Transformations de langage de haut niveau pour exécution partielle en logique reconfigurable

Sylvain Guérin Ronan Keryell Bernt Weber

Département Informatique, ENST Bretagne
26 avril 2004

Table des matières

1 Introduction

Le but final de notre projet est de traduire du code d'un langage de haut niveau en codes de différent formats, nécessaires pour une architecture cible hétérogène. Cette architecture hétérogène comportera un ou plusieurs processeurs séquentiels et un ou plusieurs noeuds de matériel reconfigurable. Le matériel reconfigurable peut, dans le cadre de notre travail, correspondre par exemple à des FPGA (reconfiguration à grain fin) ou des *datapaths* reconfigurables (grain moyen)[DCPS02].

Le compilateur que nous comptons concevoir utilisera la boite à outils existante PIPS [IJT91] pour le *front end* et certaines transformations de code existantes. PIPS dispose notamment d'un *front end* pour le langage Fortran, un autre pour le C en cours de développement. Comme exemple de transformation de code utile à notre tâche, on peut citer l'atomisation (exécuter un seul opérateur par ligne) et le déroulage de boucles.

Pour réaliser notre nouveau compilateur, il nous faudra donc créer des nouvelles transformations de code spécifique à notre tâche et un ou plusieurs nouveaux *back ends*. Une nouvelle transformation de code séparera par exemple les parties à exécuter sur des noeuds de différent type. En ce qui concerne le *back end*, il nous faudra générer du code SmallTalk pour les outils MADEO [LLFP01] et ou un synthétiseur VHDL commercial.

1.1 Exploration interactive / itérative des solutions

En transformant un code de langage haut niveau en configuration pour du matériel reconfigurable, le développeur dispose de beaucoup de libertés. Il faut prendre par exemple des décisions sur les traitements à exécuter en parallèle ou en séquentiel. Les contraintes imposées par le flux de contrôle et les dépendances des données doivent être respectées. Les outils PIPS nous fournissent les analyses du programme source nécessaire. En partant de là, on pourrait imaginer de compiler / synthétiser tout automatiquement vers un mélange de code binaire et de configurations de ressources reconfigurables. Nous pensons que ceci peut-être un objectif à long terme.

Pour progresser vers l'objectif de la compilation tout automatique, nous proposons une méthodologie interactive. Dans cette logique, notre système met à disposition du développeur des analyses pour explorer les possibles alternatives pendant la compilation / synthèse. Puis, le système automatisera les transformations nécessaires. En utilisant et développant ce système, nous apprendrons progressivement quelles analyses et transformations supplémentaires sont nécessaires et utiles. Progressivement, on pourrait automatiser de plus en plus le processus de compilation en utilisant une programmation *scripting* et ou la méthode des fichier `make` comme PIPS les inclut déjà maintenant.

L'utilisateur travaillera aussi de manière itérative. En partant du code source de langage haut niveau, une itération comportera d'abord les analyses et les transformations pour obtenir deux parties de code source, une pour exécution sur des processeurs séquentiels et une pour configurer des ressources. Le code source pour le reconfigurable sera synthétisé par un autre système de synthèse. Le système de synthèse pourrait annoter en commentaire au code initial de langage de haut niveau certains résultats comme par exemple l'utilisation de ressources ou les délais de propagation. Puis, l'utilisateur pourra commencer la

prochaine itération en profitant des résultats de la première.

2 Travaux similaires

2.1 SpecC

L'article [DGG02] décrit la méthodologie de conception et le flux de développement qui s'articule autour du langage SpecC. Ce langage spécifiquement développé pour les besoins de conception de systèmes matériel/logiciel permet d'exprimer la spécification, toutes les modèles intermédiaires et aussi les modèles d'implémentation du coté *hardware* et *software*.

2.1.1 Flux de développement

Le flux de conception *(design flow)* passe par un modèle de spécification, par un modèle d'architecture et un modèle de communication vers le modèle d'implémentation (voir figure 2 de [DGG02]).

Le **modèle de spécification** définit des blocs fonctionnels arbitrairement enchaînés en série et en parallèle. Les blocs communiquent à travers de variables et se synchronisent par des événements.

Le modèle de spécification utilisé en entrée pour la méthodologie utilise la hiérarchie pour maîtriser la complexité et définit la granularité exploitable pendant le procès de conception. Le modèle initial sépare les calculs de la communication et expose le parallélisme pour le choix des compromis nécessaires.

Pendant l'**exploration architecturale**, on exécute le partitionnement des comportements et variables et on prépare l'ordonnancement. Le **partitionnement des comportements** distribue les comportements sur un nombre de PE *(processing elements)* en ajoutant un nouveau niveau de hiérarchie. Le **partitionnement des variables** alloue de la mémoire partagé aux variables locales ou les distribue sur les PE en spécifiant l'envoi de messages.

Le premier résultat intermédiaire est le **modèle architectural**. Ce modèle consiste de plusieurs PE qui travaillent en parallèle et qui communiquent à travers de canaux de communications abstraits. A l'intérieur de chaque PE, les traitements se font toujours en séquentiel.

La **synthèse de communication** distribue les canaux abstraits sur des bus, génère les protocoles de bus nécessaires et ajoute ces protocoles aux comportements des PE concernés. Les PE ne sont ainsi plus interconnectés que par des lignes de bus au lieu des canaux abstraits qu'il y avait auparavant.

Le résultant **modèle de communication** se compose maintenant de comportements séquentiels incluant les protocoles de bus qui sont associés à un nombre de PE interconnectés par les lignes de bus.

La dernière étape décrite dans le papier est le ***backend***. Au cours de cette étape, s'exécute la synthèse du matériel, du logiciel et des interfaces.

Pendant la **synthèse du matériel** qui est exécutée pour les PE à réaliser en hardware, on effectue une synthèse de haut niveau. Il s'agit de dessiner les limites des cycles d'horloge entre les instructions. Les instructions entre deux limites représentent le travail à effectuer par un chemin de données pendant un cycle d'horloge. Chacune des groupes correspond à un état de l'automate de contrôle.

La **synthèse du logiciel** (pour les PE à réaliser par un processeur séquentiel) consiste en une compilation habituelle. Elle se fait typiquement en deux étapes, d'abord on transforme en code C pour compiler ensuite avec un compilateur existant.

La **synthèse des interfaces** se fait de manière différente s'il s'agit d'une implémentation en matériel ou en logiciel. Pour le matériel, le protocole de bus est réalisé par un automate et un chemin de données associé. Pour le logiciel, on utilise des routines qui incluent par exemple les instructions d'entrée sortie du processeur.

Le modèle résultant, le **modèle d'implémentation**, est une description structurelle de tous les composants réalisés en matériel en plus du code machine qui s'exécutera sur les processeurs séquentiels. Le modèle est une description exacte qui décrit le système entier avec son comportement au cycle d'horloge prêt.

2.1.2 Comparaison avec PHRASE

Le projet SpecC propose une méthodologie très complète et les outils associés pour la conception de systèmes matériel/logiciel. Nous reprenons l'idée d'exprimer le comportement du matériel au cycle d'horloge près dans un langage de haut niveau.

SpecC est un langage conçu pour cette méthodologie. Notre approche se distingue, par le souhait d'utiliser de codes source existants qui représentent des efforts de développement considérables. Ces projets rendent le redéveloppement dans un nouveau langage économiquement difficile.

Nous utilisons les analyses et transformation source à source PIPS. Ceci permet de transformer et optimiser un logiciel, initialement prévu pour une exécution sur des processeurs séquentiels, pour une réalisation dans du matériel reconfigurable.

2.2 SystemC

SystemC est un langage de modélisation de systèmes entiers, composés du matériel et du logiciel associé. SystemC est réalisé en forme de bibliothèque de classes C++. La bibliothèque de classes, la documentation et autres outils utiles sont distribués sous une licence *open-source* sur le site [sys].

En utilisant la bibliothèque de classes, le développeur accède à des types d'objets spécifique au matériel et l'implémentation de l'ordonnancement associé. Le code source se compile avec un compilateur C++ et le résultat devient un simulateur du système modélisé. Ce noyau de simulation utilise le même cycle évaluation / mise à jour avec des cycles delta connu par les HDL *Hardware Description Languages* communs (VHDL et Verilog).

2.2.1 Flux de conception

SystemC peut-être utilisé comme unique langage de modélisation au cours d'un flux de conception qui va d'une description fonctionnelle de très haut niveau jusqu'à la *netlist* du matériel et aux exécutables du logiciel associé (voir figure 2 de [Pan01]). La description fonctionnelle de haut niveau est d'abord

traduit, possiblement de manière manuelle, vers un niveau accessible aux outils de *hardware / software codesign*. Pendant la suivante cosynthèse, se fait la répartition matériel / logiciel et la synthèse du code *glue*. Reste à compiler la partie logicielle et à traiter la partie matérielle avec les méthodes de synthèse comportementale ou architecturale pour obtenir une description de niveau RTL *(Register Transfer Logic)*. En dernière étape, la synthèse logique produit une *netlist* de composants d'une bibliothèque de logique numérique adapté à un FPGA spécifique ou des *standard cells*.

2.2.2 Comparaison avec les HDL communs

Le processus et la modélisation pour une synthèse comportementale ou architecturale est plus ou moins identique à celle des HDL classiques. La bibliothèque de classes dispose de types pour la modélisation du matériel (types de valeurs logiques et de données du matériel, vecteurs, des processus, la représentation du temps et des horloges, des modules, des interfaces et des ports). Comme dans les HDL communs, il faut respecter un style ou sous-ensemble synthétisable.

Pour une modélisation de plus haut niveau que possible avec les HDL communs, SystemC dispose des classes suivantes inspiré par SpecC pour représenter :

- les événements
- les canaux de communication

SystemC donne plus de facilité pour réaliser des bancs de test par l'accès à l'ensemble du langage C++. Les HDL communs sont assez limités par rapport à :

- l'accès aux fichiers
- le traitement de chaînes de caractères
- des types de données complexes

La réalisation de la partie logicielle entier se fait naturellement en C++. Elle est donc directement compilable et utilisable.

2.2.3 Comparaison avec PHRASE

Comme PHRASE, SystemC utilise un langage de haut niveau standard (ici C++). PHRASE essaye de traiter des codes existants et consistants d'un grand nombre de lignes. SystemC est pensé comme langage de spécification en début du processus de développement. La transformation de logicielles C++ existants en SystemC synthétisable n'est pas prévue et paraît plus difficile suite aux concepts orientés objet.

La nouveauté de nos propositions consiste d'abord dans les analyses (dépendances, parallélisme intrinsèque *etc.*) du code écrit initialement pour une exécution en logiciel. Puis, des transformations automatisées, guidées interactivement par un développeur génèrent du code exécutable sur des machines hétérogènes (processeurs séquentiels plus matériel reconfigurable).

2.3 MATCH

Les articles [BSC+00] et [HNK+] décrivent un compilateur de programmes MATLAB sur une architecture hétérogène qui peut contenir des cartes à multiples FPGA, DSP ou processeurs universels. La machine expérimentale est

construite d'un châssis bus-VME et de diverses cartes COTS *(Commercial Of The Shelf)*.

2.3.1 Compilateur fondé sur des bibliothèques

Le compilateur dispose de bibliothèques de fonctions MATLAB préimplémentées. En plus, comme MATLAB permet de programmer dans un style procédural, le compilateur MATCH sait traduire ces codes en VHDL synthétisable pour un ou plusieurs FPGA ou en code C pour un ou plusieurs DSP. Pour les bibliothèques préimplémentées, le compilateur dispose de données chiffrées sur les ressources et les performances qu'on puisse atteindre sur les différents supports cibles.

Le compilateur permet de repartir les calculs par directives dans le code source ou de manière automatique. Pour repartir les calculs automatiquement, le compilateur reformule le problème d'optimisation en forme de problème *mixed integer linear programming*. Cette partie, non encore réalisée, pourra utiliser les algorithmes standards pour cette classe de problème. Ceci permet de spécifier des contraintes de performance ou de ressources.

Après la répartition, on reçoit un ensemble *d'Abstract Syntax Trees* (AST), un pour chaque processeur. Ces AST sont traduits en code C ou VHDL, puis compilés et synthétisés avec les outils de la cible choisi. Le code C contient les éventuels instructions pour charger des configurations/programmes dans une carte DSP ou FPGA, pour lancer les calculs et pour rapatrier les résultats.

2.3.2 Types dynamiques et déduction des types

Le langage MATLAB utilise des types dynamiques. Le compilateur MATCH contient des mécanismes de déduction de types automatique (comme dans le projet MADEO de l'UBO [LLFP01]) ou permet de les spécifier par des directives. Les mécanismes de déduction ne permettent pas de variables qui changent dynamiquement de type.

2.3.3 Comparaison avec PHRASE

Nous décrivons dans ce document une traduction de boucles intérieures qui est fortement inspirée de MATCH [BSC+00]. Nos propositions élargissent ces concepts à des boucles imbriquées et des portions de code de taille variable et non limités.

MATCH permet de diriger la transformation en matériel par des directives ou de manière entièrement automatique. Nous pensons que la transformation manuelle est trop laborieuse et qu'une transformation entièrement automatisée mène à des résultats peu performants.

Notre proposition est d'assister et guider un développeur dans une transformation interactive. PIPS fournit au développeur des analyses poussées du code source existant. Puis, notre compilateur met à disposition du développeur des transformations qui permettent d'optimiser le code pour une exécution en matériel. Le résultat est finalement transformé automatiquement en configurations de matériel reconfigurable et en code binaire de processeur séquentiel pour une architecture cible hétérogène.

2.4 DART

L'architecture DART [DCPS02] est découpée en un nombre de *clusters* qui travaillent indépendamment en parallèle. Un contrôleur de tâche distribue des configurations et les données à chacun des *clusters*.

2.4.1 Le *cluster*

Chaque *cluster* contient sa propre mémoire de données avec son contrôleur associé. Pour exécuter les calculs, le *cluster* dispose de 6 chemins de données reconfigurables (*DataPath Reconfigurable DPR*). De plus, on trouve un FPGA associé à sa logique de contrôle et sa mémoire de configurations.

Le FPGA n'est pas spécifié plus précisément et dans l'état actuel d'avancement du projet, il n'est pas exploité, ni manuellement, ni par les outils de compilation/synthèse automatiques. La section 2.4.2 approfondie l'architecture de DPR.

2.4.2 Le *DataPath* Reconfigurable DPR

Un DPR contient 4 mémoires de données indépendantes, chacune associée à son propre générateur d'adresses. De plus, le DPR possède deux registres, deux multiplieurs avec registre et deux ALU avec registres. Tous ces composants sont interconnectés entre eux et avec les DPR voisins par un réseau dynamiquement reconfigurable. Ce réseau est construit de manière à pouvoir alimenter en données les ALU et multiplieurs en parallèle, à chaque cycle d'horloge. Nous décrivons les possibilités de reconfiguration du réseau d'interconnexion dans la section 2.4.3.

Le DPR ne contient pas de contrôleur supplémentaire. Toute reconfiguration de son comportement se fait au niveau du *cluster* qui peut ainsi faire travailler les DPR sur des flux d'instructions différents ou sur des tâches parallèles.

2.4.3 Configuration HW/SW

Le réseau d'interconnexion interne aux DPR peut se reconfigurer dynamiquement de deux manières distinctes appelées configuration *hardware* (HW) et *software* (SW). La configuration HW est très flexible, mais nécessite plus de bits de configuration et donc plus de temps. La configuration SW réduit le nombre de possibles interconnexions et permet alors de reconfigurer très rapidement, en un seul cycle d'horloge.

Les deux modes de reconfiguration s'utilisent de la manière suivante : pour les cœurs de boucles, on applique la configuration HW et pour le reste du programme, la configuration SW trouve son utilité. On se rappelant la règle du 80/20 qui constate qu'un programme passe en général 80 % du temps dans seulement 20 % du code, le lecteur comprendra le raisonnement qui a mené à cette heuristique. On peut rentabiliser le surcoût de la reconfiguration HW utilisée dans les cœurs de boucles qui sont exécutées souvent sans reconfiguration. De l'autre coté, tout le reste de code, qui n'est exécuté que très peu de fois, n'utilisera pas tout le parallélisme disponible et se limite à la reconfiguration SW qui n'utilise pas toutes les possibilités du réseau d'interconnexion.

2.4.4 Flot de conception

Pour compiler/synthétiser vers l'architecture DART, les outils SUIF [WFW+94] servent à analyser le code C de l'application. L'équipe de Lannion a développé des phases supplémentaires permettant de dérouler les boucles et d'extraire les coeurs boucles. Ainsi, le compilateur peut séparer les parties destinées à la reconfiguration HW du reste pour une reconfiguration SW.

Pour l'optimisation et la génération de code, l'équipe utilise l'infrastructure Armor/Calife de l'IRISA/INRIA [CM99]. Les aspects de la reconfiguration SW on été modélisés en langage Armor utilisée par les outils Calife. Ainsi, l'outil CDART, développé par les auteurs, optimise et génère du code DPR utilisant la reconfiguration SW. Un deuxième outil nommé GDART génère du code DPR pour la reconfiguration HW.

2.4.5 Comparaison avec PHRASE

Comme notre projet, le compilateur de DART se fonde sur une boite à outils de compilateur de langage haut niveau existant. DART utilise SUIF, un concurrent direct de PIPS.

En différence avec le notre, le compilateur DART se limite à dérouler et déporter vers le reconfigurable uniquement les boucles intérieures. Notre approche permet de déporter des boucles imbriqués. De plus, les transformations de PIPS existants permettent d'aller plus loin, ils permettent de dérouler, de fusionner, d'échanger, d'ordonnancer *etc.* Au lieu d'automatiser complètement comme DART le fait, nous avons choisi de travailler interactivement en guidant le développeur par des analyses de code source poussées.

2.5 GARP

L'architecture GARP [CHW00] consiste d'un processeur MIPS couplé à un FPGA utilisé comme coprocesseur reconfigurable. Le FPGA est développé et optimisé spécialement pour cette application en coprocesseur. Un cache et des unités appelées *memory queues* permettent de mettre les données à disposition plus rapidement. Les *memory queues* chargent en autonomie des données rangées dans la mémoire de manière régulière.

2.5.1 La compilation

Du code C ANSI sans directives supplémentaires est utilisé en entrée. Pour la compilation, le projet utilise le compilateur SUIF pour produire une représentation intermédiaire composée de blocs basiques, des séquences d'instructions sans branches. Les blocs basiques reliés par des pointeurs forment le graphe de flux de contrôle.

Le compilateur développé pour GARP extrait automatiquement des boucles pour les exécuter sur le coprocesseur reconfigurable. La sélection des instructions à déporter se fait à l'aide de *profiling* et d'estimations de temps d'exécution.

Le compilateur regroupe les blocs basiques dans des *hyperblocks* un concept qui vient de la compilation pour VLIW *(Very Large Instruction Word)*. Le *hyperblock* correspond à la représentation d'une boucle du graphe de flux de contrôle.

Un *hyperblock* ne contient pas obligatoirement le code entier d'une boucle. Des branches rarement exécutés ou impossibles à implanter dans le coprocesseur FPGA sont exclus et interrompent l'exécution sur le coprocesseur si un tel chemin du graphe de contrôle est atteint.

Pour chaque *hyperblock*, le compilateur fait la fusion des graphes de flux de données pour exploiter le parallélisme au niveau instruction (ILP) et pour permettre le *pipelining* d'itérations consécutives.

A l'intérieur d'un *hyperblock*, les branches conditionnelles sont exécutés en parallèle et le résultat est choisi en fonction de la condition, évaluée en parallèle à l'exécution *(predication)*.

2.5.2 Comparaison avec PHRASE

Comme dans les projets MATCH et DART, uniquement des boucles intérieures peuvent être déportées sur le matériel reconfigurable. Comme dans DART, la compilation ne peut se faire que de manière complètement automatisée. Les inconvénients et différences avec PHRASE sont les mêmes (voir sections 2.3.3 et 2.4.5).

L'idée d'exclure des chemins de contrôle des boucles intérieures qui sont rarement parcourus et de les implanter en *hardware* paraît intéressante. Il est possible d'utiliser ce même concept avec notre compilateur. En remplaçant les boucles par des structures n'utilisant que des GOTO, le problème est déjà résolu. Il suffit de faire un saut à l'extérieur de la partie de code à déporter vers le reconfigurable (voir4.5).

2.6 SA-C

"Single-Assignement-C" [NBD+03] est conçu comme un dialecte de C qui enlève des caractéristiques difficilement à traduire en matériel et ajoute d'autres qui facilitent ce processus. SA-C interdit d'assigner une valeur à une variable plus qu'une seule fois, élimine l'utilisation des pointeurs (opérateurs "&" et "*") et interdit la récursivité.

SA-C ajoute des champs multidimensionnels et des types supplémentaires comme par exemple des entiers de largeur à spécifier. En plus, le SA-C connaît une nouvelle boucle à exécution parallèle semblable à celle connus du HPF *(High Performance Fortran)*.

Le code SA-C n'est pas pensé pour exprimer l'ensemble du programme, il est destiné pour les parties susceptibles de contenir des boucles exécutables sur les FPGA. Le programmeur code le reste du programme en C standard.

2.6.1 Architecture cible

L'architecture cible consiste d'un ordinateur hôte avec une carte FPGA. La carte FPGA peut contenir plusieurs FPGA et de la mémoire. Le compilateur traduit le code source en code pour la machine hôte qui configure et contrôle l'exécution et une deuxième partie qui s'exécute sur la carte FPGA.

2.6.2 La compilation

Flux de compilation [NBD+03, figure 2] montre le flux de conception de la compilation. Le résultat de toute la compilation comporte deux parties, le code à exécuter sur le processeur hôte et les données de configuration de FPGA.

Une première étape du compilateur identifie des boucles parallélisables pour exécution sur les FPGA. Puis, le compilateur intègre le reste du code SA-C dans le programme C pour former le programme du processeur hôte.

Ensuite, le compilateur transforme les boucles en SA-C en graphes de dépendance de données et flux de contrôle. Dans une prochaine étape, le compilateur produit une architecture abstraite. Cette description est enfin traduite en code VHDL qui est synthétisé avec des outils commerciaux.

Optimizations Sur les graphes de dépendance et de flux de contrôle s'exécutent des transformations pour aplatir la hiérarchie, fusionner des boucles adjacentes, dérouler des boucles, propager des constantes et faire du *strip mining*. En ce qui concerne le *strip mining*, le programmeur doit spécifier son utilisation avec des pragmas.

2.6.3 Comparaison avec PHRASE

Les idées sur les optimisations effectuées dans les graphes de dépendance et de flux de contrôle paraissent très proches des nôtres. Notre approche se distingue par l'utilisation d'un langage de haut niveau standard.

On peut générer du code en forme *single-assignment* de manière automatique. Nous ne voyons pas l'intérêt de limiter l'abstraction dans le code d'entrée par une telle contrainte. Les outils PIPS existants nous permettent de générer automatiquement des versions du code sans assignation multiple, récursivité et utilisation de pointeurs. A notre avis, la nécessité d'éliminer ces abstractions ne justifie pas d'en priver l'utilisateur.

2.7 Imagine, StreamC/KernelC

Les auteurs de [KRD+03] utilisent un modèle de calcul en flux qui permet de décomposer un calcul en noyaux *(kernels)* interconnectés par des flux *(streams)* (voir dans [KRD+03, figure 2] l'exemple d'un encodeur MPEG2 décomposé en noyaux et flux). La machine Imagine est programmée avec les langages **StreamC** et **KernelC**. Les auteurs citent StreamIt comme langage semblable [TKA02].

2.7.1 Langage exposant les différents niveaux de localité

La décomposition en noyaux et flux rend les différents **niveaux de localité** explicite. Ainsi, un noyau génère en interne des résultats intermédiaires. Il s'agit du niveau **local**. Les **flux** entre les noyaux représentent un deuxième niveau de localité. Les flux qui passent par les équipements d'entrée et de sortie définissent les variables du niveau **global**.

2.7.2 Les différents types de parallélisme exploités

Le modèle de calcul en flux expose le parallélisme à différents niveaux. A l'intérieur d'un noyau, on peut exploiter le parallélisme au **niveau instruction**

(ILP Instruction Level Parallelism), comme c'est fait dans les modèles de calcul traditionnels. Puis, les noyaux exécutent par définition le même calcul sur chaque élément des flux entrant et rendent ainsi le parallélisme au **niveau des données** explicite.

2.7.3 Architecture

Architecture globale L'article donne un schéma d'architecture de haut niveau (voir [KRD+03, figure 2]) d'un processeur de flux. Cette architecture est composée d'un fichier de registres de flux *(Stream Register File)* qui est connecté à une unité d'exécution de noyaux, une interface de DRAM et un processeur d'application. Le **fichier de registres de flux** stocke toutes les données qui correspondent à la communication par flux. En début du calcul, les données entrent dans le fichier de registres par l'**interface DRAM** et puis les différents noyaux sont appliqués par l'**unité d'exécution de noyaux**. Le **processeur d'application** sert à agencer l'ensemble des opérations.

Architecture de l'unité d'exécution des noyaux L'unité d'exécution des noyaux consiste de N groupes d'ALU identiques, contrôlées de façon SIMD par un microcontrôleur. Chaque groupe d'ALU contient en outre des ALU les bus et les registres nécessaires pour exécuter les instructions d'un noyau.

Localité explicite exploitable par une hiérarchie de mémoire L'architecture décrite réalise une hiérarchie en trois niveaux. Au plus haut niveau, il y a une mémoire DRAM classique. Puis, il y a les fichiers de registres de flux. Le dernier niveau le plus local de la hiérarchie consiste des registres à l'intérieur de l'unité d'exécution des noyaux. Les trois niveaux de la hiérarchie correspondent directement aux niveaux de globalité explicites du langage de programmation StreamC.

2.7.4 Comparaison avec les processeurs de vecteurs

Un processeur de vecteurs utilise un fichier de registres comparables au fichier de registres de flux d'Imagine. Puis, il exécute une série d'instructions identiques. Imagine ajoute un niveau de hiérarchie de mémoire et un niveau d'exécution d'instructions (à l'intérieur des unités d'exécution de noyaux). Il devient ainsi possible d'exécuter toutes les instructions qui concernent les mêmes données de façon groupée en utilisant des registres locaux plus rapides. De l'autre coté, la bande passante nécessaire du fichier de registres de flux diminue. Les ressources libérées peuvent servir à réaliser plus d'ALU et augmenter le parallélisme.

2.7.5 Comparaison avec PHRASE

Le projet Imagine simplifie le processus de la compilation en utilisant des langages de programmation propres et en créant une architecture de machine cible associée. Phrase utilise des langages de haut niveau existants et cible des architectures hétérogènes, sans en retenir une seule et unique.

2.8 RAW

Les fréquences d'horloge augmentent, la taille des structures dans le silicium diminue et les surfaces des puces augmentent. Ces trois tendances font que les délais de transmission à l'intérieur d'un processeur ne sont plus négligeables. Actuellement, avec une fréquence d'horloge de 2 GHz, un signal prend deux cycles d'horloge d'un bord à l'autre d'une puce processeur et il est prévisible que ce facteur passera à 10 ou plus avec les processeurs cadencés à 10 GHz.

Les délais de communication posent alors des problèmes aux concepteurs de processeur actuels ISP ("Instruction Set Processor"). Par exemple, ils doivent prévoir des cycles et de registres supplémentaires pour raccourcir les chemins critiques introduits par la communication à l'intérieur de la puce. Les auteurs du papier [TKM+02] concluent que les architectures actuelles se prêtent mal à l'exploitation des densités croissantes du silicium et qu'il faut des changements profonds des architectures.

2.8.1 Architecture

La solution proposée, les processeurs RAW, consiste à faire des processeurs très simples selon les concepts RISC, de les répliquer en forme de tuiles régulières sur le silicium et de les interconnecter par un réseau de communication simple.

Un processeur contient de la mémoire d'instructions, des registres, de la mémoire, une ALU et une partie de logique reconfigurable. Selon les principes des processeurs RISC, les processeurs utilisent le *pipelining*.

Le réseau de communication ne permet que les transferts vers les tuiles de processeurs voisins, avec des temps de latences minimales. Des transferts à longue distance doivent se composer de petits pas de tuile en tuile. Comme la largeur d'une tuile est conçu pour correspondre à peu près à la distance qu'on peut parcourir pendant un cycle d'horloge, il est de toute façon impossible organiser des communications synchrones plus rapides.

Les temps de latence sont du même ordre de grandeur pour les transferts au sein d'un processeur et entre tuiles voisines. L'infrastructure de communication de l'architecture RAW permet donc, en plus du *pipelining* à la RISC à l'intérieur du même processeur, de créer des pipelines à travers de tuiles voisines.

Le réseau de communication fonctionne en deux modes, le mode appelé "statique" qui est décidé au moment de la compilation et un mode plus dynamique qui change en fonction des résultats de calcul pendant l'exécution du programme. Les chemins à parcourir en mode routage statique sont indépendants des calculs et connus d'avance. L'aiguillage peut donc être préparé dans les cycles d'horloge avant le transfert. Par conséquence, les temps de latences sont plus faibles en routage statique qu'en routage dynamique.

2.8.2 La compilation

La configuration de l'infrastructure de communication de chaque tuile est reprogrammée pendant chaque période d'horloge avec des instructions comparables aux instructions qui spécifient les calculs à effectuer. Le compilateur se charge donc d'optimiser les communications entre les tuiles et des calculs sur chaque tuile.

L'article parle de l'existence du compilateur RAWcc. Ce compilateur peut, à partir de code source C ou Fortran, automatiquement :

– partitionner le graphe du programme
– placer les instructions sur les processeurs
– programmer le routage statique entre les tuiles

Pour les tests de performance Specfp, le compilateur atteint une accélération de 6 à 11 sur un processeur RAW à 16 tuiles par rapport à une seule tuile. Pour un RAW à 32 tuiles, il atteint des accélérations de 9 à 19.

Trois compilateurs ont été développés pour l'architecture RAW. Le premier est construit sur la base de gcc et permet la programmation des calculs et de la communication tuile par tuile. Ce mode de fonctionnement est intéressant pour des tâches très structurées, le mode développement est comparable à celui applicable aux ASIC.

Le deuxième compilateur, RAWcc, s'adresse à des applications qui sont typiquement exécutées sur les processeurs universels actuels. Ce compilateur-ci, utilise les multiples tuiles processeurs avant tout pour exploiter le parallélisme au niveau instruction (ILP) (voir section 2.8.3).

Un troisième compilateur est spécialisé aux applications de flux de données *(streaming)* (voir [GTK+02]).

2.8.3 Ordonnancement local/temporel des blocs fondamentaux pour exploiter l'ILP

L'article [LBF+98] parle du deuxième compilateur qui utilise les multiples tuiles processeurs pour exploiter le parallélisme au niveau instruction (ILP) d'applications codées pour un seul processeur.

Un bloc fondamental *(basic block)* comporte le code de programme entre deux instructions de contrôle. Il ne comporte pas de branches conditionnelles et pas de boucles.

L'ordonnancement se fait en 6 étapes :
– Transformations initiales de code
– Partitionnement des instructions
– Partitionnement global des données
– Placement des données et des instructions
– Génération du code de communication
– Ordonnancement des évènements

Transformations initiales de code D'abord, le code du bloc fondamental est transformé en forme SSA *(Static Single Assignment)*. Puis, des instructions supplémentaires *read* et *write* sont ajoutées en début et fin du bloc pour toutes les variables qui entrent et sortent. Ensuite, le compilateur transforme toutes les instructions en forme trois opérateurs en utilisant des nouvelles variables temporaires. Le code résultant est déjà plus proche du code RISC. En dernier, le graphe de dépendance des données est élaboré *(DFG, data flow graph)*.

Partitionnement des instructions Les instructions du *dataflow graph* sont d'abord regroupées en minimisant la communication entre les groupes. Le compilateur utilise pour ceci l'hypothèse d'un coût de communication indépendant de la distance en parcourus. En fin de cette phase, le nombre de groupes est réduit pour correspondre au nombre de tuiles de la configuration de la machine cible.

Partitionnement global des données Les données sont partitionnées en groupes et associées au groupes d'instructions.

Placement des données et des instructions Arrivé à cette phase, le compilateur peut associer les groupes de données et d'instructions aux tuiles de l'architecture RAW.

A ce stade, le compilateur peut optimiser le coût de communication en fonction de la distance à parcourir.

Génération du code de communication Maintenant, le compilateur connaît, pour chaque arête du graphe de flux de données, la route exacte que les données doivent parcourir. Les instructions de routage, `send`, `route` et `receive` sont générées et ajoutées au graphe de flux de données qui est déjà placé sur les tuiles.

Ordonnancement des évènements L'ordre d'exécution des instructions de calcul et de routage peut être déterminé pendant cette dernière phase.

2.8.4 Ordonnancement du flux de contrôle

Le flux de contrôle entre les blocs fondamentaux se fait en utilisant le concept de **branches globales asynchrones**. La tuile qui fait le dernier calcul sur la variable booléenne qui conditionne le saut, l'envoie en mode *multicast*. Cette communication se fait par le même réseau de communication statique que les communications des données à l'intérieur du bloc fondamental. Puis, toutes les tuiles concernées effectuent le branchement de manière asynchrone.

Pour les boucles intérieures qui contiennent des branches *if-then-else*, le compilateur essaye de **localiser le flux de contrôle** sur une seule tuile. Ceci est intéressant dans les cas de boucles déroulées, parce qu'on évite le surcoût de communication et de synchronisation des branches globales asynchrones. Plusieurs itérations sont ainsi exécutées en parallèle sur des tuiles différentes. Différents itérations ne sont pas forcement synchronisées, même s'elles contiennent des *if-then-else*.

2.8.5 Résistance aux retards imprévisibles

Le compilateur RAWcc essaye de prédire le comportement dynamique du réseau de tuiles et d'optimiser l'ordonnancement. Il reste néanmoins des événements non prévisibles qui dépendent des données. Une requête cache non satisfaite par exemple bloque le processeur. L'asynchronisme entre les tuiles processeur fait que seulement une seule tuile est directement affectée par un tel événement. D'autres tuiles seront bloquées seulement au moment qu'ils doivent attendre un résultat d'une tuile bloquée. Cet asynchronisme donne un avantage de performance sur des machines synchrones comme par exemple les machines VLIW qui sont bloquées entièrement.

2.8.6 Comparaison avec PHRASE

La logique reconfigurable est utilisée en forme d'ALU reconfigurable. Les calculs qui s'y exécutent sont peu complexes. Ils doivent s'exécuter en un seul

cycle d'horloge. Le contrôle reste sous la maîtrise du logiciel et n'est jamais exercé par du matériel reconfigurable.

De plus, d'après [AAB+97], la partie du compilateur qui traite la logique reconfigurable de RAW n'est pas encore très avancée. L'article projette d'identifier automatiquement des fonctionnalités à exécuter sur le reconfigurable et de synthétiser pour les ALU reconfigurables. En somme, RAW dispose de ressources reconfigurables très limités par rapport à l'architecture PHRASE.

Le compilateur décrit dans [LBF+98] développe un certain nombre d'heuristiques pour distribuer et ordonnancer le code sur les tuiles de RAW. On pourrait en déduire des nouvelles stratégies pour guider l'utilisateur de notre compilateur interactif.

2.9 SCORE

En comparant des applications réalisées en logiciel et par du matériel reconfigurable, les auteurs de [CCH+00] font remarquer deux difficultés majeures des réalisations reconfigurables :

La première concerne la **compatibilité entre différentes architectures** cibles. Deux architectures sont compatibles avec une réalisation logicielle s'elles respectent la même ISA *(instruction set architecture)*. Par contre, deux architectures de FPGA, y inclus du même constructeur, ne sont jamais compatibles. Dans tous les cas, il faut passer par les outils de CAO (conception assistée par ordinateur).

La deuxième difficulté est le manque des abstractions de **ressources virtuelles**. Pour une application logicielle, on dispose de l'abstraction de la mémoire virtuelle. Pour une application réalisée par du matériel reconfigurable, on manque de concepts équivalents pour la mémoire et pour les ressources logiques. Le concepteur programmeur doit alors veiller manuellement au respect des limitations de ressources.

2.9.1 Modèle de calcul

Pour le développeur, les créateurs de SCORE définissent un modèle de calcul qui spécifie la sémantique de calcul que le matériel met à disposition. Ce modèle est donné à deux niveaux, un niveau bas qui donne une vue du comportement au moment de l'exécution et un niveau plus élevé et accessible au programmeur. Voir les deux sections suivantes (2.9.1 et 2.9.1).

Modèle d'exécution Le modèle d'exécution introduit les abstractions de la page de calcul, du lien de flux et des segments de mémoire. La **page de calcul** *(compute page)* représente un bloc de logique reconfigurable de taille fixe qui sert d'unité de base de logique virtuelle. Un **lien de flux** *(stream link)* correspond à une interconnexion entre deux pages de calcul qui véhicule des données d'un page à une autre. Les **segments de mémoire** servent garder l'information qui est émise sur un lien de flux vers une page de calcul qui n'est pas chargée à ce moment, ou alternativement à stocker des valeurs intermédiaires locales à une page de calcul.

Modèle de programmation Au niveau du modèle de programmation, on retrouve des abstractions semblables, mais de plus haut niveau. La page de

calcul devient un **opérateur** qui regroupe des calculs, mais sans limitation de ressources. De la même manière, la **page de mémoire** existe aussi dans le modèle de programmation, mais elle aussi sans limitation de taille. Le **flux** n'ayant pas de limitation dans le modèle de bas niveau continue à exister dans le modèle de programmation.

Puis, le modèle de programmation ajoute la possibilité de créer des opérateurs qui consomment des données à des débits dépendants des données. En plus, on peut créer et interconnecter des et opérateurs dynamiquement.

Pour programmer dans ce modèle, les auteurs utilisent un mélange de C++ et un langage intermédiaire expérimental appelé TDF. TDF décrit les opérateurs et leur composition.

2.9.2 Architecture

L'architecture comporte un ou plusieurs processeurs ISA (ou séquentiels), des pages de calculs formées d'un FPGA adapté aux contraintes et des blocs de mémoire configurable. De plus, il existe un réseau d'interconnexion et des mémoires cache.

Le processeur sert à exécuter les opérateurs difficilement exécutables par la logique reconfigurable et aux tâches d'ordonnancement et de contrôle. Les blocs de mémoire contiennent leurs propres générateurs d'adresses pour réaliser les opérations sur des flux.

2.9.3 Apport principal

Les deux modèles de programmation créent l'abstraction **mémoire et ressources de calcul virtuelles**. Le modèle de programmation met à disposition des ressources infinies. Les programmes qui respectent le modèle sont destinés à être exécutés suivant un ordonnancement en temps partagé par les ressources limitées du modèle d'exécution.

Un deuxième apport est un niveau de compatibilité introduit par le modèle d'exécution. Tout système respectant ce modèle peut, indépendant de la performance des pages de calcul et de leur nombre, exécuter toute application compilée / synthétisée pour le même modèle.

2.9.4 Comparaison avec PHRASE

Le partitionnement en pages est faite de manière manuelle. De même, le partitionnement entre le processeur ISA et les pages de logique reconfigurable n'est pas automatisé. Il n'y a pas encore de compilation/synthèse automatique. Jusqu'ici, uniquement le système d'exécution *(run time)* est réalisé en forme de simulateur.

Sera-t-il possible de compiler automatiquement d'un langage haut niveau vers le langage TDF ? Le projet n'a pas encore abordé cette question. Deux solutions semblent possible : soit on compile d'un langage de haut niveau vers le mélange de TDF/C++, soit on utilise des langages faits pour le nouveau paradigme de programmation, qui sont facile à traduire en TDF.

Le projet SCORE essaye de résoudre le problème de la compilation vers des ressources reconfigurables par un nouveau paradigme de programmation et une nouvelle architecture matérielle adapté. Le projet n'aborde pas les problèmes

de compilation associés et accepte le risque de perdre la compatibilité avec les langages de haut niveau actuels.

2.10 FlexWare

L'article [PS02] décrit toute une chaîne de développement *retargetable* adaptable au processeur cible pour des systèmes intégrés sur une puce *(SoC, System on Chip)*. Ces systèmes sont typiquement composés d'un processeur RISC standard et d'un ASIP *(Application Specific Instruction-Set Processor)*. Les ASIP sont souvent des DSP spécialisés.

Un ASIP est un processeur avec un jeu d'instructions optimisé pour une classe d'applications. Les ASIP permettent ainsi de faire du développement au niveau logiciel avec des langages de haut niveau et remplacent ainsi de plus en plus le matériel spécifique.

Le prix pour cette méthode de développement consiste dans le fait qu'il faut développer une chaîne de compilation / simulation / *debugging* (et éventuellement d'établissement de profils d'exécution *profiling*) pour chaque nouveau type de processeur. Pour limiter le coût du développement de cette chaîne de compilation, ST Microelectronics développe ces outils de manière à ce qu'ils soient adaptables au processeur cible.

Ce qui nous intéresse pour le projet PHRASE, c'est le compilateur de FlexWare. En fait, il y deux compilateurs FlexCC1 et FlexCC2.

FlexCC1 est construit à partir de gcc.

FlexCC2 est construit à partir du produit commercial CoSy de l'entreprise ACE (www.ace.nl). CoSy sert comme *front end* du compilateur avec son format intermédiaire de haut niveau et met à disposition un nombre d'optimisations *(constant folding, tail recursion, switch optimization...)*. ST Microelectronics a programmé deux autres optimisations plus spécifiques aux DSP. Il s'agit de la transformation de variables de type champ vers des pointeurs et de l'optimisation de boucles exécutées par le matériel.

ST Microelectronics a ajouté aussi un *backend* propre qui effectue des optimisations bas niveau (allocation de registres, ordonnancement et *peephole optimization*)

Dans les directions futures en fin d'article, il est expliqué que la chaîne de développement sera bientôt élargie pour compiler sur des SoC multiprocesseur et qu'il y a déjà des travaux en cours pou utiliser du *hardware / software codesign*.

2.10.1 Comparaison avec PHRASE

Nous sommes curieux de voir des articles sur FlexWare avec le *hardware / software codesign* et des systèmes multiprocesseur.

2.11 ROOM

Les auteurs de [RL02] explorent la possibilité de compiler du code objet en code HDL (VHDL) synthétisable (sur un FPGA). L'architecture de la machine synthétisée consiste d'un processeur objet pour chaque classe du programme source et d'un réseau de communication permettant de transférer des messages entre objets. Chacun de ces processeurs objet contient une mémoire d'instances, une unité de calcul, une mémoire de code pour les méthodes, et un séquenceur

qui applique les méthodes de la mémoire de code. Le réseau de communication est couplé par des FIFO pour permettre l'exécution parallèle asynchrone entre plusieurs processeurs objet.

En principe, uniquement les processeurs objet qui réalisent les classes des types de base contiennent des unités de calcul, parce que seulement eux nécessitent des opérateurs (par exemple les types `bool` et `int`). Pour optimiser la performance, on peut ajouter des opérateurs spécifiques à toute autre classe et les réaliser par des unités d'exécution dans le processeur correspondant.

2.11.1 Compilation

L'intérêt de l'architecture décrite consiste en la possibilité de la générer de manière automatique à partir d'un logiciel objet. Ce processus ce fait de la manière suivante :

Les processeurs objet pour les types simples (contenant des opérateurs) et un processeur pour les classes complexes du programme à compiler sont développés une foi pour toute. D'abord, un compilateur produit du code assembleur exécutable par les processeurs objet. Ce code assembleur est annoté pour décrire sur quel processeur objet il doit être exécuté. Puis, un assembleur génère le code machine pour chaque processeur objet ainsi que les instructions de communication nécessaires.

2.11.2 Comparaison avec PHRASE

PHRASE ne prend pas pour l'instant en compte les langages objets. ROOM ne compile que ces langages ci. Le concept de compilation de source orienté objet vers du FPGA est récent et nécessite encore la résolution d'un certain nombre de problèmes. Considérons les suivants :

Implémentation des abstractions des langages objet La classe d'architecture élaborée par ROOM ne permet que de réaliser une petite partie des concepts des langages objet haut niveau. Par exemple, l'architecture n'a aucun support pour l'héritage.

Parallélisme L'architecture ne permet pas d'exploiter du parallélisme qui utilise plusieurs fois le même opérateur sur des objets de la même classe. Il n'existe qu'un seul processeur par classe et ce processeur garde toutes les instances. Ce processeur ne peut donc pas facilement être répliqué.

***Swapping* des données** Un processeur objet peut garder un nombre limité d'instances dans sa mémoire interne. Cette mémoire est réalisée à l'intérieur du FPGA. Aucun mécanisme de hiérarchie de mémoire pour des grandes quantités de données n'est prévu dans le concept.

Swapping du logiciel Tous les processeurs de classes restent en permanence configurés sur le FPGA. Aucun mécanisme de *swapping* n'est prévu. Ceci limite le nombre de classes.

Conception des opérateurs spécifiques Les gains de performance de l'architecture présentée sont possibles en utilisant des opérateurs spécifiques. Le problème de la synthèse automatique de ces parties reste le même que le développement de configurations FPGA habituel.

3 Exemple de code source

L'exemple de code source donnée dans la figure 1 représente des parties de code d'un filtre de Deriche. Ce filtre sert à la détection de contours et est utilisé par exemple dans des applications d'imagerie médicale pour marquer les limites d'une tumeur. Les deux figures montrent uniquement l'entête de procédure Fortran et une première boucle de lissage en direction horizontale. La procédure complète se poursuit par le même lissage en direction verticale pour calculer finalement le gradient sur toute l'image.

FIG. 1 – Initialisations et lissage horizontal

```
SUBROUTINE Deriche(TT_Image, alpha)
  INTEGER :: col,lig,SV,SH,frame_height,frame_width,rs,inc
  REAL(4) :: gamma,y,y_1,y_2,a1,b1,b2
  frame_height = Timage%Height
  frame_width = Timage%Width
  rs = Timage%Stride
  inc = Timage%Increment
  image = Timage%ptrImage
  gamma = exp(-alpha)
  a1 = Carre( 1 - gamma )
  b1 = 2 * gamma
  b2 = -Carre( gamma )
  DO lig = 0,frame_height-1
    y_1 = 0
    y_2 = 0
    col = 0
    DO WHILE (col < frame_width*inc)
      pixel = image(lig*rs + col+1)
      y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
      y_2 = y_1
      y_1 = y
      image(lig*rs + col+1) = y
      col = col+inc
    END DO
    ! ... calcul de lissage dans le sens du retour
  END DO
```

4 Transformations en vue d'une exécution sur matériel reconfigurable

Dans cette section du rapport, nous montrons comment il est possible de transformer du code de langage de programmation de haut niveau en configurations pour du matériel reconfigurable. Nous commençons avec des lignes seules qui sont transformées en simple fonction combinatoire. Puis, nous montrons comment transformer plusieurs lignes, des boucles et des boucles imbriqués. Il est ainsi possible de transformer, en commençant par les boucles intérieures et en remontant la hiérarchie, des parties de complexité variable en configurations FPGA.

4.1 Déporter un calcul en forme de fonction combinatoire

Dans un premier temps, le programmeur place une directive de compilation, juste avant la ligne à déporter sur le matériel reconfigurable (voir figure 2). Le nouveau compilateur fondé sur PIPS analyserait ce code et ajouterait ce qui est nécessaire pour configurer le FPGA, transférer les données, lancer le calcul et récupérer le résultat (figure 3). En plus, il générerait du code VHDL ou SmallTalk pour les outils MADEO (figure 4) qui décrit les configurations FPGA associées.

FIG. 2 – Réaliser une ligne en forme de fonction combinatoire

```
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  $PRAGMA_RECONFIGURABLE
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
```

FIG. 3 – Reste de code pour processeur séquentiel

```
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  !$PRAGMA_RECONFIGURABLE
  !y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  lib_configureFPGA('FUNC1')
  lib_sendParametersToFPGA('FUNC1', a1, pixel, b1, y1, b2, y2)
  lib_executeFPGA('FUNC1')
  lib_receiveResultsFromFPGA('FUNC1', pixel)
```

FIG. 4 – Code SmallTalk pour générer une configuration FPGA

```
FUNC1:
y := [a1 * pixel + b1 * y_1 + b2 * y_2]
```

4.1.1 Réutiliser une configuration dans une boucle sans reconfiguration inutile

Dans le code du paragraphe 4.1, le matériel reconfigurable utilise de façon répétitive la même configuration à l'intérieur de la boucle. Pour éviter de recharger la même configuration, on pourrait sortir l'appel de la fonction de librairie `lib_configureFPGA` de la boucle (figure 5). Le compilateur pourrait même le monter de plusieurs niveaux de boucles, aussi loin qu'il peut sans interférer avec d'autres configurations.

FIG. 5 – Sortir la configuration de la boucle

```
!$PRAGMA_RECONFIGURABLE
!pixel = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
lib_configureFPGA('FUNC1')
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  lib_sendParametersToFPGA('FUNC1', a1, pixel, b1, y1, b2, y2)
  lib_executeFPGA('FUNC1')
  lib_receiveResultsFromFPGA('FUNC1', y)
```

Une solution plus élégante consiste à éviter la reconfiguration au moment de l'exécution. L'implémentation de l'appel de `lib_configureFPGA` vérifierait si la dernière configuration est la même que celle nouvellement demandée et retourne éventuellement reconfiguration et sans perdre plus de temps.

4.1.2 Déporter plusieurs lignes en forme de fonction combinatoire

Le paragraphe 4.1 présente un cas spécial en déportant une seule ligne dans le matériel reconfigurable. Nous proposons dans la figure 6 l'utilisation du couple de directives `PRAGMA_RECONFIGURABLE_BEGIN` et `PRAGMA_RECONFIGURABLE_END` pour déporter le calcul correspondant à plusieurs lignes de Fortran.

FIG. 6 – Réaliser plusieurs lignes en forme de fonction combinatoire

```
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  $PRAGMA_RECONFIGURABLE_BEGIN
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  y_2 = y_1
  y_1 = y
  $PRAGMA_RECONFIGURABLE_END
```

Dans cet exemple apparaît la variable "y" qui n'est utilisée que comme variable intermédiaire du calcul fonctionnel. Le compilateur peut, en utilisant les fonctionnalités existantes de PIPS, reconnaître ce fait et générer les codes pour le processeur séquentiel et le matériel reconfigurable en fonction.

4.1.3 Atomiser des lignes de code source

L'expression de langage de haut niveau `(a1*pixel + b1*y_1 + b2*y_2)` est relativement complexe. Pour pouvoir placer les directives qui pilotent la transformation en code de matériel reconfigurable de manière plus fine, il peut être intéressant de transformer cette ligne de code en plusieurs, plus simples. PIPS nous met à disposition la transformation adaptée, appelée "atomisation". L'atomisation sépare des assignations d'expressions complexes pour obtenir une série équivalente d'assignations où l'expression consiste d'une seule opération et de seulement deux paramètres (figure 7).

FIG. 7 – Atomiser une ligne de code source

```
tmp1 = a1*pixel
tmp2 = b1*y_1
tmp3 = b2*y_2
tmp4 = tmp1 + tmp2
y    = (TT_type) tmp4 + tmp3
```

4.1.4 Insérer des registres dans une fonction combinatoire

Le principe du fonctionnement synchrone des automates veut que le temps de propagation à travers de la logique combinatoire soit à peu près le même pour toutes les fonctions. Dans le cas le plus commun, on n'utilise qu'une seule horloge pour le circuit entier.

Si on transforme de lignes de code source de langue haut niveau en logique combinatoire pour faire utiliser le résultat immédiatement par une machine à états, il faut limiter les temps de propagation de la logique combinatoire pour s'adapter à la fréquence d'horloge commune. Pour limiter la complexité et le temps de propagation des fonctions combinatoires implantées dans le matériel reconfigurable, on peut alors les diviser en ajoutant des registres. Pour exprimer ceci dans le code du langage de haut niveau, nous proposons la nouvelle directive `PRAGMA_CLOCK` (figure 8).

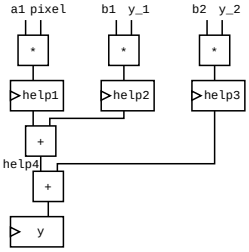
FIG. 8 – Insertion de registres dans une fonction combinatoire

```
$PRAGMA_RECONFIGURABLE_BEGIN
tmp1 = a1*pixel
tmp2 = b1*y_1
tmp3 = b2*y_2
$PRAGMA_CLOCK
tmp4 = tmp1 + tmp2
y    = (TT_type) tmp4 + tmp3
$PRAGMA_CLOCK
$PRAGMA_RECONFIGURABLE_END
```

Notre compilateur produit à partir de ce code d'entrée le reste de code à

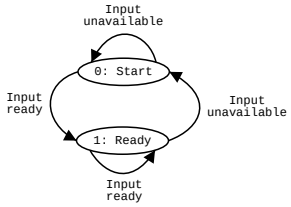
exécuter sur un processeur séquentiel et du code pour synthétiser une configuration FPGA. Nous montrons dans la figure 9 en forme de schéma RTL *(register transfer logic)* la logique à produire.

FIG. 9 – Schéma RTL de la fonction combinatoire avec des registres



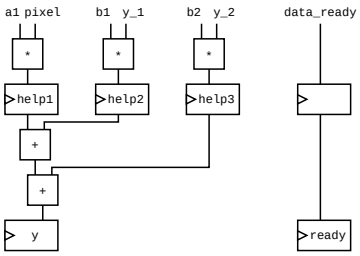
Pour pouvoir signaler la disponibilité de données à une machine à états consécutive, nous associons un automate de contrôle simple à chaque registre inséré dans la fonction combinatoire (figure 10). Cet automate consiste d'une seule bascule. L'entrée du registre reçoit le signal "données disponibles", et l'état du registre correspond à la disponibilité des données pour la logique en aval.

FIG. 10 – Diagramme d'états de l'automate associé à chaque registre inséré dans une fonction combinatoire



La figure 11 montre le schéma RTL de notre circuit exemple avec la bascule supplémentaire pour chaque étage de registres. Il reste à intégrer l'attente de ce signal dans les fonctions de transition de toute machine à état consécutive.

FIG. 11 – Schéma RTL avec automate de contrôle associé



4.1.5 Accès à une mémoire externe dans une fonction combinatoire

Dans tous les exemples montrés jusqu'ici, nous avons évité d'inclure l'accès à des variables de type *array*. Dans le cas général, il est impossible de stocker la

totalité de ces variables dans des mémoires de type registre au sein du matériel reconfigurable de grain fin.

Regardons l'exemple de la figure 12 : les deux lignes de code source contiennent un calcul d'adresse, un accès à la variable *array* et le calcul qui exploite la donnée. Les directives qui entourent ce segment de code, spécifient qu'il faut exécuter les trois opérations pendant un seul cycle d'horloge.

FIG. 12 – Fonction combinatoire avec accès à un champs

```
$PRAGMA_RECONFIGURABLE_BEGIN
pixel = image(lig*rs + col+1)
y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
$PRAGMA_RECONFIGURABLE_END
```

Nous proposons d'automatiser la transformation qui sépare l'accès à la mémoire externe. Pour l'effectuer, PIPS doit détecter cet accès à la mémoire et introduire un registre juste avant et après (voir figure 13).

FIG. 13 – Séparation du calcul d'adresse et de l'accès au champs

```
$PRAGMA_RECONFIGURABLE_END
adresse = lig*rs + col+1
$PRAGMA_CLOCK
pixel = image(adresse)
$PRAGMA_CLOCK
y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
$PRAGMA_CLOCK
$PRAGMA_RECONFIGURABLE_END
```

Le code source transformé correspondra, selon les règles décrites dans les sections précédentes, à une configuration de matériel reconfigurable comme celle montrée par la figure 14.

Nous introduisons l'interface générique de la figure 15 pour rendre transparent l'accès à des variables *array* gardées dans la mémoire. L'automate attends les signal `Enable`, prend en entrée l'index, lis la variable de la mémoire et annonce la validité de la donnée en sortie par le signal `Data ready`.

Le nombre de cycles nécessaires est variable et change avec la réalisation de la mémoire. L'interface permet même des durées d'attente non déterministe comme une mémoire hiérarchique avec différents niveaux de cache peut les produire.

Les deux signaux de synchronisation `Enable` et `Data ready` se connectent directement aux automates de contrôle de la logique combinatoire en amont et aval. L'interface d'automate de lecture se met donc à la place d'un étage de chemin de données avec son automate de contrôle associé.

Dans la figure 16, nous montrons l'automate de lecture de mémoire intégré dans la fonction combinatoire présentée dans la figure 14. L'automate de lecture remplace aussi les registres qui étaient placés avant et après la lecture de la mémoire. Ceci laisse à la réalisation la liberté d'exécuter la lecture en un nombre de cycles variant de `0` à `n`.

FIG. 14 – Un cycle séparé pour l'exécution de la lecture de mémoire

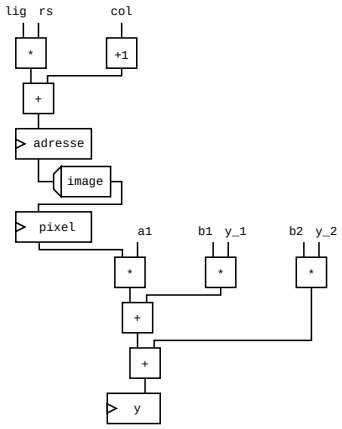


FIG. 15 – Automate d'accès à la mémoire

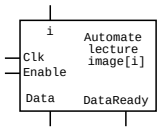
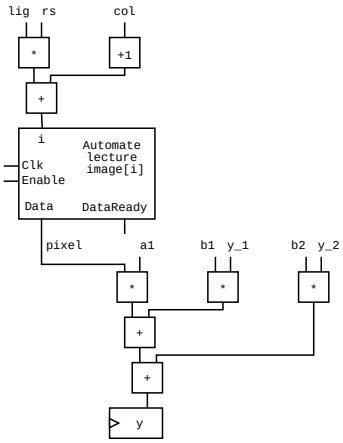


FIG. 16 – Automate d'accès à la mémoire dans le *data path*



Ce que nous avons montré jusqu'ici pour la lecture de variables, peut se faire de la même manière pour les écritures. La partie du programme qui s'exécute en logique reconfigurable contiendra un certain nombre d'automates de lecture et d'écriture de variables qui sont tous stockés dans une même mémoire. L'ensemble de ces automates réalise alors la résolution des conflits d'accès et garantit le l'ordre d'exécution des lectures et écritures. L'ensemble des automates peut aussi réaliser des architectures de mémoires complexes comme par exemples des mémoires à double accès.

4.2 Exécution parallèle

Le matériel peut exécuter des tâches de manière parallèle. Pour pouvoir exprimer cette possibilité en langage de haut niveau, nous proposons la directive `PARALLEL_GROUP` (voir figure 17).

FIG. 17 – Exécution parallèle

```
DO WHILE (col < frame_width*inc)
  $PRAGMA_DO_PARALLEL_BEGIN
  $PRAGMA_PARALLEL_GROUP
  pixel = image(lig*rs + col+1)
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  y_2 = y_1
  y_1 = y
  image(lig*rs + col+1) = y
  $PRAGMA_PARALLEL_GROUP
  pixel_b = image((lig+1)*rs + col+1)
  y_b = (TT_type)(a1*pixel_b + b1*y_1_b + b2_b*y_2_b)
  y_2_b = y_1_b
  y_1_b = y_b
  image(lig((lig+1)*rs + col+1) = y_b
  col = col+inc
  $PRAGMA_DO_PARALLEL_END
END DO
```

La directive garantit au compilateur que le code des groupes parallèles puisse s'exécuter en parallèle sans changer la sémantique du programme. Le code montré dans l'exemple exécute en parallèle le lissage de l'algorithme de Deriche.

Si un des deux groupes contient des directives `PRAGMA_CLOCK`, il faut synchroniser les deux flux de contrôle après l'exécution des groupes parallèles. Cette synchronisation pourrait consister simplement d'un automate qui attend les différentes branches du flux de contrôle avant de continuer.

4.3 Boucles intérieures et automate de contrôle associé

Pour une boucle, il est facilement possible de générer une machine à états fini si elle ne contient pas d'autre boucle dans son corps. Le projet MATCH montre ce procédé pour la transformations de boucles intérieurs du langage Matlab en VHDL [BSC+00].

Prenons comme exemple le code de la figure 18 : La directive `$PRAGMA_RECONFIGURABLE` désigne la boucle à transformer en machine à états.

FIG. 18 – Une boucle à transformer en machine à états fini

```
$PRAGMA_RECONFIGURABLE_BEGIN
y_1 = 0
y_2 = 0
col = 0
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  ! ...
  col = col+inc
END DO
```

La figure 19 montre le code Fortran obtenu par transformation automatique avec le nouveau compilateur. Ce code contient le code déporté, mis en commentaire. Ce code original est remplacé par les appels de librairies qui configurent le matériel avec l'automate à états finis `FSM1`, envoient les paramètres en entrée, lancent l'exécution et récupèrent les résultats.

FIG. 19 – Reste de code pour automate déporté sur le FPGA

```
!$PRAGMA_RECONFIGURABLE_BEGIN
! Initialisation ...
!DO WHILE (col < frame_width*inc)
!  pixel = image(lig*rs + col+1)
!  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
! ...
lib_configureFPGA('FSM1')
lib_sendParametersToFPGA('FSM1', image)
lib_executeFPGA('FSM1')
lib_receiveResultsFromFPGA('FSM1', image)
```

La section suivante (section 4.3.1), montre comment construire des machines à états pour les charger et exécuter sur le matériel reconfigurable.

4.3.1 Construction d'une machine à états simple

Dans [BSC+00], Banerjee *et al* proposent de créer une machine avec des états séparés pour l'initialisation, la vérification de la condition, le calcul du corps de boucle et un état à part pour la fin (voir diagramme d'états de la figure 20). Nous proposons une machine à états qui exécute une itération par cycle d'horloge (figure 21).

Pour des boucles avec des conditions simples comme les boucles de type `for`, il est peu probable que l'évaluation de la condition fasse partie du chemin

FIG. 20 – Diagramme de la machine à états pour une boucle avec état séparé pour la vérification de la condition

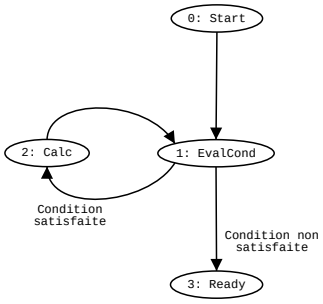
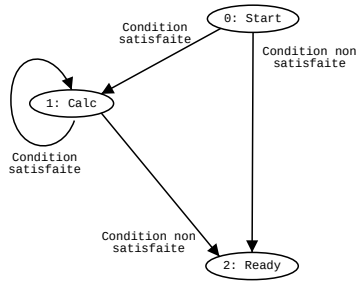


FIG. 21 – Diagramme de la machine à états pour une boucle avec un seul état pour la vérification de la condition et le calcul



critique. Dans ce cas, il n'y a pas de surcoût à évaluer la condition en même temps que les calculs du corps de boucle s'effectuent.

Le nouveau compilateur pourrait donner le choix de la méthode de construction des automates au développeur. En fonction des contraintes de la boucle à transformer et du matériel cible, il pourrait ainsi choisir le meilleur compromis entre performance et ressources reconfigurables.

Pour générer les configurations à l'aide des outils MADEO, notre compilateur génère du code SmallTalk qui décrit l'automate correspondant à la boucle transformée (figure 22). Ce code SmallTalk identifie les entrées, les sorties, les variables d'états et définie les fonctions combinatoires de transition et l'initialisation en début de l'exécution.

FIG. 22 – Code SmallTalk pour un automate

```
FSM1:
Input:            image_input
StateVariables:   state, col, y_1, y_2, image_loop
Output:           image := image_loop
Initialisation:   col := 0. y_1 := 0. y_2 := 0.
                  image_loop := image_input
Transition:
state := [state <= 1 ifTrue: [col < frame_width*inc ifTrue: 1
                                                    ifFalse: 2]
                     ifFalse: 2]
y     := [state == 1 ifTrue: [a1*pixel + b1*y_1 + b2*y_2]
                     ifFalse: y]
image := [state == 1 ifTrue: [image at: [lig*rs + col+1]
                                   put: [y].image]
                     ifFalse: image]
y_2   := [state == 1 ifTrue: y_1 ifFalse: y_2]
y_1   := [state == 1 ifTrue: y   ifFalse: y_1]
col   := [state == 1 ifTrue: col + 1 ifFalse: col]
```

Remarque : Pour certaines architectures cibles, l'initialisation pourrait se faire au moment de la configuration ou pendant l'exécution d'une première transition d'état de l'automate. L'exemple dans [BSC+00] initialise pendant la première transition, notre exemple SmallTalk part de l'hypothèse d'une initialisation pendant la phase de configuration pour garder notre exemple plus simple.

Comment peut le compilateur identifier les variables qui deviennent variables d'état ? Une première variable d'état est générée pour garder l'état d'avancement de l'exécution de la boucle. Puis, le compilateur en ajoute d'autres pour les variables qui transportent des valeurs d'une itération à l'autre. En utilisant les outils que nous met à disposition PIPS, le compilateur dispose de graphes de flux de données, ce qui permet d'identifier ces variables. Les autres variables, qui ne servent que comme valeur intermédiaire pendant les itérations, correspondent éventuellement à des connexions à l'intérieur de la fonction de transition si l'optimisation logique ne les fait pas disparaître.

4.3.2 Intégrer des instructions séquentielles qui précèdent la boucle

Il est possible de transformer des instructions séquentielles qui précèdent une boucle en fonction combinatoire et de synthétiser la logique correspondante (voir section 4.1). Si, comme dans l'exemple de la figure 23, on avance la directive `$PRAGMA_RECONFIGURABLE_BEGIN` pour inclure d'autres instructions, la logique correspondante serait naturellement intégrée dans l'automate et confondue avec la logique d'initialisation. L'exécution se ferait donc au moment du premier cycle d'horloge de l'automate.

FIG. 23 – Intégrer des calculs combinatoires qui précédent la boucle

```
$PRAGMA_RECONFIGURABLE_BEGIN
a1 = Carre( 1 - gamma )
b1 = 2 * gamma
b2 = -Carre( gamma )
y_1 = 0
y_2 = 0
col = 0
DO WHILE (col < frame_width*inc)
  ! ...
```

Le placement exact de la directive `$PRAGMA_RECONFIGURABLE_BEGIN` détermine la limite entre les instructions à exécuter sur le processeur séquentiel et sur le matériel reconfigurable. Tout calcul résultat des calculs précédant la limite est transmis vers la logique reconfigurable. Notre compilateur utilise les graphes de flux de données élaborés par PIPS pour détecter les variables concernées.

Dans le cas extrême, il n'y a aucun calcul d'initialisation à faire en logique reconfigurable par l'automate. Si ceci est le cas et si on utilise un diagramme d'état avec des états séparés pour l'initialisation et la vérification de la condition de boucle, il est possible de supprimer l'état d'initialisation entièrement. On peut réduire ainsi les temps de latence d'un cycle d'horloge.

4.3.3 Intégrer des instructions séquentielles qui suivent la boucle

De la même manière qu'on intègre de la logique combinatoire dans l'état d'initialisation, on peut en intégrer dans le dernier état. Au niveau du code source de haut niveau, ceci correspond à des calculs qui suivent directement la boucle.

Pour marquer les calculs qui précédent la boucle, il suffit de déplacer la directive `$PRAGMA_RECONFIGURABLE_BEGIN`. Si le développeur veut de la même manière inclure des instructions qui suivent directement la boucle, il doit déplacer la directive
`$PRAGMA_RECONFIGURABLE_END` (figure 24).

FIG. 24 – Intégrer des calculs combinatoires qui suivent la boucle

```
$PRAGMA_RECONFIGURABLE_BEGIN
DO WHILE (col < frame_width*inc)
  ! ...
END DO
! Instructions supplémentaires pour logique combinatoire
! à exécuter pendant l'état "READY"
$PRAGMA_RECONFIGURABLE_END
```

4.3.4 Boucles successives, logique combinatoire avec registres avant et après la boucle

Deux boucles successives peuvent s'exécuter sur le matériel reconfigurable sans rendre le contrôle au processeur séquentiel. Pour obtenir ceci, le compilateur doit fusionner les automates de contrôle des deux automates et générer un seul automate. Le résultat est un seul automate de contrôle et une seule configuration.

En ce qui concerne les directives `PRAGMA_CLOCK`, il est possible d'intégrer les automates de contrôle associés à de la logique combinatoire avec des registres (section 4.1.4). Ainsi il devient possible de déporter sur le matériel reconfigurable l'exécution de successions de boucles et le code entre ces boucles.

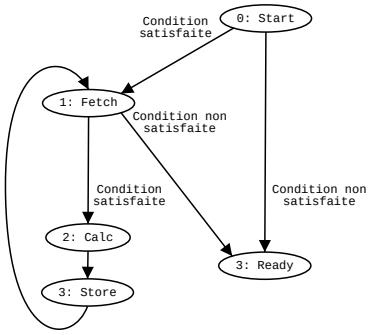
4.3.5 Automate avec lecture d'une mémoire externe

La boucle de l'exemple de la figure 18 contient des accès au champ de données nommé `"image"` (`pixel = image(lig*rs + col+1)` et `image(lig*rs + col+1) = y`). Dans l'automate que nous avons construit dans la section précédente, nous partons de l'hypothèse que le champ entier est chargé dans des registres de la logique reconfigurable. En pratique, la taille des images devait dépasser les ressources de registres disponibles. Pour la lecture et l'écriture des ces mémoires, il faut donc prévoir des accès externes, sans doute plus coûteux en temps. En reprenant une autre idée de [BSC+00], nous ajoutons des états supplémentaires pour effectuer ces lectures/écritures (figure 25). En fait, il faut ajouter un état supplémentaire pour chaque écriture ou lecture. Cet état supplémentaire pourrait servir aussi à effectuer des éventuels cycles d'attente si la mémoire n'est pas assez rapide.

4.3.6 Ajouter des étapes supplémentaires dans les instructions du corps de boucle.

Si le corps de boucle contient des directives `PRAGMA_CLOCK`, le compilateur peut les traiter en ajoutant des états supplémentaires au graphe de contrôle et en générant les registres correspondants. Pendant l'exécution de la boucle par le matériel reconfigurable, chaque itération dure un cycle d'horloge supplémentaire par directive `PRAGMA_CLOCK` ajoutée.

FIG. 25 – Diagramme d'états avec des états supplémentaires pour la lecture d'une mémoire externe



4.3.7 Boucles imbriquées

Pour pouvoir déporter des boucles imbriquées sur le matériel reconfigurable, on génère un automate de contrôle pour la boucle extérieure suivant les principes décrites dans les sections à partir de 4.3.1. Puis, il faut intégrer les états de l'automate de la boucle intérieure à l'endroit qui correspond à l'emplacement dans le corps de la boucle extérieure. De manière récursive, on peut remonter les niveaux de boucles en construisant l'automate de contrôle.

4.3.8 Dérouler des boucles

En utilisant les transformations existants de PIPS, notre compilateur interactif peut dérouler des boucles partiellement ou entièrement. Le programmeur peut demander de façon interactive cette opération. La figure 26 montre le résultat du déroulage avec un facteur 2.

FIG. 26 – Dérouler une boucle partiellement (facteur 2)

```
DO WHILE (col < frame_width*inc)
  pixel = image(lig*rs + col+1)
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  y_2 = y_1
  y_1 = y
  image(lig*rs + col+1) = y
  col = col + inc

  pixel = image(lig*rs + col+1)
  y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
  y_2 = y_1
  y_1 = y
  image(lig*rs + col+1) = y
  col = col + inc
END DO
```

Pour pouvoir transformer le corps de boucle facilement en fonction de logique

combinatoire nécessaire pour la génération d'automate, il faut transformer le corps de boucle en forme d'assignation unique (voir figure 27). Les variables qui apparaissent plus qu'une seule fois du coté gauche d'une assignation dans le segment de code sont renommées de façon à garder une assignation unique.

FIG. 27 – Transformation en forme d'assignation unique

```
pixel_h1 = image(lig*rs + col + 1)
y_h1 = (TT_type)(a1*pixel_h1 + b1*y_1 + b2*y_2)
y_2_h1 = y_1
y_1_h1 = y_h1
image(lig*rs + col+1) = pixel_h2
col_h1 = col + inc

pixel = image(lig*rs + col_h1 + 1)
y = (TT_type)(a1*pixel_h2 + b1*y_1_h1 + b2*y_2_h1)
y_2 = y_1_h1
y_1 = y
image(lig*rs + col_h1+1) = y
col = col_h1 + inc
```

PIPS met déjà à disposition la transformation en forme de d'assignation unique. Dans le cas général, il faut l'appliquer sur le corps de toute boucle traduit en automate.

On peut utiliser ceci pour faire un automate qui fait n fois plus de calculs par cycle d'horloge. Dans la section qui suit, nous montrons la méthode inverse qui permet de diviser un calcul en plusieurs cycles d'horloge.

Une fois le code transformé comme montré, on peut utiliser les directives `PRAGMA_CLOCK` et `PRAGMA_DO_PARALLEL`. Ceci permet de contrôler le degré de parallélisme et définir exactement quelle opération s'exécute pendant quel cycle d'horloge.

4.3.9 Exécution parallèle de toutes les itérations d'une boucle

La section précédente montre comment dérouler une boucle et paralléliser plusieurs itérations. Si le nombre d'itérations est connu à la compilation, on peut dérouler la totalité des itérations, faire ainsi disparaître la boucle et puis exécuter le code de chacune des itérations en parallèle avec la directive `PRAGMA_DO_PARALLEL`. Pour tout nombre d'itérations qui dépasse 2 ou 4, cette méthode rend le code source difficilement intelligible.

En HPF (*High Performance Fortran*) [Hig97], il y a la directive `INDEPENDENT` qui permet de spécifier qu'il est possible d'exécuter les itérations d'une boucle suivante sans ordre fixe ou en parallèle. De la même manière, on peut imaginer pour notre compilateur une directive `PRAGMA_LOOP_PARALLEL` qui serait équivalent à la boucle entièrement déroulée avec un groupe parallèle par itération.

4.4 Exécution conditionnelle

De la même manière qu'on construit un graphe de contrôle pour une boucle (section 4.3.1), il est possible d'en construire un qui traite les structures d'exécution conditionnelles des langages de haut niveau (`IF THEN ELSE`, `CASE` etc.). Par exemple, on pourrait évaluer la condition pendant un premier état. Puis l'exécution continuerait avec un état ou plusieurs états correspondants à la branche sélectionnée.

Alternativement, dans la recherche d'autres compromis performance / ressources, il est possible d'exécuter plusieurs branches à la fois en parallèle du calcul de la condition. Une fois la condition connue, on sélectionne le résultat de la branche correspondante. Dans un cas extrême, le tout peut se faire dans un seul cycle d'horloge et donc sans registre en forme de logique combinatoire (prédication), voir par exemple [CHW00]).

En ce qui concerne les directives pour notre compilateur, il n'est pas nécessaire d'en ajouter des nouvelles. Le compilateur peut être adapté pour traiter les directives `PRAGMA_RECONFIGURABLE` et `PRAGMA_CLOCK`, même si elles incluent des instructions à exécution conditionnelle.

Les `IF THEN ELSE` et `CASE` généreraient des états supplémentaires dans les automates de contrôle. Si le programmateur veut éviter des états et les cycles d'horloge associés, il peut exprimer la prédication dans le code source. PIPS pourrait effectuer automatiquement cette transformation à la demande.

4.5 IF et GOTO comme unique syntaxe de contrôle

Il est possible de simplifier la synthèse de configurations pour le matériel reconfigurable en se limitant aux uniques structures de contrôle `IF` et `GOTO`. Ceci représente une alternative à l'utilisation des boucles comme elle est présentée dans ce document.

PIPS transformerait alors toute les boucles, procédures récursives, `CASE` *etc.* en utilisations de `IF` et de `GOTO`. Puis, PIPS générerait des automates de contrôle de la même manière qu'il en était fait pour les boucles.

5 Transformations avancées

5.1 Pipelining

Le *pipelining* est une technique de parallélisation couramment utilisée dans les architectures *hardware*. Dans ce paragraphe, nous montrons, que les directives de compilation que nous avons introduit au cours de la section 4 sont suffisantes pour exprimer le *pipelining* en langage de haut niveau.

On utilise le *pipelining* dans les cas où il y a une série de traitements identiques à exécuter. Le *pipelining* peut éviter de dédoubler des unités fonctionnelles ou résoudre certain cas de dépendance entre traitements successif. Regardons l'exemple de la figure 28, qui exécute pour chaque itération les opérations `A` et puis `B`.

On peut reformuler cette boucle comme c'est montré dans la figure 29. Ainsi, les deux calculs `A` et `B` sont séparés sur deux lignes différentes. La variable `tmp` contient le résultat intermédiaire.

FIG. 28 – Boucle sans *pipelining*

```
i = 1
DO WHILE (i <= max)
  OUT(i) = B(A(IN(i)))
  i=i+1
END DO
```

FIG. 29 – Boucle sans *pipelining* avec variable temporaire

```
i = 1
DO WHILE (i <= max)
  tmp    = A(IN(i))
  OUT(i) = B(tmp)
  i=i+1
END DO
```

Maintenant, il est possible d'exécuter en parallèle dans la boucle l'opération `B` du pas `i` et l'opération `A` du pas `i-1`. En début et fin de boucle, il faut exécuter le calcul `A` du premier pas tout seul (prologue). De même, il faut exécuter le calcul `B` du dernier pas (épilogue). La figure 30 montre le résultat.

FIG. 30 – Pipelining

```
tmp = A(IN(1))
i = 2
DO WHILE (i <= max)
  $PRAGMA_DO_PARALLEL_BEGIN
  $PRAGMA_PARALLEL_GROUP
  OUT(i) = B(tmp)
  $PRAGMA_PARALLEL_GROUP
  tmp    = A(IN(i))
  $PRAGMA_PARALLEL_GROUP
  i=i+1
  $PRAGMA_DO_PARALLEL_END
END DO
OUT = B(tmp)
```

Si maintenant, on encadre le code la figure 30 par les directives `PRAGMA_RECONFIGURABLE_BEGIN` et `PRAGMA_RECONFIGURABLE_END`, et on fait traduire le code source par notre compilateur, on reçoit une configuration de matériel reconfigurable qui correspond à un *pipeline*. Le prologue exprime la phase de remplissage du *pipeline* et l'épilogue le *pipeline* qui se vide.

Comme nous l'avons décrit au cours de la section 4, le compilateur générerait les fonctions combinatoires en double, une fois pour le prologue / épilogue et une fois pour le corps de boucle. Pour éviter ce problème, nous intégrons l'épilogue

et le prologue par prédication dans le corps de boucle (voir figure 31).

FIG. 31 – Pipelining avec intégration de prologue et épilogue

```
i = 1
DO WHILE (i <= max + 1)
  $PRAGMA_DO_PARALLEL_BEGIN
  $PRAGMA_PARALLEL_GROUP
  IF i > 1
    OUT(i) = B((tmp)
  $PRAGMA_PARALLEL_GROUP
  IF i <= max
    tmp    = A(IN(i))
  $PRAGMA_PARALLEL_GROUP
  i=i+1
  $PRAGMA_DO_PARALLEL_END
END DO
```

La figure 32 applique cette technique à notre exemple du filtre de Deriche. Le *pipeline* représenté consiste de deux étages, le premier charge la donnée et exécute le calcul, le deuxième écrit le résultat dans la mémoire.

FIG. 32 – Exemple *pipelining*

```
col = 0
DO WHILE (col < frame_width*(inc+1))
  $PRAGMA_DO_PARALLEL_BEGIN
  $PRAGMA_PARALLEL_GROUP
  IF col > 0
    image(lig*rs + col+1-inc) = y
  $PRAGMA_PARALLEL_GROUP
  IF col < frame_width*inc
    pixel = image(lig*rs + col+1)
    y = (TT_type)(a1*pixel + b1*y_1 + b2*y_2)
    y_2 = y_1
    y_1 = y
  $PRAGMA_PARALLEL_GROUP
  col = col + inc
  $PRAGMA_DO_PARALLEL_END
END DO
```

Il est bien sur de faire de la même manière des *pipelines* de plus d'étages. Dans l'exemple montré on pourrait par exemple séparer le calcul d'adresse ou séparer la lecture du calcul arithmétique dans le premier étage du *pipeline*.

Les transformations de code source pour obtenir un *pipeline* ne doivent pas forcement se faire à la main. PIPS sait déjà faire de l'ordonnancement pour faire du *software pipelining*. Cette opération est exactement la même qu'on a besoin

pour faire du *hardware pipelining*. Il suffirait d'ajouter à la transformation PIPS correspondante l'insertion des directives présentées dans cette section.

5.2 Récursivité

Pour pouvoir utiliser la récursivité dans le code source des langages de haut niveau, nous proposons de transformer son utilisation automatiquement en itérations. Cette transformation ne devait pas être difficile à intégrer dans les outils PIPS existants.

6 Réflexions sur l'architecture et autres problèmes ouverts

Les performances du résultat d'une compilation de langage de haut niveau vers une réalisation par du matériel dépend de l'architecture des modules d'exécution. L'architecture de ces modules étant défini de manière relativement floue dans le projet PHRASE, nous listons dans cette section un certain nombre des alternatives architecturales, desquelles il faut choisir avant de synthétiser.

6.1 Architectures de mémoire

Les performances du matériel reconfigurable dépendent des ressources de calcul et de la capacité d'acheminer les données vers les ressources de calcul. La méthode que nous décrivons dans ce document permet de contrôler les ressources de calcul assez finement. En ce qui concerne l'acheminement des données, les réflexions ne sont pas très avancées.

La capacité d'acheminer des données dépend fortement de l'architecture cible. Une architecture mémoire pourrait par exemple consister de :

- bascules reparties sur les architectures reconfigurable de grain fin
- mémoires RAM spécialisées comme ils sont inclus dans certaines architectures de FPGA commerciaux
- mémoires externes associées aux composants reconfigurables
- séquenceurs rapides spécialisés pour adresser des données consécutives
- plusieurs bancs de mémoire accessibles en parallèle
- tout mélange des techniques listées ici

Est-ce que notre manière d'abstraire les accès à la mémoire est suffisante pour modéliser les contraintes de stockage ? Est-ce qu'elle est assez universelle pour représenter toutes les architectures de mémoire utiles ?

6.2 Multiples processeurs séquentiels, multiples processeurs reconfigurables

Nous n'avons pas encore traité le problème de l'utilisation de plusieurs processeurs séquentiels et de plusieurs processeurs reconfigurables. Est-ce qu'il est suffisant d'utiliser les mécanismes existants de PIPS pour exploiter plusieurs processeurs séquentiels en combinaison avec des modules reconfigurables ? Est-ce qu'il est suffisant de modéliser plusieurs processeurs reconfigurables comme un seul processeur avec d'avantage de ressources ?

6.3 Adaptation des fréquences d'horloge à la complexité de la logique combinatoire

La fréquence d'horloge appliquée au matériel reconfigurable est un des paramètres à définir au cours de la conception. L'architecture peut supporter le choix de la fréquence de différentes manières :

- fixer la fréquence une fois pour toute
- mettre à disposition un ou plusieurs générateurs d'horloge programmables pour pouvoir associer des fréquences différentes à des configuration différentes

6.4 Transmission des types vers MADEO

MADEO [LLFP01] nécessite en outre des descriptions de fonctions combinatoires et d'automates une spécification précise des types de données utilisés. MADEO est spécialisé aux types creux et sait avant tout générer de la logique optimisé pour ces cas.

Nous n'avons pas encore spécifié les sorties de types vers MADEO. On peut imaginer de transmettre l'équivalent des types de langage haut niveau sans prendre en compte que certaines valeurs ne soient jamais utilisées. Est-ce que PIPS peut déduire des informations plus précises ?

7 Conclusion

Nous avons décrit une méthode de transformer un logiciel partiellement d'un langage haut niveau en vue d'une exécution partagée sur un processeur séquentiel et un ou plusieurs noeuds de matériel reconfigurable. Les transformations se font guidées par le programmeur, par des directives dans le code source. Le programmeur peut éventuellement ajouter ces directives de manière interactive en explorant l'espace des solutions.

Les différentes directives permettent de transposer tous les éléments syntaxiques du langage de haut niveau. Il est ainsi possible de transformer une partie du programme d'origine de taille variable pour exécution sur un ou plusieurs noeuds reconfigurable. Le programmeur peut faire varier la portion à déporter en recherchant à satisfaire les contraintes de ressources et de performance.

Puis, dans le même but de satisfaction des contraintes, le programmeur transforme le code source pour obtenir des ordonnancements différents. Il peut ajouter des registres supplémentaires pour couper les chemins critiques, ou au contraire dérouler des boucles pour faire plus de calcul pendant un même cycle d'horloge.

Une dernière étape de notre compilateur fondé sur PIPS interprète les directives et sépare le code en deux parties, une pour le processeur séquentiel et une pour le matériel reconfigurable. Le logiciel pour le processeur séquentiel est augmenté d'appels à méthodes d'une librairie de communication pour interagir avec le matériel reconfigurable. Le logiciel pour le matériel reconfigurable est traité par un synthétiseur, ici MADEO, qui génère des configurations qui exécutent les parties déportées.

Le programme source augmenté des directives décrit alors quelle partie sera déportée sur le matériel reconfigurable. Pour la partie déportée, les directives

définissent en plus le moment d'exécution au cycle d'horloge prêt.

Références

[AAB+97] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, M. Srikrishna, and D. Taylor. The raw compiler project, 1997.

[BSC+00] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Proceedings of the IEEE International Symposium on FPGA Custom Computing Machines*, pages 39–48, April 2000.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and Andre DeHon. Stream computations organized for reconfigurable execution (SCORE). In *FPL*, pages 605–614, 2000.

[CHW00] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4) :62–69, April 2000.

[CM99] François Charot and Vincent Messé. A flexible code generation framework for the design of application specific programmable processors. In *Proceedings of the seventh international workshop on Hardware/Software Codesign, CODES'99*, pages 27–31, Rome, Italy, Mar 1999.

[DCPS02] Raphaël David, Daniel Chillet, Sébastien Pillement, and Olivier Sentieys. Flot de conception pour plateforme reconfigurable. In *3ème Colloque CAO de circuits et systèmes intégrés*, Mai 2002.

[DGG02] Rainer Dömer, Daniel D. Gajski, and Andreas Gerstlauer. Specc methodology for high-level modeling. In *9th IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, Oct 2002. There is a version is dated August 9, 2002 and corrects some errors that appear in the proceedings of ASPLOS (updated results, plus two new references).

[Hig97] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, Houston, Tex., 1997.

[HNK+] M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. A library-based compiler to execute matlab programs on a heterogeneous platform.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : an overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM Press, 1991.

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Matson, and John D.Owens. Programmable stream processors. *IEEE Computer*, 36(8) :54–62, August 2003.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[LLFP01] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier. Placing, routing and editing virtual FPGAs. In *FPL'O1*, Aug 2001.

[NBD+03] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8) :63–69, August 2003.

[Pan01] Preeti Ranjan Panda. SystemC. In *ISSS*, pages 75–80, 2001.

[PS02] Pierre G. Paulin and Miguel Santana. A retargetable embedded-software development environment. *IEEE Design & Test of Computers*, 19(4) :59–69, Apr 2002.

[RL02] F. Raimbault and D. Lavenier. ROOM : des machines reconfigurables orientées objets. In *Proceedings of the 8th Symposium en Architectures nouvelles de machines, SympA 2002*, 2002.

[sys] Systemc community website, http ://www.systemc.org/.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt : A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, Grenoble, France, 2002.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor : A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2) :25–35, Mar-Apr 2002.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF : An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12) :31–37, 1994.