en compte différentes latences comme les transferts de données sur le bus. Ainsi, il ne simule que les cycles utiles et ne prend pas en compte les cycles sans événement.

Simulation dirigée par le temps. Le temps de simulation d'un modèle dirigé par le temps est incrémenté par un pas d'horloge constant défini en fonction du problème adressé. Pour chaque pas l'ordonnanceur d'évènements recherche dans sa file les évènements dont la date correspond à celle de l'horloge. L'ensemble des événements sélectionnés est ensuite exécuté. Contrairement à la simulation dirigée par les événements, il est possible d'obtenir des cycles vides ce qui diminue les performances de la simulation. Le modèle dirigé par le temps est intéressant dans le cas d'une simulation RTL lorsqu'il est nécessaire de tracer à chaque cycle d'horloge les activités du circuit e.g. la mise à jour des bascules.

6.3.2 Cadriciel de simulation

Le simulateur système est fondé sur le noyau de simulation défini dans [49]. Ce noyau est composé d'un ensemble de classes qui permettent la création, l'ordonnancement et l'exécution des événements. Il s'appuie sur trois classes principales : *Simulator*, *SimulationObject* et *Event*.

L'interfaçage de ces trois classes avec *SmallSystem* est donnée par la figure 6.3. Dans une modélisation, l'instance de la classe *Simulator* correspond au point d'entrée pour démarrer et arrêter la simulation. De la même manière que les classes *SmallSystem*, le simulateur offre des capacités d'extensibilité par sous-classement de la classe *Simulator*. Ainsi, le concepteur peut spécialiser le simulateur en fonction des besoins de son domaine de modélisation.

6.3.2.1 Structure d'un modèle simulé

La modélisation d'un système est composée d'instances de *SimulationObject* qui fournit les fonctions pour l'ordonnancement d'évènements et l'application de latences dans la simulation. Les composants *SmallSystem* héritent de cette classe. Les événements sont des instances de la classe *Event* qui sont créés puis insérés dans la file d'évènements par l'instance de *Simulator*. La création d'événements est le résultat de l'ordonnancement des méthodes des composants qui correspondent à la spécification comportementale des activités internes (méthode *defineProcess*). Ces méthodes sont exécutées par la machine virtuelle Smalltalk et ne consomme pas de temps de simulation. Pour simuler les latences d'exécution des activités, des appels aux fonctions du simulateurs sont insérés dans les méthodes des composants. Deux fonctions principales sont disponibles : *holdFor : délai* et *logFor : délai name : nom*.

La première fonction crée une latence dans le simulateur d'une durée caractérisée par le paramètre *délai*. Ce délai peut être défini par une constante ou bien calculé par une loi de probabilité pour simuler des événements non-déterministes e.g. la contention sur un bus. Les lois de probabilité sont directement utilisées à partir d'un cadriciel défini dans [49].

La deuxième fonction interagit directement avec l'instance de la classe *SystemLogger*. En plus de créer un délai dans la simulation elle permet de tracer l'activité dans un journal sous l'étiquette définie par le paramètre *nom*. Ce journal sera ensuite utilisé par des outils de visualisation pour produire un diagramme de Gantt.

6.3.2.2 Combinaison des niveaux d'abstraction et des modèles de simulation

La simulation d'un SoC combine deux modèles de simulation à événements discrets. Le SoC est simulé à un niveau comportemental par un simulateur dirigé par les évènements. A cela s'ajoute la simulation, à différents niveaux d'abstraction, d'une application placée sur une unité reconfigurable.

```
connectorOut
        | readAccAdress81 value83 nextAdress79 |

        [true] whileTrue:[
                readAccAdress81 := 5.
                1 to: 12 do: [:index69 |

                        value83 := (self atChannel: #FToConOut) readWait.
                        self logFor: 1 name: #COut_Rcv.

                        (localMemory at: 2) at: readAccAdress81 put: value83.
                        self logFor: 1 name: #COut_Write.

                        nextAdress79 := readAccAdress81 + 24.
                        readAccAdress81 := nextAdress79.
                ].
        ]
```

Listing 6.1 – Code comportemental d'un processus de communication défini par une méthode Smalltalk.

Au niveau RTL l'application est simulée par un simulateur de type CABA dirigé par le temps. La simulation de deux niveaux d'abstraction différents dans un même modèle amène le problème de la combinaison de deux notions de temps différentes.

Pour synchroniser les deux modèles de simulation, la simulation CABA est embarquée dans le simulateur système comme une tâche atomique. Au niveau système, lorsqu'un signal d'activation est envoyé à une unité reconfigurable, l'application est démarrée par le lancement du simulateur CABA. Lorsque le calcul est terminé, le nombre de cycles d'horloge nécessaires à l'exécution est retourné. Cette latence est convertie dans l'unité de temps utilisée au niveau système et est utilisée pour caractériser la durée de l'événement discret. Ainsi, au niveau système la simulation apparaît comme dirigée par les événements.

6.3.3 Simulation multi-niveaux d'une application accélérée

6.3.3.1 Simulation de l'application au niveau comportemental et HL-CDFG

La simulation d'une application à un niveau comportemental est réalisée par une exécution logicielle. Les processus de communication et de calcul sont directement décrits en Smalltalk. Le code 6.1 donne un exemple de processus de communication de sortie. Il réalise les lectures dans les mémoires locales modélisées et applique les latences opératoires par des appels aux fonctions du simulateur (réception d'une donnée et écriture de la donnée en mémoire).

La simulation du HL-CDFG se situe à un niveau comportemental. Elle est réalisée par un composant inclus dans *SmallSystem*. Ce composant est un visiteur qui simule le CDFG suivant un ordonnancement ASAP. Pour chaque opérateur une latence est appliquée et l'exécution est tracée par le *SystemLogger*.

6.3.3.2 Simulateur CABA pour une application placée sur une unité reconfigurable

Principe de simulation La simulation CABA consiste à calculer les valeurs courantes des données transitant dans le circuit simulé. Un circuit est composé des éléments suivants :

– **Les données** qui sont prises en entrée des opérateurs et produites en sortie.

- **Les constantes** forcent l'évaluation de leurs consommateurs.
- **Les opérateurs** calculent leurs sorties suivant une exécution dirigée par les données.
- **Les bascules** ont pour comportement de retarder le changement de leur sortie au prochain cycle d'horloge.

L'algorithme de simulation parcourt à chaque pas de simulation (qui correspond à un cycle d'horloge physique) sa liste d'événements. Un événement est inséré dans la file du simulateur lorsqu'un changement de valeur intervient sur une entrée d'un opérateur ou d'une bascule. Par exemple, un changement sur l'entrée d'une bascule-D aura pour effet de placer un événement au cycle suivant pour le calcul de sa sortie. Un pas de simulation se termine lorsque tous les changements ont été pris en compte.

Au niveau du simulateur système, les interactions avec le LL-CDFG simulé sont réalisées par la définition de stimuli. La section 6.4 détaille le principe d'interaction entre le système et le circuit simulé.

Exploration des activités internes d'un LL-CDFG La simulation d'un LL-CDFG est représentée par un événement atomique dont la durée est retournée par le simulateur CABA. Contrairement à la simulation comportementale (code Smalltalk ou HL-CDFG), les événements internes du circuit ne peuvent être observés. Pour explorer les activités d'un circuit, le simulateur CABA fournit un ensemble de fonctions qui permet la définition de sondes liées à des signaux. Ces sondes tracent et archivent les valeurs qui transitent sur les signaux. Les valeurs obtenues sont ensuite prises en entrée d'outils de visualisation de chronogramme.

Un LL-CDFG est le résultat du synthétiseur haut-niveau Madeo+. Les signaux d'un LL-CDFG correspondent à la mise en œuvre des dépendances de données d'un HL-CDFG représentées par des variables dans le code de haut-niveau Smalltalk ou AvelC. Pour faciliter les opérations d'analyses aux différents niveaux d'abstraction il est nécessaire de garder un lien entre les différents niveaux. La figure 6.4 illustre le lien (*Association*) entre les données (*Data*) d'un HL-CDFG et les signaux d'un LL-CDFG (*CompositeData*). Les variables d'un HL-CDFG sont mises en œuvre sous forme d'un agrégat de valeurs binaires i.e. par une collection de données qui forme un composite [6].

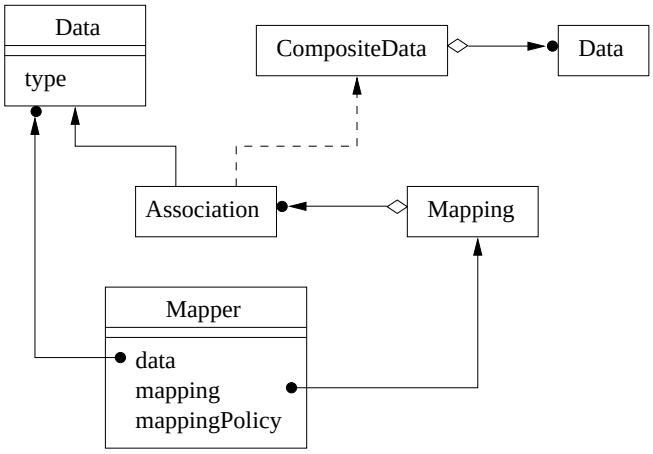


FIG. 6.4 – L'instance de la classe *Mapper* établit une correspondance entre la donnée de haut-niveau et son équivalent binaire RTL qui est un agrégat de bits.

Les liens sont conservés et gérés par un *Mapper* qui maintient la collection d'association. Le paramètre *mappingPolicy* permet la reconstruction d'une donnée composite vers une variable de haut-niveau. Par exemple, il spécifie l'application d'un ordre de type *little endian* sur la collection de bits.

6.4 Génération de bancs de test pour la validation de circuits

Dans un flot de conception matérielle standard, les applications sont validées par la simulation de spécifications HDL [111]. L'application est instanciée dans un banc de test qui définit un ensemble de stimuli agissant sur l'interface de l'application. Les bancs de test sont généralement écrits en langage de description matérielle, ce qui constitue une tâche longue et difficile dont la complexité augmente conjointement avec celle de l'application à valider. De plus, chaque banc de test est spécifique à une application et nécessite des phases de ré-écriture pour adresser d'autres cas.

Dans le but d'assurer l'interopérabilité de notre méthodologie de validation [121, 83] et d'augmenter la productivité des concepteurs, notre outil génère les bancs de test HDL à partir de modèles objets spécifiés à haut-niveau.

6.4.1 De l'objet mock au banc de test HDL

Les stimuli produits par un banc de test ont pour fonction d'imiter le comportement de l'environnement dans lequel s'exécute l'application, par exemple les activités systèmes. Dans le domaine de l'ingénierie logicielle et de la conception logicielle orientée objet, l'ensemble des stimuli que constitue le banc de test est centralisé dans un objet *mock*. Cet objet simule un autre objet en exhibant une interface identique et en fournissant des services identiques mais sans mettre en œuvre les fonctionnalités applicatives de l'objet simulé. En effet, un objet *mock* a pour première utilité de valider rapidement le comportement d'un objet de l'application en donnant au développeur un contrôle sur l'exécution.

Dans le cadre de la validation de *netlist*, l'objet *mock* est défini par l'ensemble des stimuli appliqués à la *netlist*. Les activités systèmes telles que les transferts de données dans un SoC sont modélisées et simulées dans SmallSystem. Les *netlists* à valider sont interfacées avec le modèle système par l'objet *mock* qui exploite l'API du simulateur bas-niveau. Les activités émulées par l'objet *mock* correspondent à l'intersection entre les activités systèmes et celles de la *netlist* à valider comme illustré par la figure 6.5.

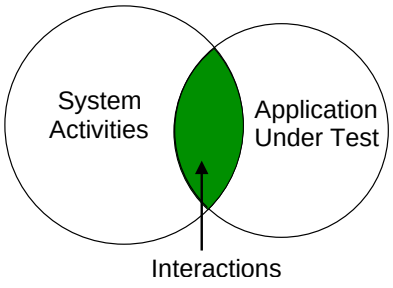


FIG. 6.5 – Interactions entre la *netlist* à valider et les activités systèmes émulées par l'objet *mock*.

```
1 Simulator
2   forceSignal: RESETn   "signal"
3   to: 1     "value"
4   at: 5.    "cycle
```

Listing 6.2 – Le signal RESETn est forcé à la valeur 1 au cycle 5 de la simulation.

Cet objet peut être créé et utilisé dans un modèle de simulation SmallSystem ou dans un script indépendant pour générer le banc de test Verilog équivalent. La figure 6.6 montre le flot de génération et l'utilisation du banc de test pour valider une *netlist* par des outils tiers de simulation.

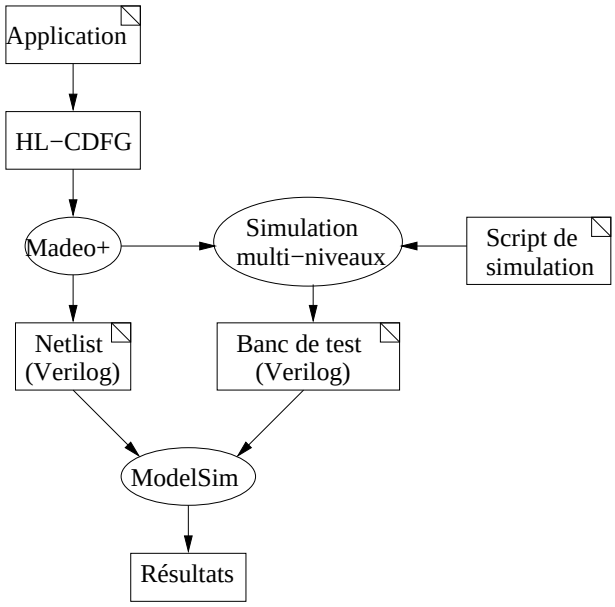


FIG. 6.6 – Simulation du banc de test généré par des outils tiers tels que ModelSim.

6.4.2 Modélisation des stimuli d'une application

Deux types de stimuli sont utilisés. Le premier type force un signal à une valeur à une date précise. Le deuxième type exécute une action définie par le concepteur, en réaction à un signal de l'application. Les stimuli sont définis dans un modèle instancié et utilisé par le simulateur de Madeo+. Le modèle obtenu correspond à un objet *mock* et permet de générer le banc de test HDL équivalent pour une simulation par des outils tiers.

Le code 6.2 donne un exemple d'instanciation d'un stimuli qui force le signal RESETn à la valeur 1 au cycle 5 de la simulation.

La mise en œuvre des communications entre une *netlist* et son environnement extérieur nécessite des synchronisations dynamiques. Pour cela, le deuxième type de stimulus permet d'exécuter une ou plusieurs actions en fonction des valeurs prises par un signal. Dans le domaine de la conception objet, ce type de mécanisme est mis en œuvre par le concept de dépendance, qui associe une méthode à exécuter à une variable. Lorsque la variable change

```
1 Simulator
2   onChange: dma_Ack_Read
3   relation: '='
4   value: 1
5   depName: 'dma_ack_read'
6   action: (SimulatorForceSignal
7                           forceSignal: (dma_Req_Read)
8                           to: 0
9                           in: 10).
```

Listing 6.3 – Dépendance associée à un signal. Le signal *dma_Req_Read* est forcé à 0 après 10 cycles lorsque le signal *dma_Ack_Read* prend la valeur 1. L'action peut être conditionnelle et plusieurs actions peuvent être associées à un signal.

de valeur, la méthode est exécutée. Le concept de dépendance est mis en œuvre par le *design pattern* : l'observateur [6].

Le code 6.3 définit une dépendance sur le signal *dma_Ack_Read*. Lorsque le signal prend la valeur 1 alors le signal *dma_Req_Read* est forcé à la valeur 0 après 10 cycles. L'envoi d'un message à la classe *SimulatorForceSignal* correspond à la création d'un stimulus[1].

L'objet *mock* peut être défini dans un modèle *SmallSystem* (concrètement dans une méthode d'un module) ou dans un script (en Smalltalk) indépendant qui définit les activités systèmes dont le circuit dépend. Ce dernier cas est principalement utilisé pour effectuer des simulations sans le coût d'une modélisation complète du système et également pour générer en Verilog le banc de test équivalent à l'objet *mock*.

6.4.3 Génération du banc de test HDL

La définition de l'objet *mock* dans un script en langage Smalltalk permet de bénéficier des fonctionnalités de test et de débogage de l'environnement du langage. De plus, l'approche *scripting* augmente la productivité du concepteur car le script nécessite deux fois moins de lignes de code par rapport à son banc de test HDL équivalent. Cela est principalement dû au niveau d'abstraction de l'API qui masque les détails bas niveaux tels que la déclaration des signaux ou l'instanciation des cellules.

La génération automatique du banc de test Verilog à partir de l'objet *mock* permet l'interopérabilité de notre méthodologie de validation avec des outils courants de simulation tels que ModelSim [111]. Le code 6.4 montre le Verilog généré pour les stimuli des codes 6.3 et 6.2. La déclaration des signaux, l'instanciation de la *netlist* et sa connexion au banc de test, non représentés ici, sont automatiquement générés.

6.5 Méthode de débogage logiciel pour le matériel

Cette section présente la méthodologie et l'environnement logiciel pour le débogage d'une application exécutée sur une unité reconfigurable.

6.5.1 Environnement logiciel pour le débogage de circuits

L'application de la méthode XP à la conception de circuit doit amener des bénéfices au niveau de la productivité des concepteurs [156]. Cependant, il est indispensable de mettre à

[1]La création consiste à instancier un bloc Smalltalk paramétrique qui définit le premier type de stimulus.

```
1  initial begin
2          fork
3                  ...
4                  #(5 * PERIOD) RESETn = 1;
5                  ...
6          join
7  end
8
9  always @(posedge dma_ack_read) begin
10   #(10 * PERIOD) dma_ack_read = 0;
11 end
```

Listing 6.4 – Verilog généré pour les deux stimuli.

disposition une méthode de débogage efficace supportée dans les outils. L'outil de débogage, nommé Red Pill, présenté dans ces travaux a été développé par L. Lagadec [84, 83]. La figure 6.7 donne une vue de l'interface principale du débogueur. Celle-ci est structurée en rubriques qui organisent les différentes parties d'une description Avel : processus de calcul, de communication et la topologie globale.

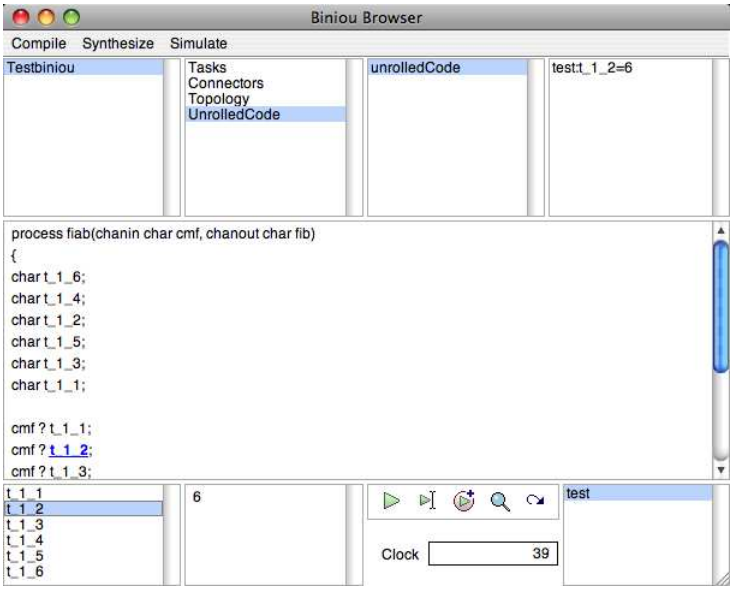


FIG. 6.7 – Vue de l'interface du débogueur Red Pill. La description Avel de l'application est organisée sous forme de rubriques dans l'interface [84].

Les sondes sont insérées dans le code AvelC des processus de calcul à travers une interface de haut-niveau inspirée par le débogueur de VisualWorks [141]. L'application est ensuite synthétisée pour produire une *netlist* instrumentée. Cette approche automatisée prévient les erreurs qui surviennent lors d'insertions manuelles et gère le statut de l'application déboguée.

6.5.2 De la sonde du logiciel au matériel

Les sondes du logiciel offre un moyen de vérifier l'état d'une exécution à un instant donnée. Une insertion ne modifie pas le code source mais affecte le flot d'exécution. De la même

manière, utiliser une sonde dans un circuit n'affecte pas son comportement mais peut entraîner des perturbations.

On distingue deux types principaux de sondes : le *Point d'Observation* (PO) et le *Point d'Arrêt* (PA). La première permet l'observation des valeurs que prend une variable au cours de l'exécution. Les valeurs sont collectées sans perturber l'application. La seconde arrête l'exécution lorsque la valeur d'une variable satisfait une condition définie par le concepteur. A partir du PA, le débogueur autorise l'exploration du flot d'exécution, par une exécution pas à pas et par l'analyse des contextes. Dans le cas de Smalltalk, l'état des objets ainsi que le comportement des méthodes sont modifiables dynamiquement.

6.5.2.1 Le point d'observation

L'équivalent matériel des variables du logiciel sont les signaux. Les PO sont insérés automatiquement au niveau de la *netlist* par la connexion du signal à l'interface *top* (voir figure 6.8). Ainsi, n'importe quel signal interne peut être rendu visible. Les traces des valeurs sont ensuite analysées après récupération.

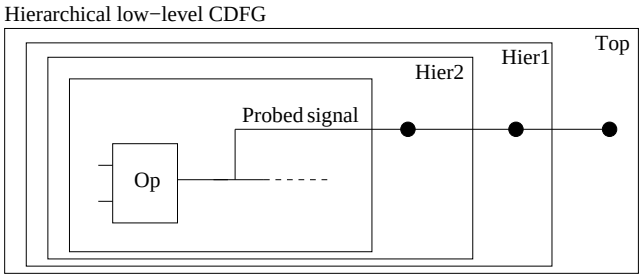


FIG. 6.8 – Un PO connecte le signal sondé à l'interface du module *top*. Les points représentent la traversée des hiérarchies.

Dans certains cas il n'est pas nécessaire de tracer l'ensemble des valeurs qui transitent sur le signal sondé. Pour permettre une filtration des valeurs, les PO peuvent être conditionnels. Un opérateur est utilisé pour établir une condition booléenne sur le signal. Il prend en entrée le signal sondé, la valeur à comparer et les bits de contrôle pour la sélection de l'opérateur de comparaison. Les valeurs sont tracées uniquement si elles satisfont la condition.

Les PO conditionnels sont également utilisés pour la mise en œuvre de PA. Ces derniers arrêtent l'exécution du circuit lorsque leur condition est évaluée à *vrai*.

6.5.2.2 Contrôle de l'exécution dans le matériel

Les outils logiciels de débogage ont la capacité d'arrêter l'exécution à un instant donné, d'exécuter pas-à-pas ou de relancer l'exécution d'une fonction. Dans le matériel, le flot d'exécution d'un circuit est commandé par un contrôleur local [106]. Pour rendre disponible des fonctionnalités de débogage similaires à celle du logiciel, le contrôleur local doit être étendu. Pour cela, un module de débogage est automatique ajouté au contrôleur local du circuit lors de l'étape de synthèse (il est enlevé une fois le circuit validé).

L'arrêt de l'exécution du circuit est réalisé par l'inhibition des registres. Le pas-à-pas est mis en œuvre par deux états : le premier active les registres et le suivant revient dans l'état initial d'arrêt (voir figure 6.9).

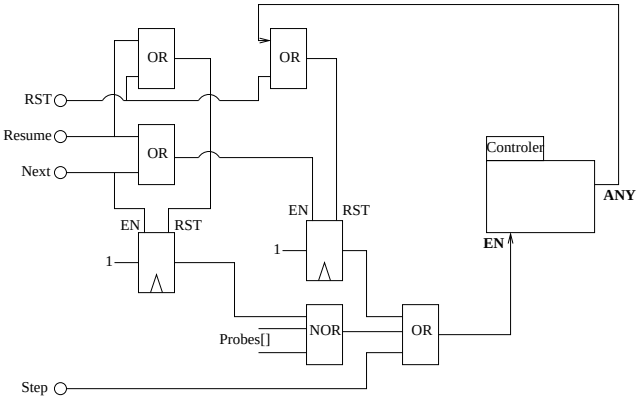


FIG. 6.9 – Schéma du contrôleur de débogage connecté au contrôleur central du chemin de données.

6.5.2.3 Le point d'arrêt conditionnel

La figure 6.10 illustre la mise en œuvre de deux PA conditionnels. Chacun prend en entrée la sortie de l'opérateur qu'il sonde puis réalise une comparaison sur la valeur. L'opérateur relationnel utilisé ainsi que la valeur de comparaison sont définis par les ports *OpSel* et (*Value*).

Leurs états sont capturés par le contrôleur de débogage par un *ou* logique. Par conséquent, l'activation d'une seule condition est suffisante pour arrêter l'exécution.

6.5.3 Coût du débogage

Notre méthode de débogage implique une instrumentation de la *netlist* qui entraîne un coût supplémentaire en ressources. Cette section évalue l'impact du débogage pour un filtre FIR-3 défini en section 5.4.1.4. L'architecture reconfigurable considérée est l'eFPGA M2000 avec l'utilisation de ses outils propriétaires pour le placement-routage de la *netlist* [3].

Le tableau 6.1 montre le nombre de fils et de cellules alloués en fonction du nombre de sondes insérées dans la *netlist*. L'application synthétisée *Base* ne contient aucune sonde. Par la suite des PO (non-conditionnels) et des PA sont ajoutés progressivement. Les PO sont placés sur les signaux qui correspondent aux réceptions et aux envois effectués dans les processus de communication (*Synth 1, Synth 2*). Un PO additionnel est placé sur l'opérateur de sortie du processus de calcul dans *Synth 3*. Les PO sont conservés et des PA sont ajoutés sur les même signaux dans *Synth 4, 5* et *6*.

Design	# of PO	# of PA
Base	0	0
Synth 1	1	0
Synth 2	2	0
Synth 3	3	0
Synth 4	1	1
Synth 5	2	3
Synth 6	3	4

TAB. 6.1 – Nombre de sondes insérées dans l'application et résultats de synthèse.

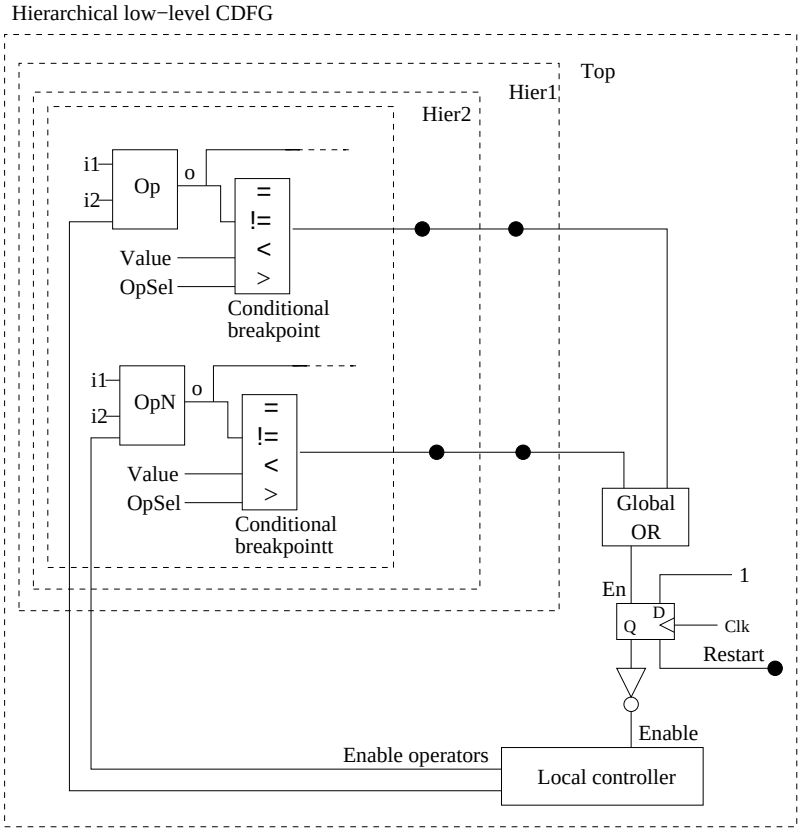


FIG. 6.10 – Structures des points d'arrêt conditionnels et connexion au contrôleur de débogage.

La figure 6.11 illustre le coût croissant de fils et de cellules. Pour des synthèses avec uniquement des PO insérés (*Synth 1, 2, 3*), le nombre de cellules reste constant puisqu'aucune logique supplémentaire n'est requise. Cependant, le nombre de fils augmente pour chaque PO et PA inséré. Un écart est présent entre *Synth 3* et *4* qui correspond à l'insertion des PA. Le nombre de cellules augmente pour la mise en œuvre des opérateurs de comparaison et du contrôleur de débogage.

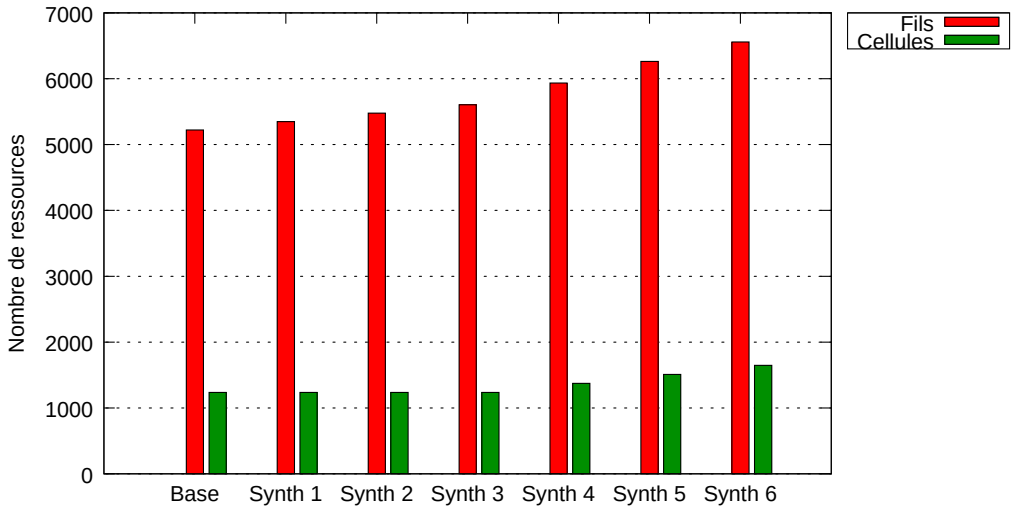


FIG. 6.11 – Evolution du nombre de fils et de cellules en fonction du nombre de sondes dans le circuit.

6.6 Conclusion

Les unités reconfigurables prototypées par DRAGE ont pour vocation à être embarquées dans des SoC. Par conséquent, la validation fonctionnelle des applications passe par la prise en compte de l'intégration système des unités.

Ce chapitre a présenté une méthodologie de validation fonctionnelle d'une application exécutées sur les unités reconfigurables d'un RSoC. Le flot de validation est fondé sur des cycles itératifs courts inspirés par les méthodes de développement logiciel telles que l'XP [156].

La plate-forme d'exécution est modélisée dans le cadriciel à composant SmallSystem. Pour la simulation, l'application est intégrée dans le modèle avec la prise en compte des activités systèmes. Deux niveaux de simulation de l'application sont disponibles : comportementale et RTL. Les résultats de simulation sont la visualisation des activités systèmes par la production de diagrammes de Gantt et d'interactions entre les composants. L'exécution de l'application synthétisée à un niveau RTL est détaillée par des chronogrammes qui tracent les valeurs des signaux.

La simulation est interfacée avec des outils tiers du commerce (e.g. ModelSim) par la génération d'un banc de test Verilog. Pour cela, les stimuli des activités systèmes sont capturés dans un modèle de simulation qui fait office d'objet *mock*. Cette technique est classiquement utilisée dans la conception orientée objet de logiciel pour valider le comportement des objets. L'objet *mock* constitue un modèle de simulation qui est exporté sous forme d'un banc de test.

Les étapes de simulation ont permis la mise en place de phases de test pour vérifier les résultats de l'application. Dans le cas où les tests échouent l'application doit être déboguée. La méthodologie proposée est une transposition du débogage du logiciel vers la synthèse de circuit. Des sondes sont directement insérées dans le code comportemental de l'application puis synthétisées pour obtenir une exécution performante. Le comportement d'un débogueur logiciel est restitué par l'ajout d'un contrôleur spécifique qui met à disposition des fonctionnalités équivalentes. Ainsi, la productivité des méthodes de conception logicielle est exploitée pour la validation fonctionnelle d'applications pour RSoC.

Chapitre 7

Cas d'étude

Les chapitres précédents ont abordé les trois axes principaux de notre méthode pour l'exploration architecturale d'unités reconfigurables :

- prototypage par modélisation des architectures (voir chapitre 4) ;
- programmation à haut-niveau des unités reconfigurables (voir chapitre 5) ;
- validation fonctionnelle multi-niveaux des applications sur un modèle de SoC (voir chapitre 6) ;

Ces trois axes fonctionnent en synergie et offrent au concepteur une approche globale pour converger vers une solution dans l'espace de conception. Ce chapitre donne une vision synthétique de leur application à des architectures reconfigurables grain-fin et gros-grain. En particulier, les capacités de mise en œuvre de la virtualisation des ressources sont illustrées sur une architecture gros-grain par un exemple significatif. En plus de valider le flot sur des cas concrets, ce chapitre offre également un tutoriel pour les futurs utilisateurs des outils.

7.1 Modélisations architecturales

Dans cette section nous modélisons et évaluons deux architectures reconfigurables de natures différentes. La première est une matrice d'opérateurs à gros-grain qualifiée par le terme CGRA. Elle est organisée en étages pipelinés similaires à l'architecture PiCoGA [98]. La deuxième architecture est à grain-fin et est inspirée d'un modèle générique similaire à VPR [16]. Elle est qualifiée par le terme FPGA.

7.1.1 Architecture à gros-grain

La figure 5.15 illustre une matrice CGRA de taille 3×3. La matrice est composée de deux types principaux de tuile : entrées-sorties et calcul. Pour une description complémentaire de l'architecture, le lecteur peut se référer à la section 5.4.2.

Chaque type de tuile de la matrice est modélisée en Madeo-ADL sous la forme d'un *composite*. Le code 7.1 donne la description partielle de la tuile *PRStripeEastCell*. Les éléments principaux sont la fonction, qui définit trois opérations possibles (addition, multiplication et *identité*), les multiplexeurs d'entrées-sorties de la tuile, les registres pour le tamponnage entre étages et l'aiguilleur qui établit les connexions dans un étage.

Madeo permet la modélisation de matrices de tuiles hétérogènes au niveau des ressources de calcul et de routage. Cependant dans notre exemple, les tuiles de calcul ont une structure

```
1  (((COMPOSITE
2    (((FUNCTION
3      (INPUTS ((WIRE (WIDTH 4)) NAMED fin0)
4         ((WIRE (WIDTH 4)) NAMED fin1))
5      (OUTPUTS ((WIRE (WIDTH 4)) NAMED fo))
6      (AMONG add mul id))
7    NAMED f)
8
9  ((REGISTER (INPUT ((WIRE (WIDTH 4)) NAMED in))
10   (OUTPUT ((WIRE (WIDTH 4)) NAMED out)))
11 NAMED regStripe0)
12 [...]
13 ((MULTIPLEXER
14   (INPUTS
15     ((WIRE (WIDTH 4)) NAMED inReg)
16     ((WIRE (WIDTH 4)) NAMED in0)
17     [...]
18     ((WIRE (WIDTH 4)) NAMED in5))
19   (OUTPUT ((WIRE (WIDTH 4)) NAMED out)))
20 NAMED inputMux0)
21
22 ((MULTIPLEXER
23   (INPUTS
24     ((WIRE (WIDTH 4)) NAMED in0)
25     ((WIRE (WIDTH 4)) NAMED in1)
26     ((WIRE (WIDTH 4)) NAMED in2))
27     (OUTPUT  ((WIRE (WIDTH 4)) NAMED out)))
28 NAMED outputMux0)
29
30 ((WIRE ( WIDTH (VALUE channelSize '#(4)'))) NAMED inputGlobalBus0)
31 ((WIRE ( WIDTH (VALUE channelSize))) NAMED east0)
32 ((LINK '(self relativeAt: -1@0) east0') NAMED west0)
33 [...]
34 ((((SWITCHBLOCK
35   (RESOURCES (foutSwitch east0 east1 east2 west0 west1 west2))
36   'self foutSwitch connectTo: self east0'
37   'self foutSwitch connectTo: self east1'
38   'self foutSwitch connectTo: self east2'
39   'self foutSwitch connectTo: self west0'
40   'self foutSwitch connectTo: self west1'
41   'self foutSwitch connectTo: self west2'
42   'self west0 connectTo: self east0'
43   'self west1 connectTo: self east1'
44   'self west2 connectTo: self east2')
45 NAMED switch) PRODUCE SwitchPRCell) CATEGORY CGRA))
46
47 (CONNECTION
48   'self inputGlobalBus0 connectTo: (self inputMux0 at: #in0) using: RaBuffer new'
49   [...]
50   'self regStripe0 connect: #out to:  (self inputMux0 at: #inReg) using: RaBuffer new'))
51 ) PRODUCE PRStripeEastCell) CATEGORY CGRA)
```

Listing 7.1 – Description en Madeo-ADL des éléments principaux d'une cellule de calcul de l'architecture CGRA.

homogène mais avec des différences sur les interconnexions. Par exemple, la tuile de calcul B de la figure 7.2(a) définit deux canaux de routage (*WIRE* en Madeo-ADL, voir section 3.3.2) pour se connecter aux tuiles A et C. Cette dernière utilise un alias pour se connecter à B et définit un canal pour les tuiles de droite. Ce type d'interconnexion est typique d'une modélisation Madeo et exploite ses mécanismes de réplication.

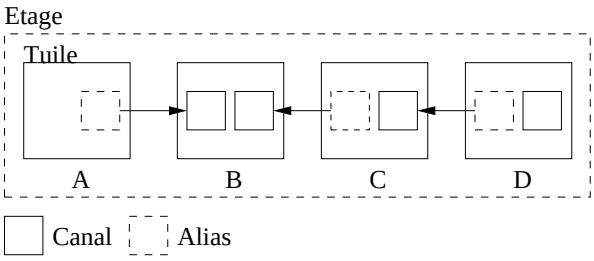


FIG. 7.1 – Différences entre les connexions des cellules de la matrice.

La tuile D est structurellement identique à la tuile C. Cependant, pour la génération du prototype matériel, elle doit être isolée par un nommage différent (paramètre de *PRODUCE*). Sa position périphérique implique qu'elle ne définit pas de connexions avec une tuile à droite contrairement à la tuile C. De ce fait, son interface de connexion inférée par l'outil sera différente avec moins de ports définis. Ainsi, les tuiles sont identiques au niveau du modèle Madeo mais différentes au niveau du prototype matériel.

Le positionnement des tuiles est donné par notre outil de visualisation dans la figure 7.2(a). Chaque type de tuile est associé à une couleur unique. Ce degré d'hétérogénéité (noté H) est caractérisé par le rapport entre le nombre de tuiles différentes (D) et le nombre total de tuiles de la matrice (T) :

$$H = \frac{D}{T} \tag{7.1}$$

Dans le cas de la 7.2(a), le degré d'hétérogénéité est fort avec $H = 0,83$. En revanche, la figure 7.2(b) montre qu'il diminue avec l'augmentation de la taille de la matrice avec $H = 0,03$.

La valeur H donne un indicateur sur le taux de réutilisation des descriptions des tuiles. Ainsi, plus la valeur H est faible et plus le nombre de descriptions à produire est faible en comparaison de la taille de la matrice. Cependant, cette valeur ne prend pas en compte la complexité de chaque tuile et ne suffit pas à définir complètement l'effort de modélisation. Pour cela, la section 7.1.3 prend en compte le nombre de lignes nécessaires à la description d'une architecture. Les résultats sont comparés au nombre de lignes VHDL générées.

7.1.2 Architecture à grain-fin

Notre flot d'exploration s'appuie sur la généricité de la modélisation Madeo. Dans cette section nous présentons une architecture à grain-fin de type FPGA détaillée par la figure 7.3.

Le FPGA est inspiré d'un modèle classique tel qu'utilisé par VPR [16]. Similairement à l'architecture CGRA la matrice est composée de deux types de tuile : entrées-sorties (IOB) et calcul. Cependant, la structure des IOB diffère du fait de la granularité des ressources de routage. Pour l'architecture CGRA, un canal de largeur n est considéré comme atomique et

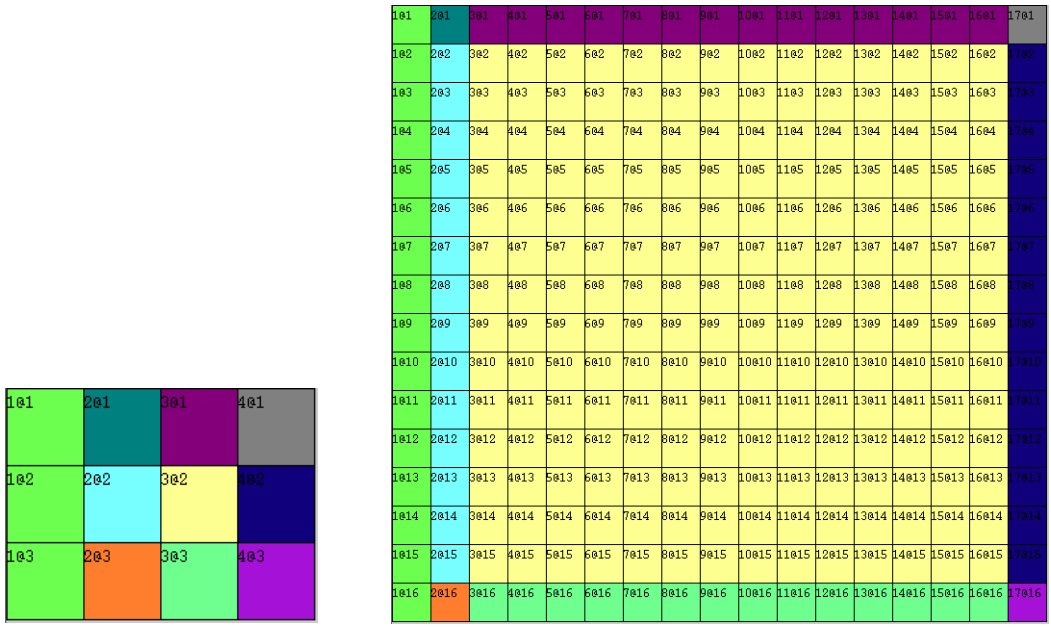


FIG. 7.2 – (a) Composition de la matrice CGRA visualisée dans notre outil de définition des zones reconfigurables. Chaque couleur est associée à une tuile différente. (b) L'augmentation de la taille de la matrice met en évidence les avantages de la réplication.

sera routé comme un seul fil. Ainsi, une pin d'un IOB a une largeur supérieure à 1 bit et est associée à un canal de routage de même largeur (voir figure 5.15). La relation de cardinalité entre les deux éléments est de 1 : 1.

Concernant le FPGA, ses canaux de routage sont discrétisés en fil de largeur 1 bit et les pins des IOB ont une largeur de 1. Par conséquent, un fil doit être sélectionné parmi les n fils d'un canal de routage. La relation de cardinalité entre un canal et une pin d'IOB est $n : 1$ pour une sortie et $1 : n$ pour une entrée. La sélection des fils est réalisée par des multiplexeurs et des tampons trois-états.

Une tuile de calcul est composée de deux canaux de routage interconnectés par un aiguilleur et de quatre blocs logiques. Un bloc logique contient une LUT à quatre entrées dont la sortie peut être tamponnée dans un registre. Les connexions des entrées sont réalisées par des multiplexeurs et celle de la sortie par des tampons trois états. Ainsi, on a pour les entrées une relation $1 : 1$ et pour la sortie une relation $1 : n$, avec n le nombre de fils dans un canal de routage.

Le code 7.2 donne un extrait de la description Madeo-ADL d'une tuile de calcul. Par rapport à CGRA la fonction ne définit pas d'opérateurs (mot-clé *AMONG*) et est considérée par défaut comme une LUT. Les canaux de routage (*WIRE*) ont l'attribut *EXPANDED* qui précise que le routage est effectué au niveau de leurs sous-éléments (les fils).

Les différentes tuiles de la matrice sont données par la figure 7.4. De la même manière que pour CGRA, les différences portent principalement sur les interconnexions entre les tuiles. Les IOB étant placés sur toute la périphérie de la matrice le nombre de tuiles de liaison avec les tuiles de calcul est plus important.

```
1  (((COMPOSITE
2    (((FUNCTION
3      (INPUTS ((WIRE (WIDTH 1)) NAMED in0)
4        ((WIRE (WIDTH 1)) NAMED in1)
5        ((WIRE (WIDTH 1)) NAMED in2)
6        ((WIRE (WIDTH 1)) NAMED in3))
7      (OUTPUTS ((WIRE ( WIDTH 1)) NAMED o)))
8    NAMED f1)
9  [...]
10 ((WIRE (WIDTH (VALUE channelW '#(20)')) EXPANDED) NAMED south    )
11 ((WIRE (WIDTH (VALUE channelW)) EXPANDED ) NAMED east)
12 ((LINK '(self relativeAt: -1@0) east') NAMED west)
13 ((LINK '(self relativeAt: 0@1) south') NAMED north)
14 [...]
15 ((MULTIPLEXER (INPUTS ((WIRE (WIDTH 20) EXPANDED) NAMED in))
16   (OUTPUT ((WIRE (WIDTH 1)) NAMED out)))
17 NAMED inputMuxf10)
18
19 ((((SWITCHBLOCK
20   (RESOURCES (south east north west))
21   'self north connectTo: self east'
22   'self north connectTo: self south'
23   'self north connectTo: self west'
24   'self west connectTo: self east'
25   'self west connectTo: self south'
26   'self east connectTo: self south')
27 NAMED switch) PRODUCE SwitchFPGAMiddleCell) CATEGORY LPPGA))
28
29 (CONNECTION
30   'self south connectTo: (self inputMuxf10 at: #in)'
31   [...]
32   '(self inputMuxf10 at: #out) connectTo: ( self f1 at: #in0)'
33   [...]
34   'self f1 connect: #o  to:self east'
35   [...])
36 ) PRODUCE FPGAMiddleCell) CATEGORY FPGA)
```

Listing 7.2 – Description en Madeo-ADL des éléments principaux d'une cellule de calcul de l'architecture FPGA.

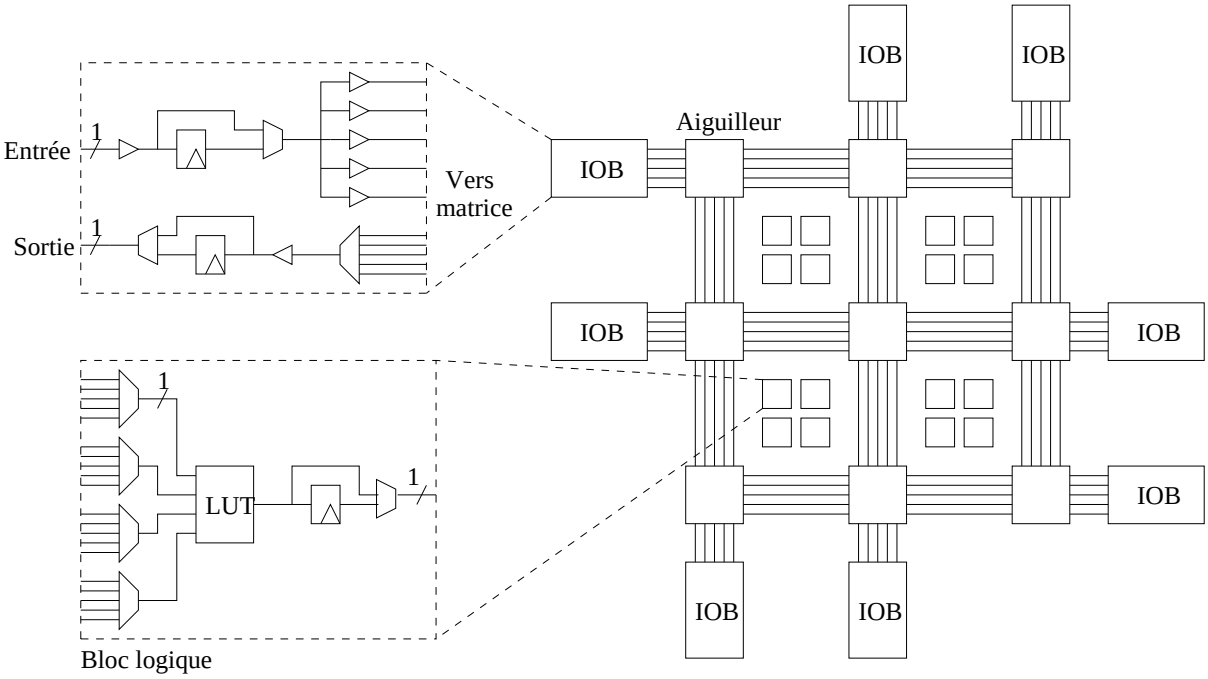


FIG. 7.3 – Structure de l'architecture FPGA.

La figure 7.4(a) montre tous les types de tuile de la matrice FPGA. Le degré d'hétérogénéité est de $H = 0,76$ pour la matrice de la figure 7.4(a) et de $H = 0,03$ pour la figure 7.4(b). La densité de calcul étant plus faible que pour CGRA la matrice doit être d'une certaine taille pour être exploitable. Par conséquent, le taux de réutilisation des descriptions tend à être élevé.

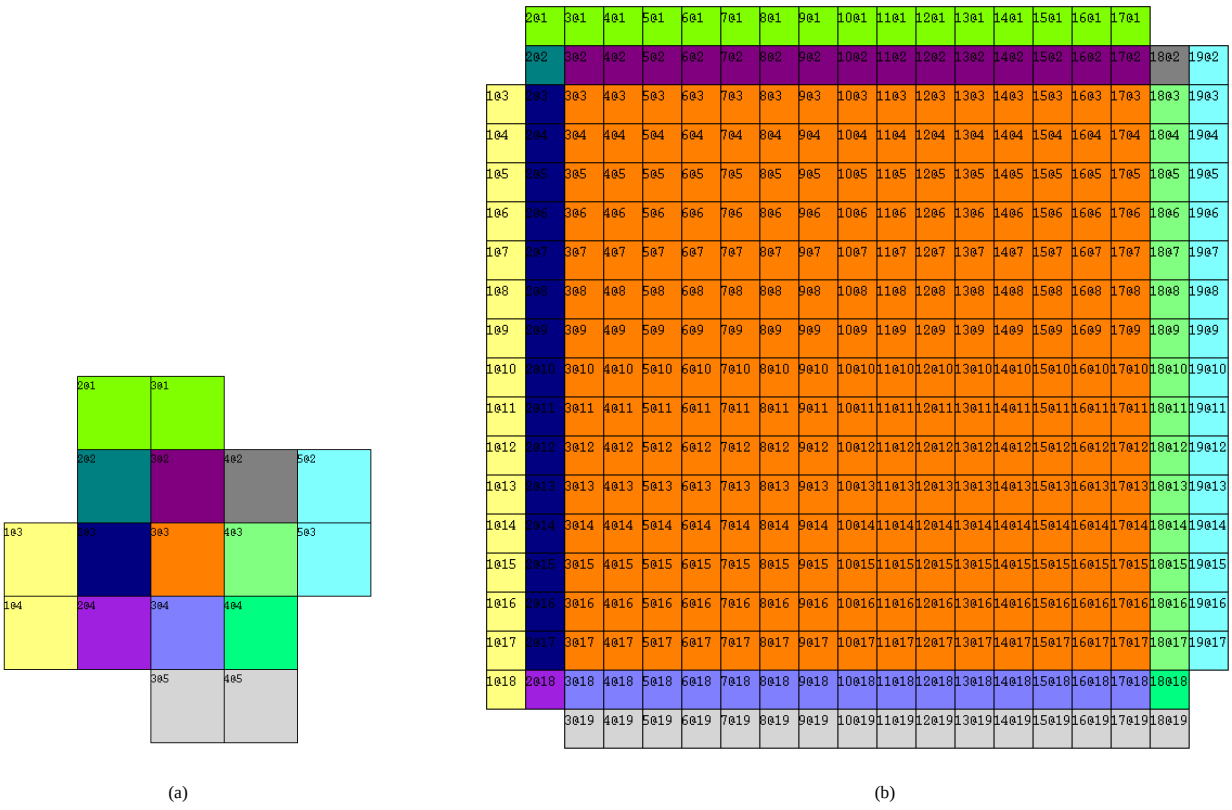
7.1.3 Evaluation de la charge des modélisations

Le prototypage d'une architecture reconfigurable par une approche ADL a pour objectif principal d'augmenter la productivité des concepteurs. Un premier gain de temps réside dans la diminution de la charge de programmation, autrement dit, du nombre de ligne de code à écrire. Cette section évalue le gain en terme de nombre de lignes de codes ADL par rapport au VHDL généré. Les impacts des variations de paramètre de la matrice et du plan de configuration sont inclus.

Le tableau 7.1 donne le nombre de lignes de code Madeo-ADL nécessaires pour la description des architectures. Dans le but d'évaluer des descriptions efficaces, certains aspects facultatifs qui concernent la représentation graphique de l'architecture ne sont pas pris en compte (e.g. déclaration d'un lien *LINK* pour représenter l'entrée d'un élément).

	CGRA	FPGA
Nombre de lignes	1120	1095

TAB. 7.1 – Nombre de lignes de code Madeo-ADL pour la description de l'architecture CGRA et FPGA.



(a) (b)

FIG. 7.4 – Visualisation de deux tailles de la matrice FPGA.

Le nombre de lignes de la description de CGRA est supérieur à celui du FPGA. Cela est dû au nombre de canaux de routage par tuile qui est plus important.

Le nombre de lignes ADL ne varie pas en fonction de la taille de la matrice. En effet, les dimensions sont prises en paramètres de la définition des domaines (voir section 3.3.2). De ce fait les deux nombres du tableau 7.1 serviront de points de référence.

Deux paramètres influent principalement sur le nombre de lignes de code VHDL générées : les dimensions de la matrice reconfigurable et le nombre de contextes. Ils ont un impact sur le nombre de cellules instanciées (cellules de l'architecture et registres de mémoire de configuration) et sur le nombre de fils pour les interconnexions.

Les tableaux 7.2 et 7.3 détaillent le nombre de lignes de code VHDL générées en fonction des dimensions de la matrice (nombre de tuiles) et du nombre de contextes dans la mémoire de configuration (C).

Nombre de tuiles	C = 1	C = 2	C = 3	C = 4
6	1925	2059	2193	2327
20	3779	4058	4337	4616
72	4251	4530	4809	5088
272	6059	6338	6617	6896

TAB. 7.2 – Nombre de lignes de code VHDL générées pour le prototype CGRA.

Nombre de tuiles	C = 1	C = 2	C = 3	C = 4
17	24846	27982	31118	34254
41	25662	28798	31934	35070
113	28350	31486	34622	37758
353	37953	41086	44222	47358

TAB. 7.3 – Nombre de lignes de code VHDL générées pour le prototype FPGA.

Les histogrammes de la figure 7.5 illustrent le gain en pourcentage de lignes de code économisées par une description ADL en comparaison du VHDL généré. Plus l'architecture augmente en terme de dimensions et nombre de contextes, plus le gain est important. Dans le cas du CGRA, le gain est maximal (73%) pour 4 contextes et un nombre de tuiles égal à 272. Dans le cas de la description du FPGA, l'approche ADL est intéressante dès le départ du fait de la complexité des ressources de routage au niveau bit et du nombre de fils déclarés qui en résulte.

7.2 Exploration du plan de configuration de CGRA

Les critères décisionnels pour structurer le plan de configuration sont fournis par une caractérisation des zones reconfigurables suivant différents critères.

Taille des binaires de configuration La taille des binaires de configuration est automatiquement extraite du modèle de *bitstream* généré. Le tableau 7.4 détaille l'évaluation de quatre matrices CGRA de taille croissante. Ainsi, ces résultats permettent au concepteur de définir

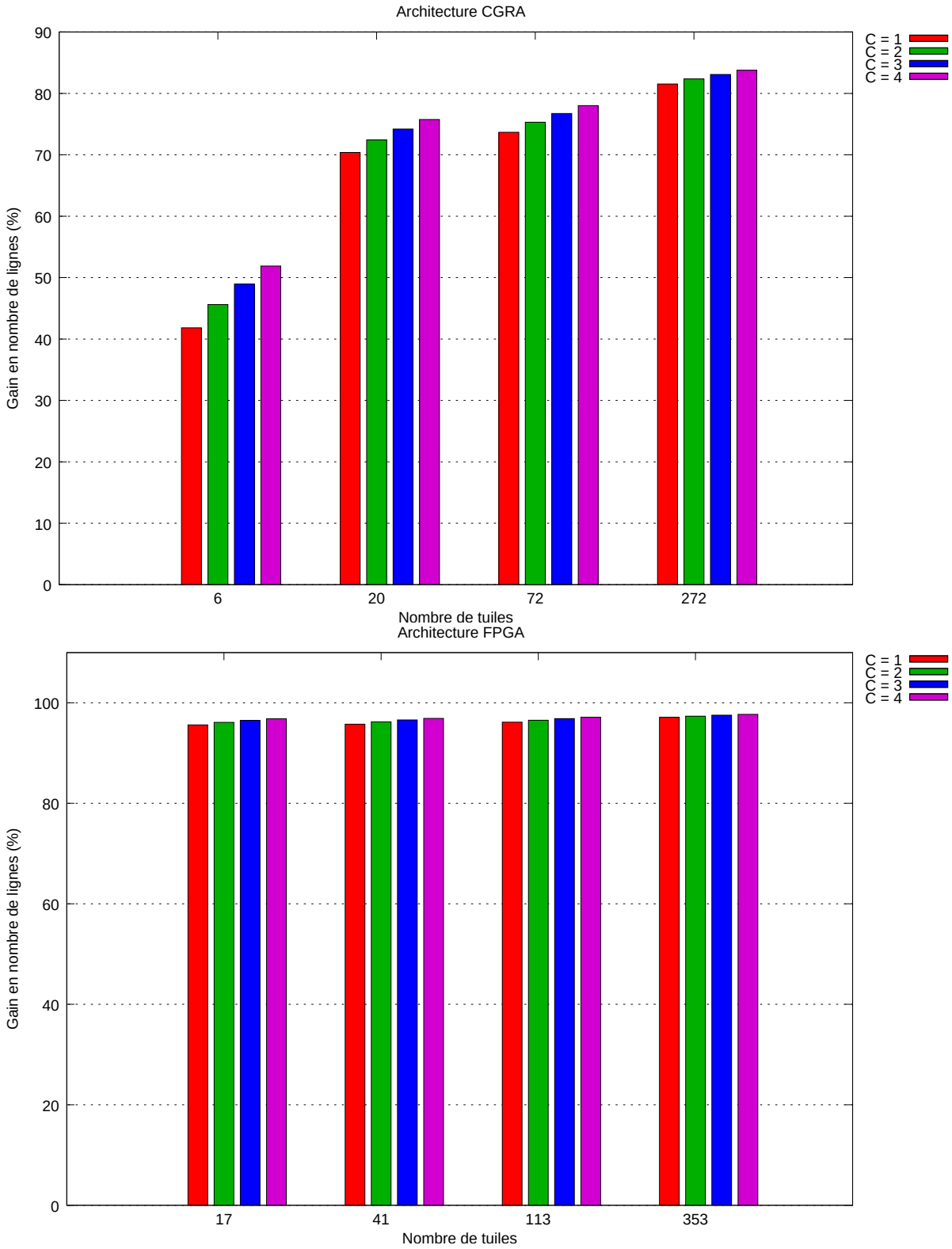


FIG. 7.5 – Gain en nombre de lignes de codes pour une description ADL par rapport au VHDL généré par l'outil.

les stratégies nécessaires au stockage des *bitstream* dans le SoC (dimensionnement mémoire, compression, etc.).

Dans le cas de l'architecture CGRA, les facteurs d'augmentation de la taille des matrices diffèrent en fonction du type de tuile (2 pour les tuiles d'entrées-sorties et 4 pour les tuiles de calcul). Ainsi, l'outil permet de mettre en évidence la croissance par un facteur non-constant de la taille du *bitstream*.

Tuiles E/S	Tuiles de calcul	Taille *bitstream* (bits)
2	4	156
4	16	536
8	64	2000
16	256	7712

TAB. 7.4 – Impact de l'évolution du dimensionnement des matrices sur les tailles de *bitstream*.

Coût et performance d'un contexte. Le nombre de registres d'une zone reconfigurable est dépendant du débit (exprimé en bits/cycle) choisi pour la zone reconfigurable (voir section 4.3.2). Si la taille du binaire de configuration n'est pas divisible par la valeur du débit, alors le nombre de registres est ajusté. L'ajustement est arrondi à la valeur supérieure ce qui constitue un sur-coût conditionné par la valeur du débit.

Le débit influe également sur les latences de configuration d'une zone. Elles ont un impact sur les temps d'exécution d'une application. Dans notre cas, nous évaluons la latence de configuration d'une zone reconfigurable qui est contrainte par le nombre de registres à configurer, le débit considéré pour le plan et le temps de configuration du contrôleur pour un mot. Pour ce dernier paramètre, nous prenons en compte un temps de configuration de 2 cycles/mot.

Le tableau 7.5 donne le nombre de registres en fonction de débits classiquement utilisés dans les mémoires. Cependant, il est possible d'explorer le plan de configuration pour n'importe quelle valeur de débit. Par exemple, la première matrice (6 tuiles) nécessite un ajustement constant de 4 bits. Or, un débit non-conventionnel de 26 bits permettrait d'éliminer ce sur-coût et d'obtenir une latence de configuration égale à 12 cycles.

Répartition des bits de configuration La figure 7.6 représente graphiquement le modèle de *bitstream* d'une tuile de calcul. La visualisation est effectuée via la génération par DRAGE d'une description Graphviz [53] du graphe. Ainsi, la répartition hiérarchique des bits de configuration est automatiquement documentée. Cela s'avère indispensable pour le développement de générateurs de *bitstream* externes à DRAGE mais compatibles avec les architectures prototypées.

La hiérarchie du modèle de la figure 7.6 est structurée sur trois niveaux. Le premier (*tilePRStripeWestCell*) est le mot de configuration de la tuile. Ses sous-divisions correspondent aux multiplexeurs d'entrées-sorties de la tuile (e.g. *inputMux0*), à l'unité fonctionnelle (*f*) et à l'aiguilleur (*switch*). Enfin, le dernier niveau regroupe les points d'interconnexion programmables de l'aiguilleur (e.g. *west0_east0*).

La représentation du modèle de *bitstream* pour un IOB est donnée par la figure 7.7. Le mot de configuration de la tuile est réparti entre les deux *iopads* qui définissent les pins d'entrées-sorties.

Nombre de tuiles	Débit (bits/cycle)	DFFs	Latence (cycles)
6	8	160	40
	16	160	20
	32	160	10
20	8	536	134
	16	544	68
	32	544	34
72	8	2000	500
	16	2000	250
	32	2016	126
272	8	7712	1928
	16	7712	964
	32	7712	482

TAB. 7.5 – Impact du choix des débits d'une zone reconfigurable sur le nombre de registres nécessaires et sur les latences de configuration.

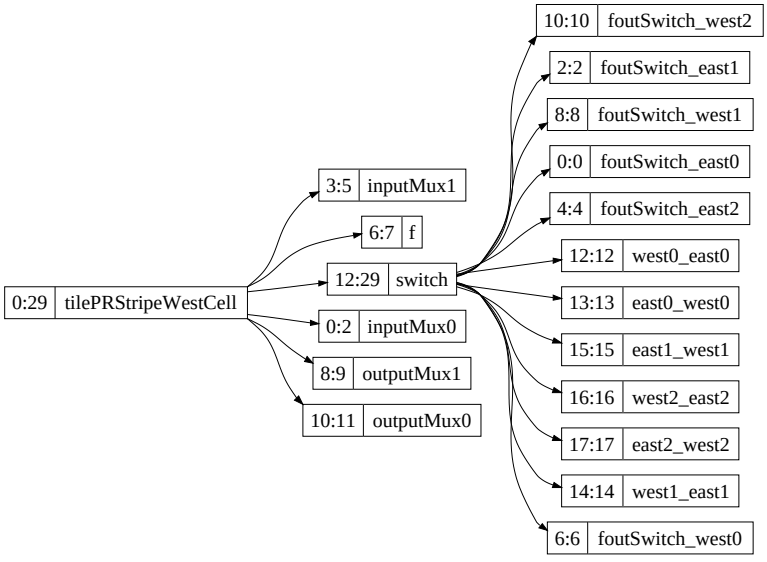


FIG. 7.6 – Répartition hiérarchique des bits de configuration pour une tuile de calcul de l'architecture CGRA. Le graphe est généré automatiquement par l'outil et permet d'obtenir une documentation complète sur la structure du modèle de *bitstream*.

FIG. 7.7 – Répartition hiérarchique des bits de configuration pour une tuile IOB de l'architecture CGRA.

7.3 Exploration de la virtualisation pour CGRA

De part sa granularité, l'architecture CGRA est orientée vers le domaine applicatif du traitement du signal. Dans cette section nous présentons la mise en œuvre d'un filtre numérique FIR sur l'architecture CGRA présentée en section 5.4.2. Pour cela, nous explorons les compromis entre ressources/performance avec trois dimensions de matrice. Si la taille d'une matrice est insuffisante pour mettre en œuvre l'application, ses étages sont virtualisés par multiplexage temporel. La virtualisation s'appuie sur les modes de reconfiguration dynamique partielle et multi-contextes. Le flot d'exécution obtenu est très proche de l'architecture PipeRench [50].

7.3.1 Mise en œuvre d'un filtre numérique

7.3.1.1 Présentation de l'application

L'application considérée pour ce cas d'étude est un filtre numérique FIR d'ordre 7 opérant sur 8 échantillons de largeur 8 bits (par la suite le nom du filtre sera FIR-8). Le corps de la fonction est décrit en AvelC pour produire un DFG. La méthode employée pour sa mise en œuvre est généralisable à des filtres de taille supérieure composés de sous-filtres cascadés.

La formule générale d'un filtre FIR est donnée par l'équation 5.3 et celle du FIR-8 est :

$$y(n) = \sum_{i=0}^{7} h(i)x(n-i) = h(0)x(n) + h(1)x(n-1) + ... + h(6)x(n-6) + h(7)x(n-7) \quad (7.2)$$

Nous prenons comme hypothèse que l'architecture CGRA ne contient pas de multiplieurs (similairement au PiCoGA) et que les coefficients sont connus à l'avance. Sur les 8 coefficients de notre exemple, 6 sont des puissances de deux ($h(1)$, $h(2)$, $h(4)$, $h(5)$, $h(6)$, $h(7)$, ce qui permet de réaliser les multiplications par des opérations de décalage logique, et deux coefficients ne le sont pas ($h(0)$ et $h(3)$). Pour ces derniers la multiplication nécessite l'utilisation d'une opération de décalage et d'une soustraction. Ce cas de figure implique l'utilisation d'opérateurs *identité* pour le transfert de données entre étages non-adjacents (voir section 5.4.2).

Le DFG du filtre FIR-8 est donné par la figure 7.8. L'ordonnancement est de typeALAP pour minimiser l'insertion d'opérateurs *identité*.

L'inconvénient majeur d'une approche sans multiplieurs est qu'une configuration est liée à la nature des coefficients (puissance de 2 ou non). L'utilisation de multiplieurs permet d'obtenir une structure de filtre plus générique. Par exemple, le multiplexage de plusieurs sous-filtres ne nécessiterait que deux types de configuration (un sous-filtre initial et un deuxième qui effectue le lien par une addition pour le cascadage). Ce choix d'exploration est possible par l'ajout de l'opérateur de multiplication dans les opérations possibles des fonctions. Cependant, l'utilisation exclusive d'opérations de décalage permet d'illustrer l'utilisation d'un nombre plus important d'opérateurs définis dans la section suivante.

7.3.1.2 Ensemble d'opérations des fonctions

L'analyse des opérateurs du DFG permet de déterminer l'ensemble des opérations possibles d'une fonction. La liste des opérations est définie par le concepteur dans la modélisation Madeo-ADL de l'architecture. Le code 7.3 donne la description d'une fonction.

Les opérations *add* et *sub* correspondent respectivement à l'addition et la soustraction. *sll* correspond à un décalage logique à gauche. Enfin, *id* est l'opérateur *identité*.

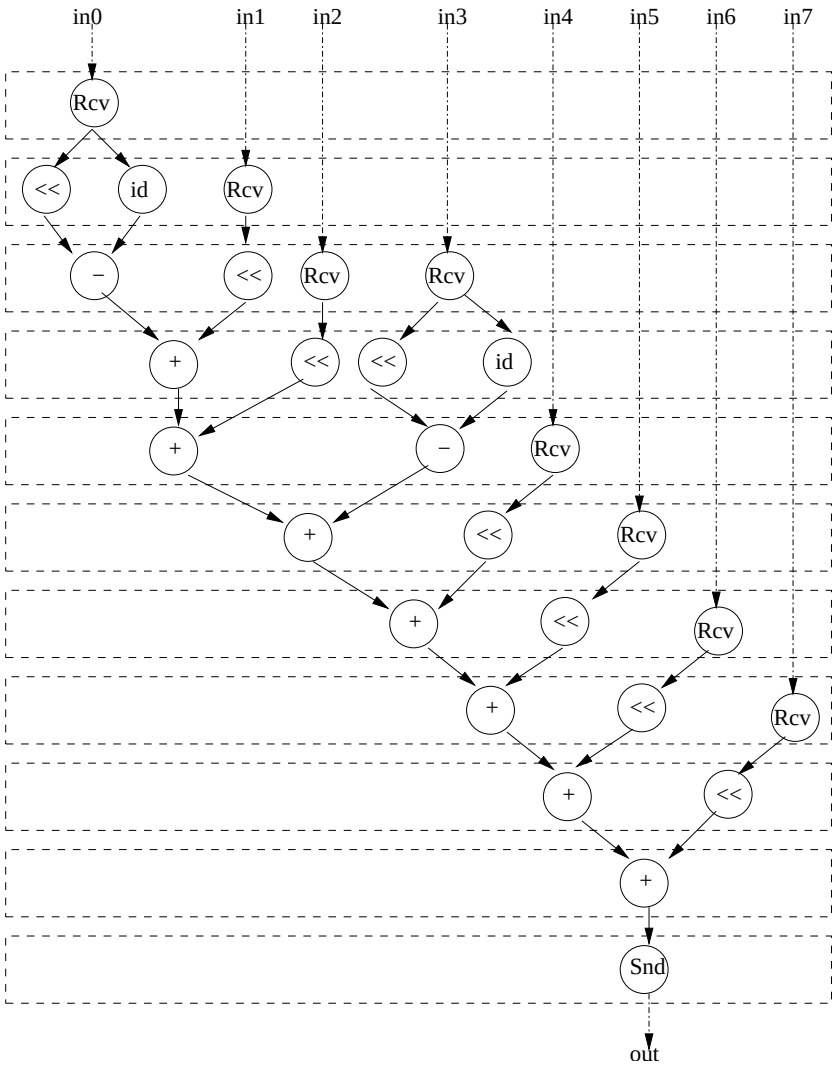


FIG. 7.8 – DFG du filtre FIR-8.

```
1 ((FUNCTION
2   (INPUTS ((WIRE (WIDTH 8)) NAMED fin0)
3       ((WIRE (WIDTH 8)) NAMED fin1))
4   (OUTPUTS ((WIRE (WIDTH 8)) NAMED fo))
5   (AMONG add sub sll id))
6 NAMED f)
```

Listing 7.3 – Description en Madeo-ADL d'une fonction de la matrice CGRA pour la mise en œuvre d'un FIR.

7.3.1.3 Mise en œuvre dans Madeo

Pour placer-router le DFG de l'application, une matrice CGRA de dimensions 10×5 est modélisée. Malgré le fait que 9 étages soient suffisants pour mettre en œuvre le DFG, 10 étages sont définis pour obtenir 5 groupes de 2. Par la suite, cette structure est utilisée pour le découpage de la configuration en vue de la virtualisation des étages (voir section 7.3.2).

Dans le cas initial le plan de configuration couvre l'ensemble de la matrice et a un seul contexte (voir figure 7.9(a)). Le nombre de ressources étant suffisant, les modes de reconfiguration dynamique partielle et multi-contextes ne sont pas utilisés.

Le tableau 7.6 donne la nature des ressources et la taille du binaire de configuration de la matrice.

Tuiles E/S	Tuiles de calcul	Taille *bitstream* (bits)
10	40	1380

TAB. 7.6 – Taille du binaire de configuration de la matrice CGRA 10×5.

La figure 7.10 donne une vue du placement-routage du FIR-8 dans l'éditeur Madeo. Les nœuds de réception sur les canaux de communication sont remplacés par une affectation des entrées de la fonction aux IOB de chaque étage. Les données envoyées à chaque étage sont les valeurs du signal d'entrée x et les coefficients du filtre. Le stockage de ces derniers comme constantes dans les tuiles n'est pas supporté par l'architecture. La mise en place de cette fonctionnalité nécessiterait une adaptation de l'extracteur de *bitstream* et du générateur d'architecture. Or, nous considérons que ce coût de développement n'est pas nécessaire dans les premières étapes d'exploration qui doivent être rapide. Par conséquent, nous privilégions une spécialisation minimale des outils et l'utilisation des algorithmes génériques. En revanche, dans les étapes de raffinement de l'architecture (e.g. dimensionnement des entrées-sorties), il s'avère indispensable.

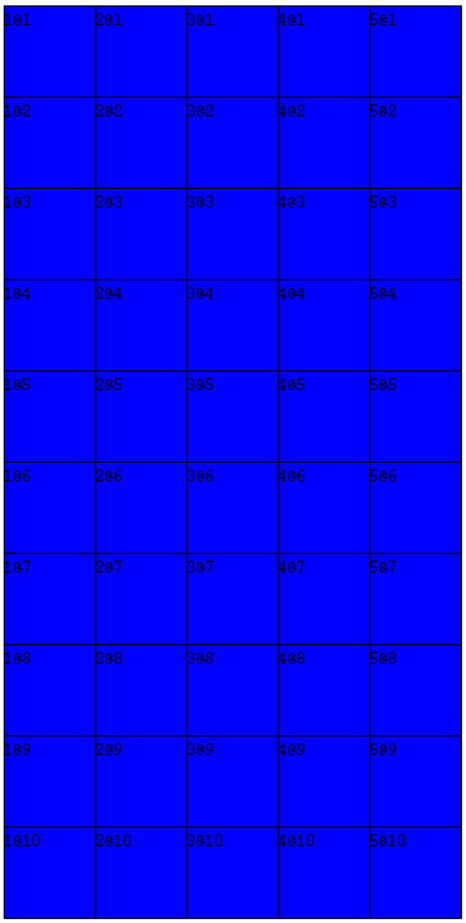
7.3.1.4 Exécution du filtre

Scénario d'exécution. La figure 7.11 illustre l'exécution du filtre sur la matrice 10×5. Le contrôleur de configuration charge la configuration dans le contexte 0 de la zone 0 (non représentée sur la figure), l'active (*Activ|0|0|*) puis se met en attente par une instruction *Sync*. Après activation, la *Zone 0* est configurée par la configuration *Config #0*.

Le chronogramme montre un exemple pour trois activations successives des étages de la matrice. Chaque étage activé est étiqueté par la valeur du signal de sortie y pour laquelle il calcul (*y0, y1, y2*). Le degré maximal de parallélisme entre les étages, pour le calcul de trois valeurs de y, est atteint au deuxième cycle. Ce degré est borné par l'intervalle $[1, N]$ avec N le nombre d'étages. La première valeur de y (*y0*) est produite au cycle 9 puis le débit est de 1 résultat par cycle.

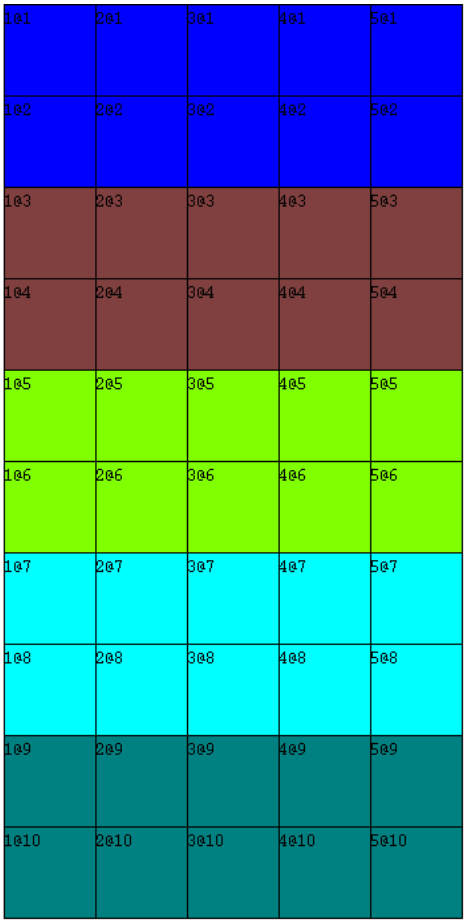
Alimentation en données. La gestion des envois des valeurs du signal d'entrée et des coefficients est prise en charge à l'extérieur de la matrice. La figure 7.12 définit une possibilité de mise en œuvre d'un *adaptateur* pour l'alimentation des étages de la matrice 10×5. Les valeurs du signal d'entrée et les coefficients sont envoyés en même temps dans des mémoires

(a)

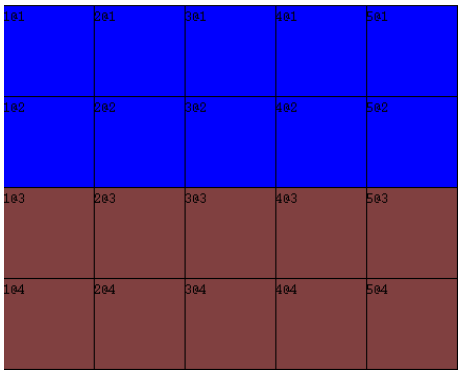



```
<configDomain named="D1" origin="1@1" corner="5@10" nContext="1"/>
```

(b)

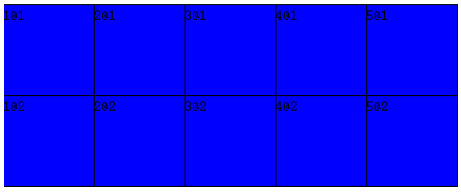



```
<configDomain named="D1" origin="1@1" corner="5@2" nContext="1"/>
<configDomain named="D2" origin="1@3" corner="5@4" nContext="1"/>
<configDomain named="D3" origin="1@5" corner="5@6" nContext="1"/>
<configDomain named="D4" origin="1@7" corner="5@8" nContext="1"/>
<configDomain named="D5" origin="1@9" corner="5@10" nContext="1"/>
```




```
<configDomain named="D1" origin="1@1" corner="5@2" nContext="3"/>
<configDomain named="D2" origin="1@3" corner="5@4" nContext="2"/>
```

(c)




```
<configDomain named="D1" origin="1@1" corner="5@2" nContext="5"/>
```

(d)

FIG. 7.9 – Les différentes structures du plan de configuration de l'architecture CGRA visualisées dans *Domain Viewer*. La description XML définit des zones représentées graphiquement dans différentes couleurs. Dans (b) le découpage permet l'extraction des pages de configuration. Elles seront ensuite multiplexées lors de l'exécution sur les zones des matrices de taille inférieure (c) et (d).

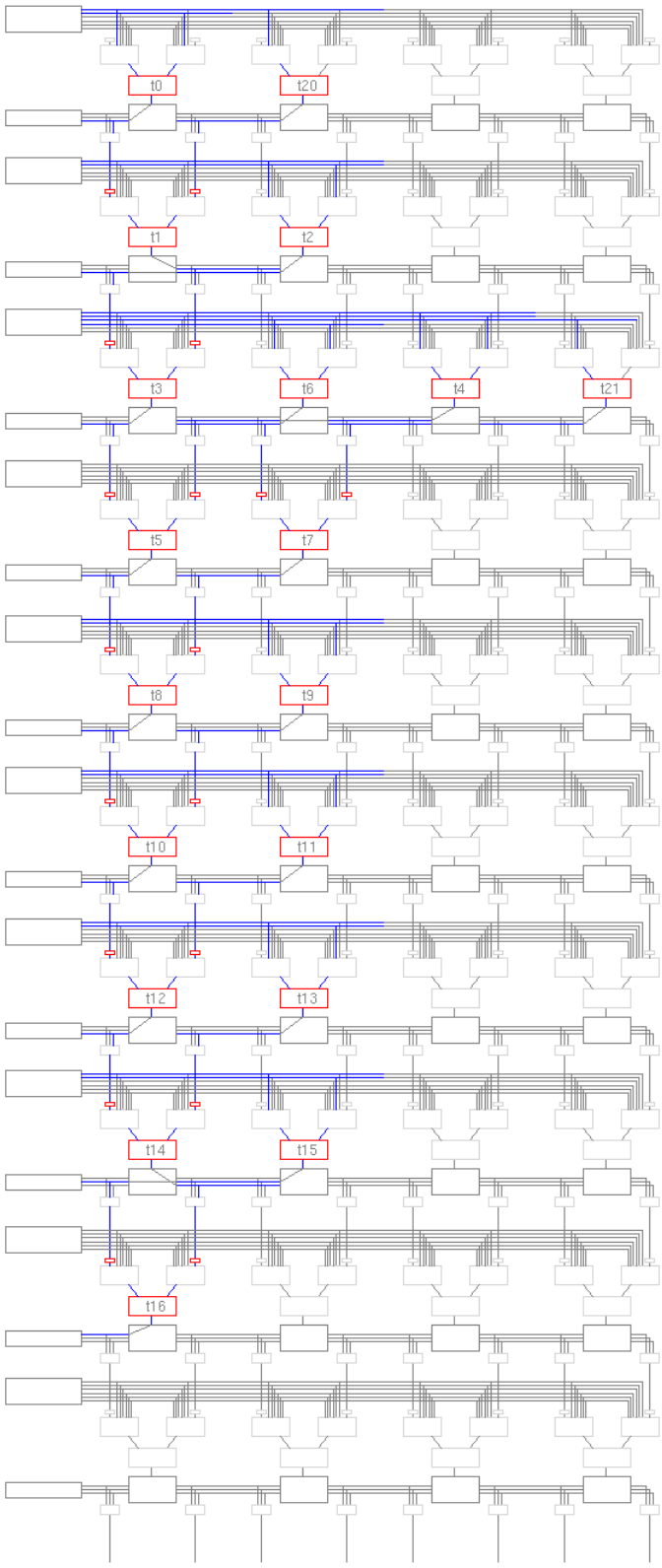


FIG. 7.10 – Vue dans Madeo du placement-routage du DFG du filtre FIR-8 sur la matrice CGRA.

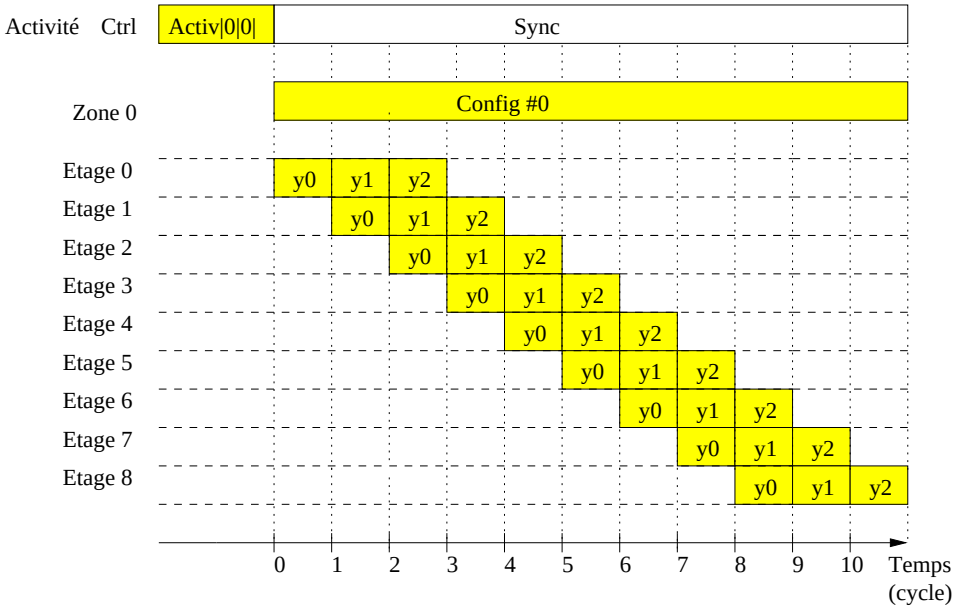


FIG. 7.11 – Chronogramme d'exécution du filtre sans virtualisation des ressources. La matrice met en œuvre 10 étages qui permettent un placement-routage complet de l'application.

locales connectées aux entrées-sorties de la matrice. Elles sont lues à chaque cycle d'horloge dans un mode FIFO par des générateurs d'adresse tels que définis à haut-niveau en section 5.2 (le pas de lecture - *step* - est égal à 1 et *count* est de la taille du signal x). Les envois sont synchronisés avec le *pipeline* de la matrice par la prise en compte des valeurs non significatives représentées par un trait dans la figure 7.12.

L'organisation et la lecture/écriture des données dans les mémoires locales de l'accélérateur sont des problématiques de niveau système. Elles sont abordées dans la section 7.5 avec la modélisation et la simulation système d'un SoC.

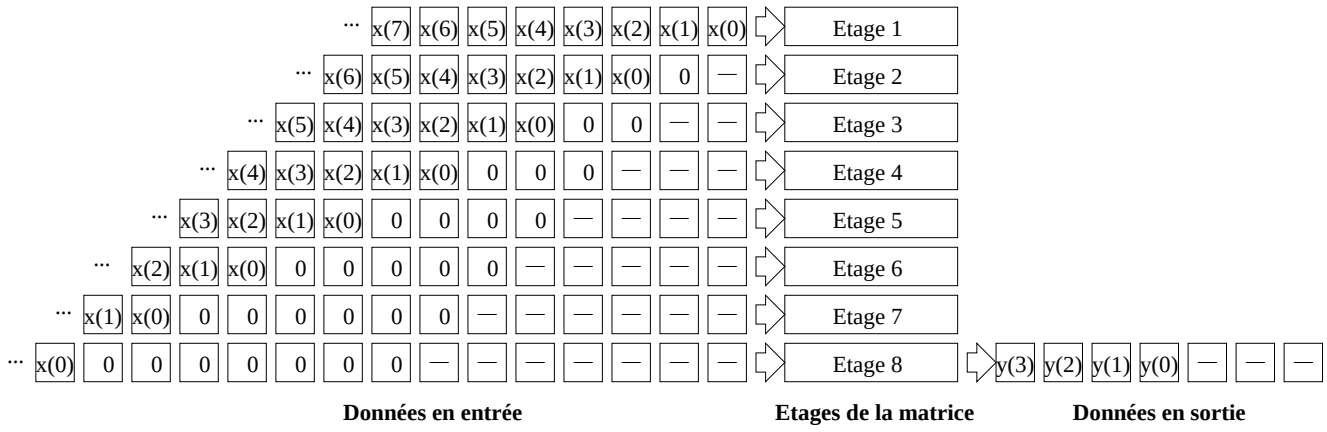


FIG. 7.12 – Envoi des données aux étages de la matrice CGRA.

7.3.2 Virtualisation des étages de CGRA

La section précédente décrit la mise en œuvre d'un filtre FIR-8 sur une matrice qui définit une seule zone reconfigurable mono-contexte. Ainsi la fonctionnalité de la matrice est déterminée par une seule configuration. Cette solution restreint le domaine applicatif à la mise en œuvre de DFG dont la profondeur ne peut dépasser le nombre d'étages de la matrice. Dans cette section nous prenons comme hypothèse que la taille de la matrice est insuffisante pour la mise en œuvre du filtre FIR-8. Pour permettre l'exécution d'applications dont la taille est supérieure au nombre de ressources, les étages de CGRA sont virtualisés.

7.3.2.1 Extraction des pages et structuration du plan de configuration

La virtualisation des étages consiste à multiplexer temporellement les étages physiques de la matrice entre plusieurs pages de configuration. Pour créer les pages, la matrice 10×5 est découpée en 5 zones reconfigurables comme illustré par la figure 7.9. Pour chaque zone un *bitstream* est généré par l'extracteur. Ainsi, le partitionnement en page est réalisé indépendamment de l'application. Il est à la charge du concepteur de garantir les interconnexions entre les pages (si elles existent) et leur cohérence à travers la définition de l'architecture.

Le nombre d'étages couverts par une zone reconfigurable est contraint par la latence du contrôleur de configuration pour un changement de contexte. Dans notre cas d'étude, une zone peut être reconfigurée tous les 2 cycles, ce qui correspond à la latence d'exécution de 2 étages de CGRA. Par conséquent, une zone reconfigurable minimale couvre 2 étages. Le tableau 7.7 donne la taille d'un contexte d'une zone de dimensions 2×5.

Tuiles E/S	Tuiles de calcul	Taille *bitstream* (bits)
2	8	276

TAB. 7.7 – Taille du binaire de configuration pour une zone de taille 2×5.

Le découpage de la matrice 10×5 permet d'extraire l'application placée-routée (voir figure 7.10) en 5 pages de configuration. Chaque zone reconfigurable étant identique en terme de structure de *bitstream*, elles sont configurables avec n'importe quelle page. Ainsi, il est possible d'exécuter différentes pages sur une même zone reconfigurable. Cette propriété est indispensable pour le multiplexage temporel.

Pour ce cas d'étude deux tailles de matrice sont considérées : 4×5 et 2×5, qui correspondent à un repliement de la matrice 10×5. Le plan de configuration de la matrice 5×4 est découpé en deux zones comme illustré par la figure 7.9(c). La première définit 3 contextes et la seconde 2 contextes. La matrice 5×2 correspond à la taille minimale pour la virtualisation et définit une seule zone de 5 contextes qui contient les 5 pages (voir figure 7.9(d)).

La figure 7.13 illustre l'organisation des contextes de configuration en fonction du nombre de zones des matrices. Plus la taille de la matrice est réduite et plus le nombre de configurations multiplexées sur chaque zone augmente.

7.3.2.2 Impact de la virtualisation sur l'exécution

La virtualisation des étages impacte le flot d'exécution de l'application sur l'architecture CGRA. La figure 7.14 détaille le multiplexage des pages sur les deux zones reconfigurables de la matrice 4×5. Avant leurs activations les pages sont toutes chargées dans les contextes

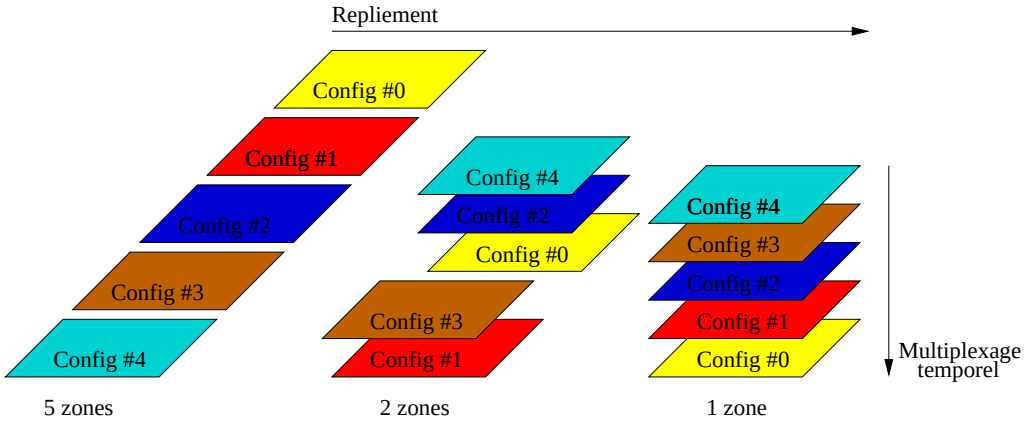


FIG. 7.13 – Repliement des contextes de configuration en fonction de la réduction des tailles de matrice.

des zones. Nous ne considérons pas de stratégie de recouvrement entre la configuration et l'exécution de l'application. Ainsi la latence de chargement est égale à celle de la matrice 10×5. Les contextes sont ensuite activés séquentiellement suivant l'ordonnancement spécifié par le microprogramme du contrôleur.

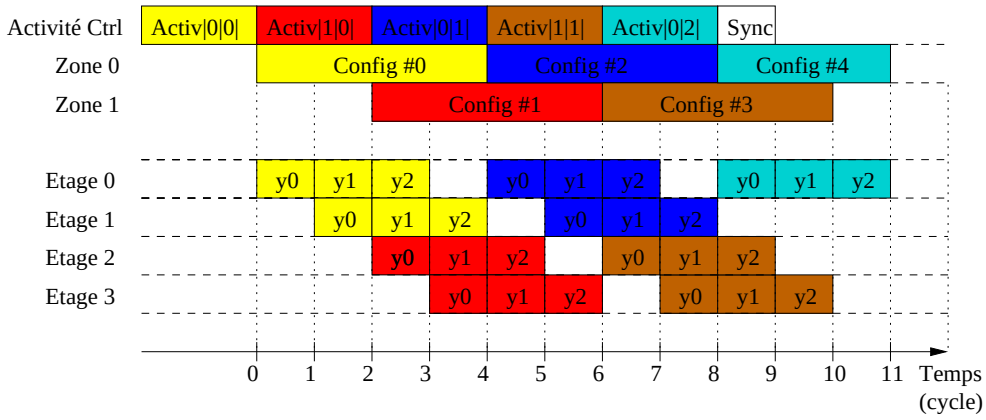


FIG. 7.14 – Chronogramme d'exécution du filtre avec la virtualisation des étages de la matrice par groupe de deux. Le nombre d'étages physiques est de quatre et les configurations sont multiplexées sur ces deux groupes de 2.

Dans le cas de la figure 7.14 les zones sont configurées en alternance par la sélection des différents contextes (voir figure 7.15).

Le nombre de zones reconfigurables utilisées conditionne le débit de l'application. Dans le cas de la figure 7.14 trois valeurs de y sont produites tous les 11 cycles. L'utilisation de deux zones permet d'obtenir un recouvrement temporel entre l'exécution de deux configurations. L'itération recommence au cycle 12 ce qui nécessite la mise en place d'un délai de 1 cycle avant l'activation de *config #0*. En revanche, dans le cas de la figure 7.16, les contextes de la zone sont activés séquentiellement. Ainsi, une valeur de y est produite après 9 cycles pour la

FIG. 7.15 – Chronogramme de simulation du multiplexage des configurations sur une matrice CGRA.

première itération puis tous les 10 cycles. Un délai de 1 cycle doit être pris en compte pour l'activation de *config #0* aux itérations suivantes.

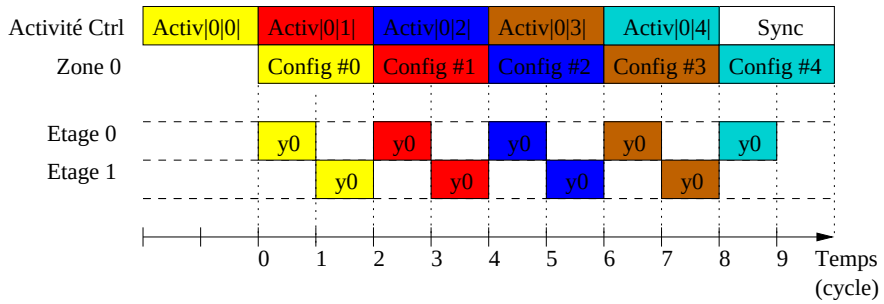


FIG. 7.16 – Chronogramme d'exécution du filtre sur un seul groupe de 2 étages. Les 5 contextes sont activés séquentiellement.

7.3.2.3 Evaluations

Les latences d'exécution de chaque matrice sont évaluées en fonction de la taille du signal d'entrée x. Nous prenons comme hypothèse que les plans de configuration ont un débit de 16 bits/cycle. Le tableau 7.8 donne les deux latences de configuration à prendre en compte dans le temps global de calcul pour les matrices 4×5 et 2×5. La latence de la matrice 10×5 correspond à un plan de configuration qui couvre l'ensemble de la matrice. Les valeurs suivantes correspondent aux latences de configuration d'une seule zone reconfigurable (36 cycles) multipliées par le nombre de pages de configuration à charger (5 pages). A cela s'ajoute la latence du contrôleur de configuration (2 cycles) qui est prise en compte dans le calcul.

Matrice 10×5	Matrice 4×5	Matrice 2×5
174	180	180

TAB. 7.8 – Latence de configuration (en cycles d'horloge) des matrices CGRA en fonction de leur taille.

Le tableau 7.9 donne les latences d'exécution en nombre de cycles d'horloge pour chaque taille de matrice. Les coût fixes d'initialisation sont les latences de chargement des configurations puis l'activation du premier contexte. Le multiplexage étant plus important sur la matrice 2×5, le coût de la virtualisation est supérieur aux autres matrices.

La figure 7.17 illustre l'évolution du sur-coût de la virtualisation en nombre de cycles supplémentaires par rapport à une exécution non-virtualisée sur une matrice 10×5. Le facteur

Taille de x	Matrice 10×5	Matrice 4×5	Matrice 2×5
128	312	654	1462
256	440	1126	2742
512	696	2062	5302
1024	1208	3942	10422
2048	2232	7694	20662

TAB. 7.9 – Temps d'exécution (en nombre de cycles) en fonction de la taille du signal x pour les différentes tailles de matrice.

principal de dégradation des performances est le débit des matrices qui est dépendant du groupement des étages par zone et du nombre de zones. Ainsi le sur-coût est d'autant plus élevé que le nombre de zones reconfigurables est réduit.

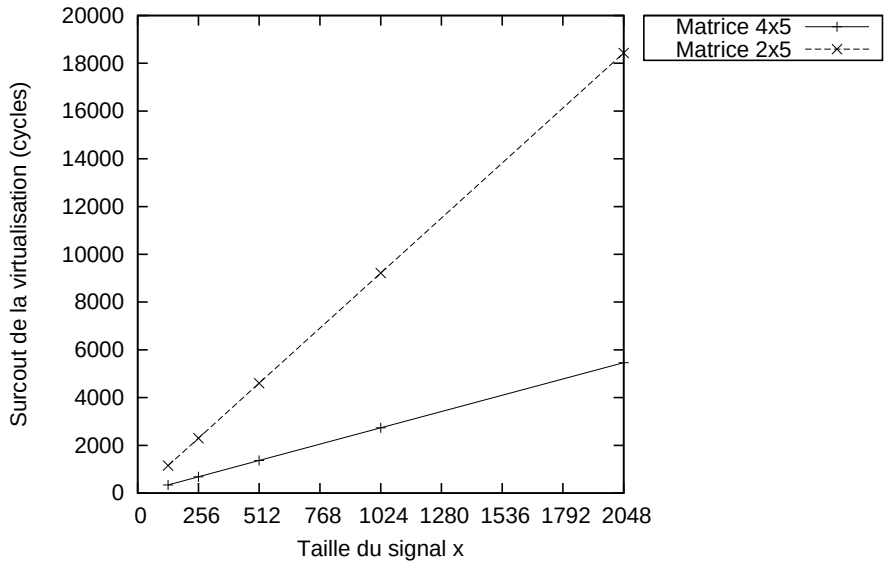


FIG. 7.17 – Evolution du sur-coût de la virtualisation en nombre de cycles par rapport à une exécution non-virtualisée.

La figure 7.18 illustre le débit moyen de chaque matrice pour différentes tailles du signal d'entrée x. L'évolution des courbes montre l'influence décroissante des coûts fixes d'initialisation (latences de configuration des matrices) au fur et à mesure de l'augmentation de la taille du signal x. Ainsi, les débits de chaque matrice tendent vers leurs valeurs maximales qui sont : 1 résultat/cycle pour la matrice 10×5, 0,27 résultat/cycle (3 résultats tous les 11 cycles) pour la matrice 4×5 et 0,1 résultat/cycle (1 résultat tous les 10 cycles) pour la matrice 2×5.

Les performances de crête (on considère un débit maximal) de chaque matrice sont caractérisées en nombre d'instructions par cycle. L'application FIR-8 requiert 8 opérations de décalage logique, 2 soustractions et 7 additions ce qui fait un total de 17 instructions pour le calcul d'une valeur de y. Le tableau 7.10 montre la dégradation des performances en fonction

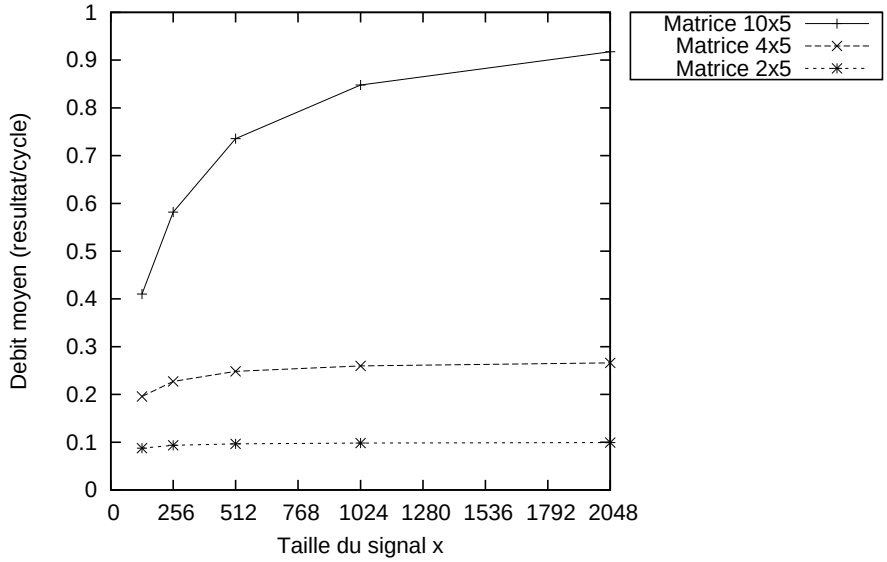


FIG. 7.18 – Débit moyen pour chaque matrice en fonction de la taille du signal d'entrée.

du degré de virtualisation. En particulier, le parallélisme d'instruction exploité par la matrice 2×5 est très faible, ce qui limite son intérêt par rapport à l'utilisation d'un processeur. Une possibilité d'amélioration des performances est de réduire la pénalité de changement de contexte par le contrôleur à 1 cycle (à la place de 2). Cela entraînerait une amélioration des débits.

Matrice 10×5	Matrice 4×5	Matrice 2×5
17 instructions/cycle	5 instructions/cycle	2 instructions/cycle

TAB. 7.10 – Performance des matrices (en nombre d'instructions par cycle) en fonction du degré de virtualisation.

Ces valeurs permettent de comparer les performances de chaque matrice et de quantifier l'impact de la virtualisation. La perte de performance engendrée par le multiplexage des groupes d'étages est à pondérer par le gain en nombre de ressources. Ainsi, le flot d'exploration permet au concepteur de déterminer un compromis ressources/performances qui répond, d'une part aux contraintes matérielles du SoC et d'autre part, à celles de l'application considérée (contraintes temps réel).

7.4 Prototypage de l'architecture CGRA sur une carte FPGA

La section précédente a détaillé l'évaluation de l'architecture CGRA par l'exploration du modèle et la simulation de l'architecture générée. Ces étapes permettent de déterminer les caractéristiques principales d'une architecture qui répond aux contraintes définies par le concepteur. Cette section aborde le prototypage matériel de l'unité reconfigurable sur une carte FPGA. La mise en œuvre a été réalisée par Y. Corre dans le cadre de ses travaux de thèse. La matrice générée et son contrôleur sont couplés à un processeur *soft-core* pour obtenir une plate-forme complète d'exécution. A ce niveau, l'unité reconfigurable est évaluée

plus finement (contraintes de ressources, gestion mémoire, etc.) en vue de sa mise en œuvre physique comme ASIC ou comme *soft-core*.

7.4.1 Principe de mise en œuvre

Plate-forme de prototypage. La plate-forme utilisée pour le prototypage matériel de l'unité reconfigurable est la carte Xilinx XUPV5-LX110T [148]. Elle embarque un FPGA Virtex-5 (XC5VLX110T) [154] d'une capacité logique de 17280 *slices* qui contiennent chacun 4 LUT-6 et 4 bascules. A cela s'ajoute 64 *slices* DSP qui regroupent des opérateurs du traitement du signal (25×18 multiplieurs, un additionneur et un accumulateur). Ces derniers ne sont actuellement pas utilisés pour la mise en œuvre de CGRA.

Couplage de l'unité. Le prototype architectural considéré est un processeur reconfigurable dont le couplage est de type coprocesseur (voir section 2.3). La figure 7.19 détaille la structure globale de l'architecture.

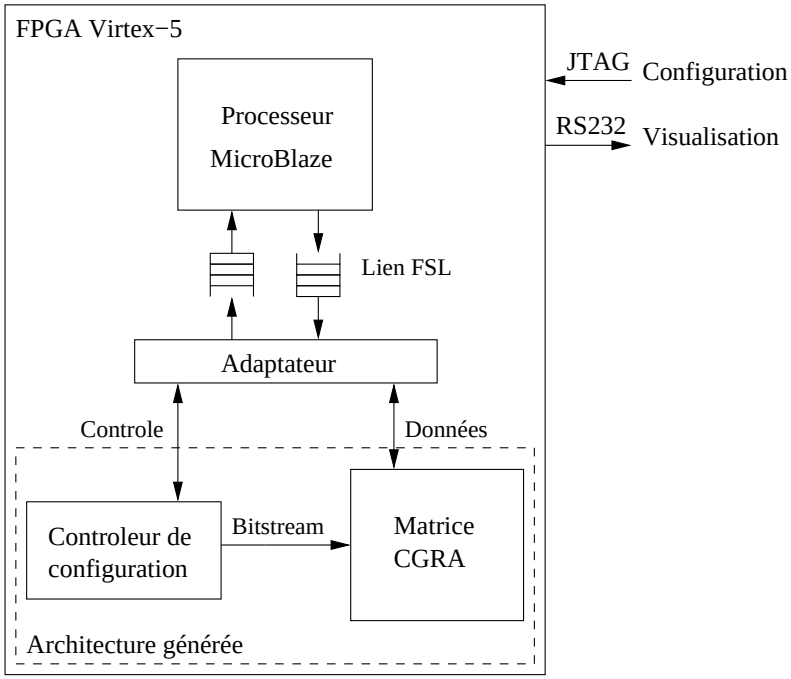


FIG. 7.19 – Principe de mise en œuvre de l'accélérateur CGRA sur la carte Xilinx XUPV5-LX110T. La matrice reconfigurable et son contrôleur de configuration sont couplés à un processeur Microblaze.

L'unité reconfigurable est couplée à un processeur *soft-core* Microblaze [147]. L'unité reconfigurable est composée des deux éléments générés qui sont la matrice CGRA et son contrôleur de configuration. Les communications avec le processeur sont réalisées par l'intermédiaire d'un adaptateur. Celui-ci est réalisé manuellement pour transférer les données nécessaires aux calculs ainsi que les informations de contrôle pour les configurations et les synchronisations. Il est connecté au Microblaze par un bus *Fast Simplex Link* (FSL) dédié au couplage d'accélérateurs mis en œuvre sur FPGA [150]. Dans le cas du Microblaze les données sont transférées

entre le banc de registre du processeur et l'accélérateur via les interfaces FSL. Les canaux de communications sont des mémoires FIFO uni-directionnelles.

Du point de vue de la programmation les données sont transférées par l'utilisation de fonctions incluses dans le jeu d'instruction du Microblaze. Elles sont utilisables comme primitives pour la mise en place d'un modèle de programmation de type Molen [140].

7.4.2 Résultats de synthèse

La figure 7.20 donne une vue du *floorplan* du processeur reconfigurable synthétisé sur le Virtex-5 de la carte par l'environnement Xilinx ISE. Les deux composants principaux sont le Microblaze et une matrice CGRA de taille 4×5. Les autres parties correspondent aux interfaces du Microblaze (UART, bus, FSL, etc.). L'adaptateur n'est pas visualisable de manière isolée car il est inclu dans le placement-routage de CGRA.

FIG. 7.20 – Vue du *floorplan* du processeur reconfigurable synthétisé sur le Virtex-5 [85]. La matrice CGRA de dimensions 4×5 est couplée à un processeur Microblaze. Les communications entre les deux composants sont réalisées par un adaptateur connecté par FSL.

Le tableau 7.11 donne les résultats de synthèse du processeur reconfigurable (Microblaze, adaptateur et unité reconfigurable) pour les trois matrices CGRA évaluées dans la section précédente : 10×5, 4×5 et 2×5. Ils fournissent au concepteur une métrique supplémentaire pour déterminer le meilleur compromis ressources/performances pour la virtualisation des étages de la matrice.

Le plan de calcul utilise principalement des LUT pour réaliser les UAL. La diminution de la taille de la matrice et, de ce fait, du nombre d'UAL, amène une réduction du nombre de LUT allouées. Cela est également valable pour les registres utilisés par ce plan. Les gains se situent au niveau des registres placés aux entrées des UAL (voir figure 5.15) et ceux utilisés par le contrôleur de configuration. En revanche, le nombre de registres alloués pour le plan de configuration reste constant car sa capacité de stockage ne varie pas.

Taille de CGRA	LUT	Registres	Slices	Occupation	Fréquence (MHz)
10×5	6954	3679	2328	13, 5%	96,9
4×5	3654	2207	1197	6,9%	96,9
2×5	1996	1706	751	4,3%	96,9

TAB. 7.11 – Résultats de synthèse pour trois tailles de matrice CGRA.

L'occupation globale est exprimée en pourcentage de *slices* alloués dans le Virtex-5. Les occupations des matrices 4×5 et 2×5 sont de 2 et 3 fois inférieures à celle de la matrice 10×5. Ces résultats montrent l'intérêt de la virtualisation dans le cas d'un nombre de ressources limité. Cependant, ces gains doivent être pondérés par la dégradation des performances qui résulte du multiplexage temporel des zones reconfigurables.

Les fréquences obtenues permettent l'évaluation des latences d'exécution des matrices CGRA dans le matériel. Les résultats représentent une estimation plus précise en comparaison des latences exprimées en nombre de cycles (tableau 7.9). La figure 7.21 donne les latences d'exécution (en microsecondes) du filtre FIR -8 sur chaque matrice en fonction des tailles de x. Les évaluations ne prennent pas en compte la gestion des données (pré-chargement, réorganisation, etc.) en entrée de la matrice et sont fondées sur les résultats du tableau 7.9. Cependant, la prise en compte de ce paramètre est indispensable pour une comparaison réaliste des matrices et constitue un travail en cours.

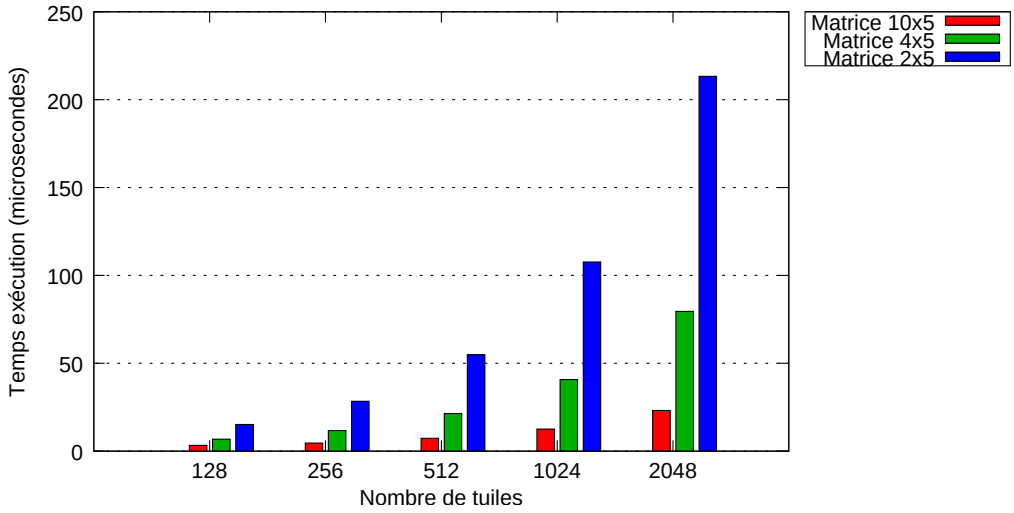


FIG. 7.21 – Temps d'exécution (en microsecondes) du filtre FIR sur les différentes matrices avec la prise en compte de la fréquence de fonctionnement.

7.5 Simulation multi-niveaux d'une application

Pour permettre l'analyse des résultats d'une simulation le concepteur doit bénéficier d'outils de visualisation des activités concurrentes des composants. Les activités système de la plate-forme d'exécution sont visualisées par un diagramme de Gantt. A cela s'ajoutent les communications inter-composants représentées par un diagramme d'interactions. L'activité de

l'application à un niveau comportemental (code et HL-CDFG) est incluse dans le diagramme de Gantt des activités système. En revanche, une simulation à un niveau RTL apparaîtra comme une tâche atomique au niveau système. Les détails de la simulation sont fournis par la production de chronogramme qui tracent, au cycle d'horloge près, l'activité des signaux liés à des sondes.

Pour illustrer l'utilisation de ces outils nous considérons une plate-forme d'exécution de type SoC reconfigurable similaire à celui présenté par la figure 5.3. L'application est structurée suivant le modèle d'exécution décrit en section 5.1.2. Le processus de calcul est un filtre FIR défini par le DFG de la figure 5.13. Il reçoit trois données depuis son processus de communication d'entrée puis produit un résultat envoyé à son processus de communication de sortie.

7.5.1 Simulation des activités systèmes

Les activités systèmes sont reportées sur un diagramme de Gantt donné par la figure 7.22. Les noms des tâches sont placés sur l'axe des ordonnées et l'échelle de temps est placée sur l'axe des abscisses.

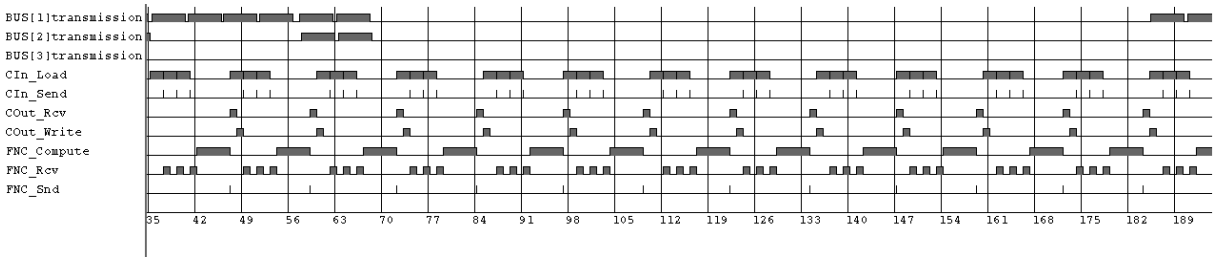


FIG. 7.22 – Diagramme de Gantt qui représente les activités systèmes de la plate-forme d'exécution [121]. Les traces sont créées par l'insertion dans le code comportemental d'appels à l'instance du *SystemLogger*.

Le diagramme représente les activité du bus d'interconnexion et de l'application modélisée à un niveau comportemental par trois processus décrits en code Smalltalk.

En terme de scénario d'exécution le processeur initialise le contrôleur DMA et active l'unité reconfigurable pour une exécution. Le contrôleur DMA exécute les accès mémoire puis effectue des demandes de transferts sur le bus.

Dans la figure 7.22 la trace préfixée par *BUS* correspond aux transferts de données entre la mémoire principale et les mémoires locales de l'unité reconfigurable. Les activités des processus de communication telles que les accès mémoire et le calcul des adresses d'accès sont préfixées par *CIn* et *COut*. Les activité du processus de calcul (réception, calcul, envoi du résultat) sont préfixées par *FNC*. Le diagramme est principalement centré sur les activités de l'unité reconfigurable, l'activité des autres éléments du système est également visualisable sous réserve d'insérer dans leurs comportements les fonctions de traçage.

7.5.2 Interaction des composants

Les systèmes modélisés sont structurés par des composants interconnectés par des canaux de communications. L'augmentation du nombre de composants entraîne une augmentation

des interactions possibles. Pour faciliter l'analyse du système un diagramme d'interactions est généré. Il représente les communications entre les composants à travers leurs ports d'entrées-sorties. La figure 7.23 illustre le diagramme d'interactions pour notre cas d'étude.

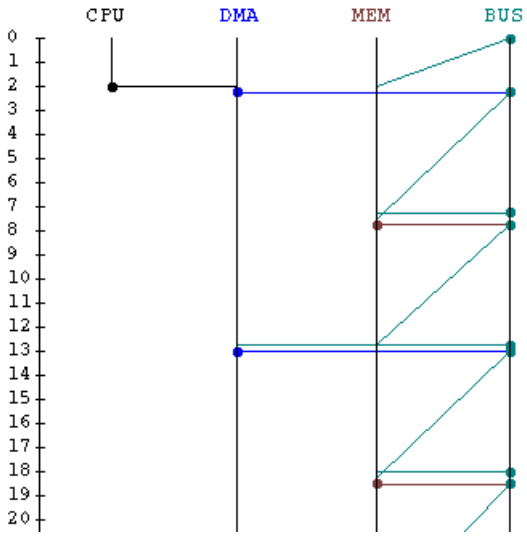


FIG. 7.23 – Diagramme d'interactions généré par le simulateur [121].

Chaque ligne verticale correspond à un composant. Une communication est représentée par un lien entre deux lignes verticales. Le point de départ de la communication est représenté par un rond.

7.5.3 Simulation RTL de l'application

La figure 7.24 montre une liste de chronogrammes générés à partir des valeurs tracées par des sondes posées dans le LL-CDFG.

Les chronogrammes représentent respectivement : (1) les valeurs lues en mémoire par le processus de communication ; (2) (3) (4) les valeurs reçues par les trois opérations de réception sur le canal du processus de calcul qui applique le filtre FIR ; (5) le résultat du calcul envoyé au processus de communication de sortie ; (6) les adresses d'écriture en mémoire et (7) l'activation de l'opération d'écriture en mémoire. Les chronogrammes de (1) à (6) représentent des valeurs numériques tandis que le (7) correspond à une valeur binaire.

La comparaison des résultats des figures 7.22 et 7.24 offre une vision multi-niveaux de l'exécution de l'application. La figure 7.22 donne une vue globale du système simulé et principalement l'ordonnancement des activités. La vision à un niveau RTL fournit des informations supplémentaires sur les valeurs consommées et produites par les processus.

Simulation RTL par des outils tiers. La génération automatique de bancs de test Verilog a pour objectif d'interfacer notre environnement de validation avec des outils standards. Ainsi, notre méthodologie est enrichie par les forces d'outils issus de l'industrie et permet une intégration facilitée dans les flots de validation préexistants.

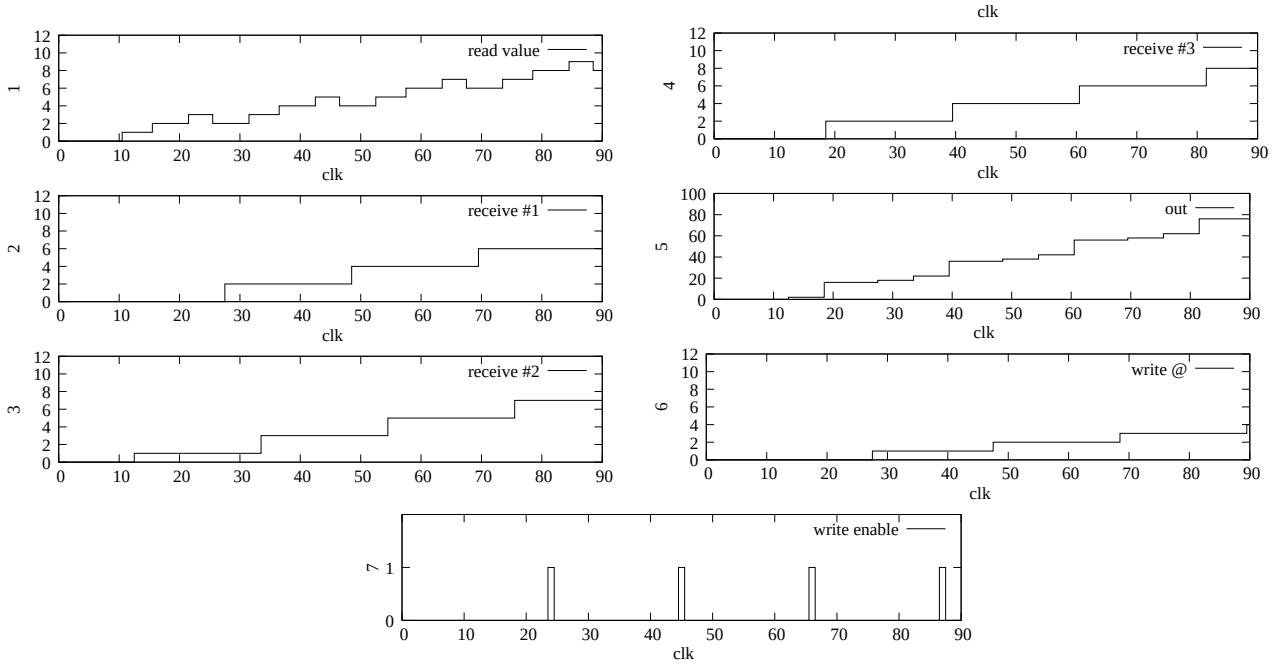


FIG. 7.24 – Chronogramme des signaux sondés dans la simulation RTL [121].

Le banc de test et la *netlist* produite par Madeo+ sont pris en entrée du simulateur RTL tiers, dans notre cas ModelSim. La figure 7.25 montre les chronogrammes des signaux de l'application.

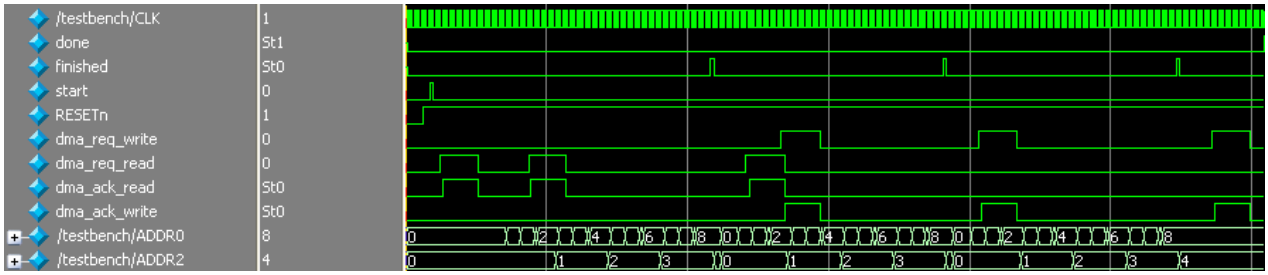


FIG. 7.25 – Simulation de l'application et de son banc de test dans ModelSim.

Les activités du contrôleur DMA (signaux *dma_**) sont le résultat des interactions entre la netlist et le banc de test (voir code 6.3). Ces signaux sont disponibles dans les figures 7.22 et 7.24. On retrouve également les résultats du calcul des adresses d'écriture et de lecture reportés dans le chronogramme 7.24. Ces calculs sont visibles dans le diagramme 7.22 à un niveau d'abstraction supérieur sans le détail des valeurs.

7.6 Conclusion

Dans les chapitres précédents nous avons présenté et validé séparément les trois axes de notre méthode d'exploration architecturale d'unités reconfigurables. Ils couvrent respective-

ment le prototypage de l'unité reconfigurable par modélisation et génération automatique, sa programmation à haut-niveau et la validation fonctionnelle multi-niveaux des applications exécutées sur un SoC. Dans ce chapitre nous avons validé le couplage des trois axes avec le traitement de cas d'étude par l'ensemble des outils.

Les évaluations ont été principalement effectuées sur une unité reconfigurable à gros-grain dont la structure est inspirée par PiCoGA [98] et PipeRench [50]. Ses plans de calcul et de configuration ont été modélisés à haut-niveau pour permettre l'exploration et la génération d'un prototype matériel simulable. Pour illustrer l'exploration des modes de configuration dynamique partielle et multi-contextes, un modèle d'exécution fondé sur la virtualisation des ressources a été mis en œuvre. Les outils ont permis une programmation rapide de l'architecture et l'analyse des performances de l'exécution virtualisée d'un filtre FIR-8.

L'unité a également été mise en œuvre sur une carte FPGA (à base d'un Xilinx Virtex-5) par la synthèse du code généré. Elle est couplée à un processeur Microblaze qui l'utilise comme accélérateur reconfigurable. Ce prototype constitue une première réalisation matérielle des architectures générées par l'outil. Les résultats de synthèse fournissent ainsi une analyse plus fine des ressources nécessaires et des performances d'exécution.

La validation et l'analyse des applications à un niveau système ont été abordées par la simulation d'un filtre FIR sur un modèle de RSoC. La plate-forme d'exécution est modélisée dans le cadriciel à composants SmallSystem qui bénéficie de la dynamicité du langage Smalltalk. L'application est simulée à plusieurs niveaux d'abstraction allant du niveau comportemental au RTL après synthèse de la *netlist*. La génération automatisée de Verilog a également permis une simulation dans des outils tiers du commerce tels que ModelSim [111].

En conclusion, le traitement dans sa globalité d'un cas d'étude significatif a permis de démontrer la synergie des axes de notre solution pour l'exploration d'unités reconfigurables. Un autre bénéfice apportée par cette étude est la mise à disposition d'un guide d'utilisation des outils pour les concepteurs.

Chapitre 8

Conclusion et perspectives

8.1 Synthèse des travaux

Les travaux présentés dans ce manuscrit ont porté sur la définition de méthodes et d'outils pour le prototypage rapide d'unités reconfigurables embarquées. Cela s'est traduit par le développement d'un flot outillé qui automatise partiellement le prototypage matériel de la cible, la programmation à haut-niveau et la validation fonctionnelle des applications. Ainsi, l'allégement de la tâche du concepteur facilite l'exploration du domaine de conception des unités reconfigurables embarquées.

Le flot proposé se décompose en trois parties principales qui adressent les différentes étapes de prototypage d'une unité reconfigurable :

Prototypage rapide d'unités reconfigurables embarquées. L'exploration de l'espace de conception par prototypage rapide d'architectures reconfigurables dynamiquement est réalisée dans l'outil Madeo/DRAGE. Il résulte du couplage entre Madeo-FET [81] et l'outil DRAGE que nous avons développé. Nous avons développé DRAGE pour la génération de prototypes matériels simulables à partir d'une modélisation à haut-niveau de l'architecture. Celle-ci est composée de deux parties principales qui sont la description du plan de calcul et celle du plan de configuration. La première spécifie les ressources de l'architecture et leurs interconnexions dans le langage de description d'architectures Madeo-ADL. La seconde est un découpage du plan en zones reconfigurables dynamiquement et multi-contextes dans une syntaxe XML. La modélisation séparée des deux plans permet une exploration des types de ressources de calcul indépendamment des capacités de reconfiguration dynamique de l'architecture. Le modèle du plan de configuration permet d'estimer pour chaque zone reconfigurable la taille du *bitstream*, le nombre de registres nécessaires et les temps de configuration en fonction du débit.

A partir des deux plans modélisés l'architecture est générée en VHDL comportemental pour être simulée. Un contrôleur de configuration microprogrammable est spécialisé en fonction du plan de configuration. Il est programmé à haut-niveau via une interface graphique qui facilite l'ordonnancement des *bitstreams*. Ceux-ci sont produits à partir du placement-routage d'une application sur le modèle Madeo-BET de l'architecture et d'un modèle de *bitstream* conforme au plan de calcul.

Programmation à haut-niveau des unités. Le prototypage d'unités reconfigurables implique une étape de programmation pour évaluer l'adéquation avec le domaine applicatif

et les performances de l'architecture. Pour minimiser la charge de cette tâche, nous proposons un flot de programmation haut-niveau pour les unités reconfigurables d'un SoC. Le modèle d'exécution considéré est un accélérateur reconfigurable alimenté en données par des mémoires locales. La spécification haut-niveau est indépendante de la nature de la matrice reconfigurable qui est prise en charge automatiquement par Madeo.

L'application est spécifiée dans le langage de composition Avel sous forme d'un réseau de processus communicants fondé sur le modèle de calcul CSP [66]. Une partie des processus est dédiée aux communications et l'autre aux calculs sur les données. Les processus de communication sont assimilables à des générateurs d'adresses et accèdent aux mémoires locales de l'unité. Les processus de calcul sont spécialisés pour le flot de données et sont spécifiés dans un sous-ensemble de C. L'ensemble des processus sont interconnectés et composés hiérarchiquement pour former le réseau.

La description complète est prise en entrée du synthétiseur Madeo+ pour cibler des architectures de différentes natures (granularité, couplage, etc.), préexistantes ou prototypées avec Madeo/DRAGE. Ainsi, trois architectures sont considérées pour montrer la portabilité d'une description Avel : processeur reconfigurable, unité à grain-fin et à gros-grain.

Validation fonctionnelle des applications. La validation fonctionnelle des applications exécutées sur les unités reconfigurables est obtenue par simulation à un niveau système. La validation est fondée sur des cycles itératifs courts inspirés par les méthodes de développement logiciel telles que l'*eXtremeProgramming* [156].

La plate-forme d'exécution est modélisée dans le cadriciel à composant SmallSystem [121]. Pour la simulation, l'application est intégrée dans le modèle avec la prise en compte des activités systèmes. Deux niveaux de simulation de l'application sont disponibles : comportementale et RTL. Les résultats de simulation sont la visualisation des activités systèmes par la production de diagrammes de Gantt et d'interactions entre les composants. L'exécution de l'application synthétisée à un niveau RTL est détaillée par des chronogrammes qui tracent les valeurs des signaux.

La simulation est interfacée avec des outils tiers du commerce (e.g. ModelSim) par la génération d'un banc de test Verilog. Pour cela, les stimuli des activités systèmes sont capturés dans un modèle de simulation qui fait office d'objet *mock*. Cette technique est classiquement utilisée dans la conception orientée objet de logiciel pour valider le comportement des objets. L'objet *mock* constitue un modèle de simulation qui est exporté sous forme d'un banc de test.

La simulation permet la vérification des résultats produits par l'application. Si les tests échouent, l'application doit être déboguée. Pour cela nous avons proposée une transposition du débogage du logiciel vers la synthèse de circuit. Des sondes sont directement insérées dans le code comportemental de l'application puis synthétisées pour obtenir une exécution performante. Le comportement d'un débogueur logiciel est restitué par l'ajout d'un contrôleur spécifique qui met à disposition des fonctionnalités équivalentes. Ainsi, la productivité des méthodes de conception logicielle est exploitée pour la validation fonctionnelle d'applications pour RSoC.

Les différentes parties du flot ont été validées par le prototypage et la programmation d'une architecture reconfigurable à gros-grain organisée en étages pipelinés d'opérateurs de type UAL. Ses plans de calcul et de configuration ont été modélisés à haut-niveau dans Madeo-BET pour permettre l'exploration et la génération par DRAGE d'un prototype matériel simulable.

Pour démontrer les capacités d'exploration des modes de configuration dynamique partielle et multi-contextes, un modèle d'exécution fondé sur la virtualisation des ressources a été mis en œuvre. Les outils ont permis une programmation rapide de l'architecture et l'analyse des performances de l'exécution virtualisée d'un filtre FIR-8.

L'unité a également été mise en œuvre sur une carte FPGA [148] (à base d'un Xilinx Virtex-5 [154]) par la synthèse du code généré. Elle est couplée à un processeur Microblaze qui l'utilise comme accélérateur reconfigurable. Ce prototype constitue une première réalisation matérielle des architectures générées par l'outil. Les résultats de synthèse fournissent ainsi une analyse plus fine des ressources nécessaires et des performances d'exécution.

La validation et l'analyse des applications à un niveau système ont été abordées par la simulation d'un filtre FIR sur un modèle de RSoC. La plate-forme d'exécution est modélisée dans le cadriciel à composants SmallSystem qui bénéficie de la dynamicité du langage Smalltalk. L'application est simulée à plusieurs niveaux d'abstraction allant du niveau comportemental au RTL après synthèse de la *netlist*. La génération automatisée de bancs de test a également permis une simulation dans des outils tiers du commerce tel que ModelSim [111].

8.2 Perspectives

Le domaine des architectures reconfigurables pour les systèmes embarqués est vaste et en perpétuelle évolution. Les méthodes et outils développés dans ces travaux s'abstraient des évolutions technologiques par un positionnement à un niveau d'abstraction élevé. Ils offrent un moyen d'anticiper les besoins futurs en terme d'architecture et facilitent la prospection. De ce fait, ces travaux constituent une ouverture vers un large domaine de recherche avec de nombreuses perspectives.

L'outil DRAGE, présenté dans le chapitre 4, permet la génération VHDL d'architectures reconfigurables synthétisables sur carte FPGA et donc utilisables comme supports d'exécution. Cependant, nous sommes conscient que les prototypes produits ne satisfont pas les exigences de résultats imposées par les industriels. Notre objectif n'est pas d'entrer en concurrence mais d'offrir une capacité de validation précoce de concepts architecturaux (e.g. structure pour la virtualisation). De plus, la caractérisation qualitative en amont des architectures permet de dresser le cahier des charges d'une réalisation optimale dans un contexte industriel (avec le développement manuel des parties critiques par des ingénieurs experts). Ainsi, DRAGE présente un réel intérêt pour la mise en œuvre d'architectures expérimentales aux fonctionnalités spécifiques non-disponibles dans les plate-formes du commerce.

Cette direction est actuellement intégrée dans des travaux du laboratoire LabSTICC/AS qui concernent l'utilisation des architectures reconfigurables dans des environnements agressifs (e.g. le domaine spatial [10]). L'objectif principal est l'utilisation des unités générées par DRAGE sous forme de couches virtuelles reconfigurables aux fonctionnalités évoluées. Elles intégreront des primitives matérielles qui assureront la fiabilité par des capacités d'auto-diagnostic et d'auto-reconfiguration. La réponse à ces besoins passe par un enrichissement de l'outil pour la génération d'unités reconfigurables tolérantes aux fautes. Par exemple, les capacités d'auto-reconfiguration ou d'auto-diagnostic impliquent l'accès au plan de configuration par les applications placées-routées. Actuellement, cette fonctionnalité est disponible sur certaines architectures commerciales (e.g. Xilinx Virtex-5 [154]) à travers l'utilisation du port *Internal Configuration Access Port* (ICAP) [153]. Dans le cas de DRAGE, le plan de configuration d'une architecture générée est accédé uniquement par le contrôleur de configuration.

Par conséquent, pour que les applications bénéficient de cet accès, il serait nécessaire d'enrichir la génération par l'ajout d'un module similaire à l'ICAP. Il serait généré et paramétré automatiquement en tant que primitive et servirait de brique de base pour la mise en œuvre de fonctionnalités systèmes cablées telles que la migration de tâche, la préemption, etc.

Au niveau programmation, la gestion de la reconfiguration dynamique à un niveau applicatif impacterait la spécification de l'application en amont. En effet, outre les flots de données et de contrôle exprimés naturellement à haut-niveau (assignations de variables, opérateurs de sélection, etc.), des opérations de reconfiguration seraient à prendre en compte. Une première piste de travail serait l'ajout d'opérateurs de configuration dans le flot de programmation présenté en chapitre 5. Ils auraient une sémantique équivalente aux microinstructions du contrôleur de configuration présenté dans la section 4.3.3. Ainsi, trois types de flots seraient exprimés dans l'application : données, contrôle et reconfiguration pour une prise en compte explicite de la nature reconfigurable du support d'exécution. Cette approche apporterait aux applications une faculté d'adaptation supérieure à un ordonnancement statique tel que celui décrit en chapitre 4.

Bibliographie

[1] Berkeley logic interchange format (blif), 1996.

[2] J. D. Ullman A. V. Aho, R. Sethi. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] "m2000 flexeostm configurable ip core. http ://www.m2000.com.

[4] Gul Agha. *Actors : a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] E. Ahmed and J. Rose. The effect of lut and cluster size on deep-submicron fpga tperformance and density. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'00)*, pages 3–12. ACM, 2000.

[6] S. R. Alpert, K. Brown, and B. Woolf. *The Design Patterns SmallTalk Companion*. Addison-Wesley Professional, 1998.

[7] Altera. Datasheet, signaltap embedded logic analyzer megafunction. Technical report, 2001.

[8] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proc. of the Symposium on Parallel Algorithms and Architectures (SPAA'92)*, pages 316–322. ACM, 1992.

[9] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26 :11–18, 1993.

[10] L. Barradon, T. Capitaine, L. Lagadec, N. Julien, C. Moy, and T. Monédière. Virtual sopc rad-hardening for satellite applications. In *Proc. of the International Workshop on Reconfigurable Communication-centric Systems on Chip (ReCoSoC'10)*, Karlsruhe, Germany, 2010.

[11] V. Baumgarte, G. Ehlers, F. May, A. Nœckel, M. Vorbach, and M. Weinhardt. Pact xpp–a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2) :167–184, 2003.

[12] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Embedded floating-point units in fpgas. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'06)*, pages 12–20. ACM, 2006.

[13] J. Becker and M. Vorbach. Coarse-grain reconfigurable xpp devices for adaptive high-end mobile video-processing. In *Proc. of the International Conference on System-on-Chip (SoC'04)*, pages 165–166. IEEE, 2004.

[14] A. Bernstein, M. Burton, and F. Ghenassia. How to bridge the abstraction gap in system level modeling and design. In *Proc. of the International Conference on Computer-Aided Design (ICCAD'04)*, Washington, DC, USA, 2004. IEEE.

[15] V. Betz and J. Rose. Cluster-based logic blocks for fpgas : area-efficiency vs. input sharing and size. In *Proc. of the Custom Integrated Circuits Conference (CICC'97)*, pages 551–554. IEEE, 1997.

[16] V. Betz and J. Rose. Vpr : A new packing, placement and routing tool for fpga research. In *Proc. of the International Conference on Field-Programmable Logic and Applications (FPL'97)*, pages 213–222, 1997.

[17] V. Betz and J. Rose. Fpga routing architecture : segmentation and buffering to optimize speed and density. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'99)*, pages 59–68. ACM, 1999.

[18] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[19] Vaughn Betz and Jonathan Rose. How much logic should go in an fpga logic block ? *IEEE Design & Test*, 15(1) :10–15, 1998.

[20] Ivo Bolsens. Challenges and opportunities for fpga platforms. In *Proc. of the International Conference on Field-Programmable Logic and Applications (FPL'02)*, pages 391–392. Springer-Verlag, 2002.

[21] L Bossuet, G. Gogniat, and J.-L. Philippe. Fast design space exploration method for reconfigurable architectures. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*, pages 23–26, 2003.

[22] K. Bouazza, J. Champeau, P. Ng, B. Pottier, and S. Rubini. Implementing cellular automata on the armen machine. In *Proc. of the international workshop on Algorithms and parallel VLSI architectures II*, pages 317–322. Elsevier Science Publishers B. V., 1992.

[23] P. Boulet, J.-L. Dekeyser, J.-L. Levaire, J. Soula, and A. Demeure. Visual data-parallel programming for signal processing applications. In *Proc of the Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, pages 105–112, 2001.

[24] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *Proc. of the International Workshop on Applied Reconfigurable Computing (ARC'07)*, pages 1–13. Springer, 2007.

[25] M. Bowen. Handel-c language reference manual. Technical report, Embedded Solutions.

[26] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4) :62–69, 2000.

[27] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, P. Rolandi, C. Mucci, A. Lodi, A. Vitkovski, and L. Vanzolini. A dynamically adaptive dsp for heterogeneous reconfigurable platforms. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'07)*, 2007.

[28] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). In *Proc. of the International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 605–614. Springer-Verlag, 2000.

[29] A. Chattopadhyay, R. Leupers, H. Meyr, and G. Ascheid. *Language-driven Exploration and Implementation of Partially Re-configurable ASIPs*. Springer Publishing Company, Incorporated, 2008.

[30] Juanjuan Chen, Xing Wei, Qiang Zhou, and Yici Cai. Power optimization through edge reduction in lut-based fpga technology mapping. In *Proc. of the International Conference on Communications, Circuits and Systems (ICCCAS'09)*, pages 1087–1091. IEEE, 2009.

[31] P. C. Clements. A survey of architecture description languages. In *Proc. of the International Workshop on Software Specification and Design (IWSSD'96)*, page 16. IEEE, 1996.

[32] Target The ASIP Company. Target compiler technologies, http ://retarget.com/.

[33] K. Compton and S. Hauck. Reconfigurable computing : a survey of systems and software. *ACM Computing Surveys*, 34(2) :171–210, 2002.

[34] Jason Cong and Yuzheng Ding. Flowmap : An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 13 :1–12, 1994.

[35] Inmos Corporation. Occam 2 reference manual. *Prentice Hall*, 1988.

[36] A. DeHon. Dpga utilization and application. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'96)*, pages 115–121. ACM, 1996.

[37] A. DeHon. Dynamically programmable gate arrays : A step toward increased computational density. In *Proc. of the Canadian Workshop of Field-Programmable Devices (FPD'96)*, pages 47–54, 1996.

[38] A. DeHon. *Reconfigurable architectures for general-purpose computing.* PhD thesis, 1996.

[39] A. DeHon. The density advantage of configurable computing. *Computer*, 33(4) :41–49, 2000.

[40] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-ol : proposition d'un formalisme tableau pour le traitement de signal multidimensionnel. In *GRETSI, Groupe d'Etudes du Traitement du Signal et des Images*, 1995.

[41] C. Ebeling, D. C. Cronquist, and P. Franklin. Rapid - reconfigurable pipelined datapath. In *Proc. of the International Workshop on Field-Programmable Logic (FPL'96)*, pages 126–135. Springer-Verlag, 1996.

[42] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *Computer*, 37(1) :47–56, 2004.

[43] A. Fauth. *Beyond Tool-Specific Machine Descriptions.* Kluwer Academic Publishers, 1995.

[44] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *Proc. of the European Conference on Design and Test (EDTC'95)*, page 503. IEEE, 1995.

[45] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, pages 327–335. ACM, 1989.

[46] Robert Francis, Jonathan Rose, and Zvonko Vranesic. Chortle-crf : Fast technology mapping for lookup table-based fpgas. In *Proc. of the Design Automation Conference (DAC'91)*, pages 227–233. ACM, 1991.

[47] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. The specc methodology. Technical report, University of California, Irvine, 1999.

[48] C. Glitia, P. Dumont, and P. Boulet. Array-ol with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 21(2) :105–131, 2010.

[49] A. Goldberg and D. Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley, Boston, MA, USA, 1983.

[50] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench : a reconfigurable architecture and compiler. *Computer*, 33(4) :70–77, 2000.

[51] G. Goossens, D. Lanneer, W. Geurts, and J. Van Praet. Design of asips in multiprocessor socs using the chess/checkers retargetable tool suite. In *Proc. of the International Symposium on System-on-Chip (SoC'06)*, pages 1–4, 2006.

[52] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging fpga circuits. In *Proc. of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 41–50, 2001.

[53] Graphviz. http ://www.graphviz.org/.

[54] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic, 2004.

[55] A. Habibi and S. Tahar. Design for verification of systemc transaction level models. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'05)*, pages 560–565. IEEE, 2005.

[56] R. Hartenstein. The microprocessor is no longer general purpose : why future reconfigurable platforms will win. In *Proc. of the International Conference on Innovative Systems in Silicon (ISIS'97)*, pages 2–12. IEEE, 1997.

[57] R. Hartenstein. A decade of reconfigurable computing : a visionary retrospective. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'01)*, pages 642–649. IEEE, 2001.

[58] R. Hartenstein. Why we need reconfigurable computing education. In *Opening Session of the 1st International Workshop on Reconfigurable Computing Education (RCE'06)*, 2006.

[59] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the kress-array for reconfigurable computing. In *Proc. of the Conference on Configurable Computing : Technology and Applications*, pages 150–161. SPIE, 1998.

[60] R. W. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC'95)*, page 77. ACM, 1995.

[61] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. In *Proc. of the 5th Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*, page 87. IEEE, 1997.

[62] J. R. Hauser and J. Wawrzynek. Garp : a mips processor with a reconfigurable coprocessor. In *Proc. of the Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*, page 12. IEEE, 1997.

[63] R. Helaihel and K. Olukotun. Java as a specification language for hardware-software systems. In *Proc. of the International Conference on Computer-Aided Design (ICCAD'97)*, pages 690–697. IEEE, 1997.

[64] J. Henkel. Closing the soc design gap. *Computer*, 36(9) :119–121, 2003.

[65] J. L. Hennessy and D. A. Patterson. *Computer Architecture ; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1992.

[66] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[67] M. Hübner, C. Schuck, M. Kuhnle, and J. Becker. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits. In *Proc. of the Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, page 97. IEEE, 2006.

[68] Ieee. *Std 1076-2000 : IEEE Standard VHDL Language Reference Manual*. IEEE, 2000.

[69] Open SystemC Initiative. http ://www.systemc.org/.

[70] A. Iqbal and B. Pottier. Meta-simulation of large wsn on multi-core computers. In *Proc. of the Spring Simulation Multi-Conference (SpringSim'10)*, Orlando, USA, 2010.

[71] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'99)*, pages 145–154. ACM, 1999.

[72] Stephen Jang, Billy Chan, Kevin Chung, and Alan Mishchenko. Wiremap : Fpga technology mapping for improved routability and enhanced lut merging. *ACM Transactions on Reconfigurable Technology and Systems*, 2 :14 :1–14 :24, June 2009.

[73] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, 1988.

[74] G. Kahn. The semantics of simple language for parallel programming. In *Proc. of the IFIP Congress*, pages 471–475, 1974.

[75] B. Kastrup, A. Bink, and J. Hoogerbrugge. Concise : A compiler-driven cpld-based instruction set accelerator. In *Proc. of the Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, page 92. IEEE, 1999.

[76] G. Kremer, J. Osmont, and B. Pottier. A process oriented development flow for wireless sensor networks. In *Proc. of the International Workshop on Smalltalk Technologies (IWST'09)*, Brest, France, 2009.

[77] I. Kuon. Automated fpga design, verification and layout. Technical report, Master's thesis, University of Toronto, 2004.

[78] I. Kuon, A. Egier, and J. Rose. Design, layout and verification of an fpga using automated tools. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'05)*, pages 215–226. ACM, 2005.

[79] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'06)*, pages 21–30. ACM, 2006.

[80] A. Kupriyanov, F. Hannig, D. Kissler, J. Teich, J. Lallet, O. Sentieys, and S. Pillement. Modeling of interconnection networks in massively parallel processor architectures. In *Proc. of the International Conference on Architecture of Computing Systems (ARCS'07)*, pages 268–282. Springer-Verlag, 2007.

[81] L. Lagadec. *Abstraction, modélisation et outils de* CAO *pour les architectures reconfigurables*. PhD thesis, Université de Rennes 1, 2000.

[82] L. Lagadec. Madeo-bet modeling & programming fpga architecture. Technical report, LabSTICC-UBO, 2003.

[83] L. Lagadec and D. Picard. Software-like debugging methodology for reconfigurable platforms. In *Proc. of the International Symposium on Parallel&Distributed Processing (IPDPS'09)*, pages 1–4. IEEE, 2009.

[84] L. Lagadec and D. Picard. Smalltalk debug lives in the matrix. In *Proc. of the International Workshop on Smalltalk Technologies (IWST'10)*, Barcelona, Spain, 2010.

[85] L. Lagadec, D. Picard, Y. Corre, and P.-Y. Lucas. Experiment centric teaching for reconfigurable processors. *International Journal of Reconfigurable Computing (à paraître)*, 2009.

[86] L. Lagadec, D. Picard, and B. Pottier. Spatial design : High-level synthesis. *Dynamic System Reconfiguration in Heterogeneous Platforms - The MORPHEUS Approach*, 40 :165–182, 2009.

[87] L. Lagadec and B. Pottier. A 6200 model and editor based on object technology. In *Proc. of the International Workshop on Field-Programmable Logic and Applications (FPL'98)*, pages 515–519. Springer-Verlag, 1998.

[88] L. Lagadec and B. Pottier. Object-oriented meta tools for reconfigurable architectures. In *Proc. of the Modeling, Signal Processing, and Control Conference*, volume 4693, pages 69–79, 2002.

[89] L. Lagadec, B. Pottier, and O. Villellas-Guillen. A lut-based high level synthesis framework for reconfigurable architectures. In S.S. Batttacharyya, E. Deprettere, and J. Teich, editors, *Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation*, pages 19–39. Marcel Dekker, 2003.

[90] J. Lallet. *Mozaïc : plate-forme générique de modélisation et de conception d'architectures reconfigurables dynamiquement*. PhD thesis, Université de Rennes 1, 2008.

[91] J. Lallet, S. Pillement, and O. Sentieys. Efficient and Flexible Dynamic Reconfiguration for Multi-Context Architectures. *Journal of Integrated Circuits and Systems*, 4 :36–44, 2009.

[92] J. Lallet, S. Pillement, and O. Sentieys. xmaml : A modeling language for dynamically reconfigurable architectures. In *Proc. of the Euromicro Symposium on Digital Systems Design*, pages 680–687. IEEE, 2009.

[93] Y. Le Moullec, J.-P. Diguet, T. Gourdeaux, and J.-L. Philippe. Design-trotter : System-level dynamic estimation task a first step towards platform architecture selection. *Journal of Embedded Computing*, 1(4) :565–586, 2005.

[94] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proc. of the IEEE*, volume 75, pages 1235–1245, 1987.

[95] G. G. Lemieux and S. D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *Proc. of the Physical Design Workshop (PDW'93)*, pages 215–226. ACM, 1993.

[96] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor description. *Design Automation for Embedded Systems*, 3 :75–108, 1998.

[97] SGS-THOMSON Microelectronics Limited. *OCCAM 2.1 reference manual*. 1995.

[98] A. Lodi, M. Toma, and F. Campi. A pipelined configurable gate array for embedded processors. In *Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA'03)*, pages 21–30. ACM, 2003.

[99] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. A vliw processor with reconfigurable instruction set for embedded applications. *IEEE Journal of Solid-State Circuits*, 38(11) :1876–1886, 2003.

[100] PA Lucent Technologies Allentown. Orca series 4 field-programmable gate arrays. Technical report, 2000.

[101] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[102] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, and H.A.E. Spaanenburg. Seeking solutions in configurable computing. *Computer*, 30(12) :38–43, 1997.

[103] G. Martin, R. Seepold, T. Zhang, L. Benini, and G. De Micheli. Component selection and matching for ip-based design. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'01)*, pages 40–46. IEEE, 2001.

[104] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Adres : An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. *Field-Programmable Logic and Applications*, 2778 :61–70, 2003.

[105] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc : A retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. of the International Conference on Field-Programmable Technology (FPT'02)*, pages 166–173. IEEE, 2002.

[106] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[107] P. Mishra and N. Dutt. Modeling and validation of pipeline specifications. *ACM Transactions in Embedded Computing Systems*, 3(1) :114–139, 2004.

[108] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *IEE Proceedings - Computers and Digital Techniques*, 152(3) :285–297, 2005.

[109] P. Mishra, A. Shrivastava, and N. Dutt. Architecture description language (adl)-driven software toolkit generation for architectural exploration of programmable socs. *ACM Transactions on Design Automation of Electronic Systems*, 11(3) :626–658, 2006.

[110] H. Miyazaki. Implementation and evaluation of the compiler for wasmii, a virtual hardware system. In *Proc. of the International Workshops on Parallel Processing (ICPP'99)*, page 346. IEEE, 1999.

[111] Modelsim. http ://www.model.com/.

[112] G. E. Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, pages 56–59, 2000.

[113] C. Mucci, F. Campi, C. Chiesa, A. Lodi, and M. Toma. How to program with griffy. Technical report, University of Bologna - ARCES, 2003.

[114] C. Mucci, C. Chiesa, A. Lodi, M. Toma, and F. Campi. A c-based algorithm development flow for a reconfigurable processor architecture. In *Proc. of the International Symposium on System on Chip (SoC'03)*, pages 69–73. IEEE, 2003.

[115] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[116] A. Orailoglu and D. D. Gajski. Flow graph representation. In *Proc. of the Conference on Design Automation (DAC'86)*, pages 503–509. IEEE, 1986.

[117] C. Patterson, P. Athanas, M. Shelburne, J. Bowen, J. Surís, T. Dunham, and J. Rice. Slotless module-based reconfiguration of embedded fpgas. *ACM Transactions in Embedded Computing Systems*, 9(1) :1–26, 2009.

[118] D. Patterson. A call to arms : A new manifesto for systems. In *CRA Conference on "Grand Research Challenges" in Computer Science and Engineering*, 2002.

[119] K. Paulsson, U. Viereck, M. Hübner, and J. Becker. Exploitation of the external jtag interface for internally controlled configuration readback and self-reconfiguration of spartan 3 fpgas. In *Proc. of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'08)*, pages 304–309. IEEE, 2008.

[120] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Lisa—machine description language for cycle-accurate models of programmable dsp architectures. In *Proc. of the Design Automation Conference (DAC'99)*, pages 933–938. ACM, 1999.

[121] D. Picard and L. Lagadec. Multi-level simulation of heterogeneous reconfigurable platforms. *International Journal of Reconfigurable Computing*, 2010.

[122] D. Picard, B. Pottier, and C. Teodorov. Process system modeling for rsoc. In *Proc. of the International Workshop on Reconfigurable Communication-centric Systems on Chip (ReCoSoC'08)*, Barcelona, Spain, 2008.

[123] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proc. of the International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pages 213–218. IEEE, 2005.

[124] MORPHEUS Integrated Project. http ://www.morpheus-ist.org/, 2006.

[125] W. Putzke-Röming. Morpheus architecture overview. *Dynamic System Reconfiguration in Heterogeneous Platforms - The MORPHEUS Approach*, 40 :31–37, 2009.

[126] R. Razdan, K. S. Brace, and M. D. Smith. Prisc software acceleration techniques. In *Proc. of the International Conference on Computer Design (ICCD'94)*, pages 145–149. IEEE, 1994.

[127] G. Sassatelli, L. Torres, P. Benoit, T. Gil, C. Diou, G. Cambon, and J. Galy. Highly scalable dynamically reconfigurable systolic ring-architecture for dsp applications. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'02)*, page 553. IEEE, 2002.

[128] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sis : A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[129] G. J. M. Smit, P. M. Heysters, M. A. J. Rosien, and B. Molenkamp. Lessons learned from designing the montium - a coarse-grained reconfigurable processing tile. In *Proc. of the International Symposium on System-on-Chip (SoC'04)*, pages 29–33. IEEE, 2004.

[130] Synopsys. http ://www.synopsys.com/.

[131] Synopsys. System-level design, http ://coware.com/.

[132] Berkeley Logic Synthesis and Verification Group. Abc : A system for sequential synthesis and verification, www.eecs.berkeley.edu/ alanmi/abc.

[133] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor : A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2) :25–35, 2002.

[134] Maxim Teslenko and Elena Dubrova. Hermes : Lut fpga technology mapping algorithm for area minimization with optimum depth. In *Proc. of the International Conference on Computer-Aided Design (ICCAD'04)*, pages 748–751. IEEE, 2004.

[135] D. E. Thomas and P. R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1996.

[136] A. Tiwari and K. A. Tomko. Scan-chain based watch-points for efficient run-time debugging and verification of fpga designs. In *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, pages 705–711. IEEE, 2003.

[137] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for systems-on-chip design. In *Proc. of the Asia Pacific Conference on Chip Design Language*, pages 109–116, 1999.

[138] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *Proc. of the Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*, pages 22–28. IEEE, 1997.

[139] S. Vassiliadis, E. A. Hakkennes, J. S. S. M. Wong, and G. G. Pechanek. The sum-absolute-difference motion estimation accelerator. In *Proc. of the Conference on EUROMICRO (EUROMICRO'98)*, volume 2, pages 559–566. IEEE, 1998.

[140] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, 53(11) :1363–1375, 2004.

[141] Visualworks smalltalk. http ://www.cincom.com.

[142] N Voros, A. Rosti, and M. Hübner, editors. *Dynamic System Reconfiguration in Heterogeneous Platforms - The MORPHEUS Approach*. Springer, 2009.

[143] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable active memories : reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration Systems*, 4(1) :56–69, 1996.

[144] R. Wittig and P. Chow. OneChip : An FPGA processor with reconfigurable logic. In *Proc. of the Symposium on FPGA-Based Custom Computing Machines (FCCM'96)*, pages 126–135. IEEE, 1996.

[145] W. A. Wulf and S. A. McKee. Hitting the memory wall : implications of the obvious. *SIGARCH Computer Architecture News*, 23(1) :20–24, 1995.

[146] Xilinx. Fpga and cpld solutions from xilinx, inc., http ://www.xilinx.com/.

[147] Xilinx. Microblaze soft processor, http ://www.xilinx.com/tools/microblaze.htm.

[148] Xilinx. Xupv5-lx110t, http ://www.xilinx.com/univ/xupv5-lx110t.htm.

[149] Xilinx. Xc4000e and xc4000x series field programmable gate arrays - product specification. Technical report, 1999.

[150] Xilinx. Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel. Technical report, 2004.

[151] Xilinx. Application note xapp138, virtex fpga series configuration and readback. Technical report, 2005.

[152] Xilinx. Ug029, chipscope pro 10.1 software and cores user guide. Technical report, 2008.

[153] Xilinx. Logicore ip xps hwicap. Technical report, 2010.

[154] Xilinx. Virtex-5 fpga user guide (ug190). Technical report, 2010.

[155] Xilinx. Virtex-6 fpga configurable logic block - user guide. Technical report, 2010.

[156] Extreme programming methodology. www.extremeprogramming.org.

Méthodes et outils logiciels pour l'exploration architecturale d'unités reconfigurables embarquées

Résumé L'augmentation continue de la complexité des applications embarquées et leur évolution rapide exigent des systèmes de plus en plus performants et flexibles. Ainsi, la réalisation de systèmes-sur-puce qui intégrent des architectures reconfigurables est aujourd'hui la norme pour répondre aux besoins applicatifs. Cependant, la course à l'innovation des fabricants nécessite une réduction conséquente des délais de mise sur le marché du produit et une forte productivité des concepteurs.

Pour permettre une exploration rapide et efficace du vaste espace de conception des architectures reconfigurables, le concepteur a besoin de méthodes et d'outils pour : (1) évaluer les variantes architecturales suivant différentes métriques (e.g. temps de configuration, nombre de ressources, etc.) ; (2) programmer rapidement et facilement l'unité reconfigurable pour itérer sur plusieurs applications ; (3) valider fonctionnellement une exécution de l'application au niveau système puis par simulation *in situ* après synthèse.

Nos travaux adressent ces trois points par une modélisation à haut-niveau de l'unité reconfigurable qui permet la génération d'un prototype matériel et la mise à disposition de ses outils applicatifs. Ainsi, nos outils offrent aux concepteurs une capacité de validation précoce de concepts architecturaux en amont d'une réalisation optimale ASIC de l'unité.

Mots clés architectures reconfigurables, systèmes-sur-puce, outils CAO, modélisation architecturale, exploration de l'espace de conception, prototypage rapide

Methods and Software Tools for Architectural Exploration of Embedded Reconfigurable Architectures

Abstract In order to cope with the increasing complexity of embedded applications as well as their fast evolution, flexible systems with high-performance are mandatory. In this context, reconfigurable system-on-chip solutions that meet application needs have become common. However, the innovation race shrinks time-to-market and puts high pressures on designers.

To enable a fast and efficient design space exploration of reconfigurable units, designers need appropriate methodologies and tools to : (1) estimate different architectural solutions according to metrics (e.g. configuration latency, ressources, etc.) ; (2) easily program the reconfigurable units in order to iterate on different applications ; (3) validate the behaviour of an execution at system-level and at post-synthesis.

We address these three issues by an high-level modeling of the reconfigurable unit enabling to generate an hardware prototype and its programming tools. Thus, designers are able to perform early validation of architectural concepts before an ASIC implementation.

Keywords reconfigurable architectures, system-on-chip, CAD tools, architectural modeling, design space exploration, fast prototyping