Conception de systèmes programmables basés sur les NoC par synthèse de haut niveau : analyse symbolique et contrôle distribué

Matthieu Payet


UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

N° d'ordre NNT :

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée au sein de

l'université Jean Monnet de Saint-Étienne

École doctorale EDSIS 488
Sciences Ingénierie et Santé

Spécialité : micro-électronique

Soutenue publiquement le 26/10/2016, par :

Matthieu Payet

Conception de systèmes programmables basés sur les NoC par synthèse de haut niveau : analyse symbolique et contrôle distribué

Devant le jury composé de :

Loic Lagadec
Professeur, Laboratoire Lab-STICC / ENSTA Bretagne, président
Amer Baghdadi
Professeur, Laboratoire Lab-STICC / Telecom Bretagne, rapporteur
Camel Tanougast
Professeur, Laboratoire LCOMS / Université de Lorraine, rapporteur
Pascal Remy
Docteur, Adacsys, examinateur
Frédéric Rousseau
Professeur, Laboratoire TIMA / Polytech'Grenoble, directeur de thèse
Virginie Fresse
Maître de conférence, Laboratoire Hubert Curien / Telecom Saint-Étienne, co-directrice de thèse

Abstract

Network-on-Chip (NoC) introduces parallelism in communications and emerges with the growing integration of circuits as large designs need scalable communication architectures. This introduces the separation between communication tasks and processing tasks, and makes the design with NoC more complex. High level synthesis (HLS) tools can help designers to quickly generate high quality HDL (Hardware Description Level) designs. But their control schemes are centralized, usually using finite state machines. To take benefit from parallel algorithms and the ever growing FPGAs, HLS tools must properly extract the parallelism from the input representation and use the available resources efficiently. Algorithm designers are used with programming languages. This behavioral specification has to be enriched with architectural details for a correct optimization of the generated design. The C to FPGA path is not straightforward, and the need for architectural knowledges limits the adoption of FPGAs, and more generally, parallel architecture. In this thesis, we present a method that uses a symbolic analysis technique to extract the parallelism of an algorithmic specification written in a high level language. Parallelization skills are not required from the users. A methodology is then proposed for adding NoCs in the automatic design generation that takes the benefit of potential parallelizations. To dimension the design, we estimate the design resource consumption using a mathematical model for the NoC. A scalable application, hardware specific, is then generated using a High Level Synthesis flow. We provide a distributed mechanism for data path reconfiguration that allows different applications to run on the same set of processing elements. Thus, the output design is programmable and has a processor-less distributed control. This approach of using NoCs enables us to automatically design generic architectures that can be used on FPGA servers for High Performance Reconfigurable Computing. The generated design is programmable. This enable users to avoid the logic synthesis step when modifying the algorithm if a existing design provide the needed operators.

Résumé

Les réseaux sur puce (NoC pour « network on chip ») sont des infrastructures de communication extensibles qui autorisent le parallélisme dans la communication. La conception de circuits basés sur les NoC se fait en considérant la communication et le calcul séparément, ce qui la rend plus complexe. Les outils de synthèse d'architecture (HLS pour « high level synthesis ») permettent de générer rapidement des circuits performants. Mais le contrôle de ces circuits est centralisé et la communication est de type point-à-point (non extensible). Afin d'exploiter le parallélisme potentiel des algorithmes sur des FPGA dont les ressources augmentent constamment, les outils de HLS doivent extraire le parallélisme d'un programme et utiliser les ressources disponibles de manière optimisée. Si certains outils de synthèse considèrent une spécification de type flot de données, la plupart de concepteurs d'algorithmes utilise des programmes pour spécifier leurs algorithmes. Mais cette représentation comportementale doit souvent être enrichie d'annotations architecturales afin de produire en sortie un circuit optimisé. De plus, une solution complète d'accélération nécessite une intégration du circuit dans un environnement de développement, comme les GPU aujourd'hui. Un frein à l'adoption des FPGA et plus généralement des architectures parallèles, est la nécessaire connaissance des architectures matérielles ciblées. Dans cette thèse, nous présentons une méthode de synthèse qui utilise une technique d'analyse symbolique pour extraire le parallélisme d'une spécification algorithmique écrite dans un langage de haut niveau. Cette méthode introduit la synthèse de NoC pendant la synthèse d'architecture. Afin de dimensionner le circuit final, une modélisation mathématique du NoC est proposée afin d'estimer la consommation en ressources du circuit final. L'architecture générée est extensible et de type flot de données. Mais l'atout principal de l'architecture générée est son aspect programmable car elle permet, dans une certaine mesure, d'éviter les synthèses logiques pour modifier l'application.

Remerciements

Un travail de doctorat consiste d'abord à l'apprentissage d'une manière de penser, de la critique tant sur la forme que sur le fond et de l'esprit scientifique. Il est évident que l'esprit doit être forgé par un entourage, une équipe, des maîtres attentionnés et qualifiés.

Il me tient à cœur donc de remercier les personnes qui ont participé à ces travaux de thèse, que ce soit par leur implication formelle ou informelle.

En premier lieu, je remercie mon directeur de thèse Frédéric Rousseau pour la rigueur de ses critiques, ses relectures qui ont apporté plus de clarté et de concision aux diverses productions écrites émanant de cette thèse. J'aimerais aussi le remercier pour sa confiance.

Je souhaite exprimer mes plus vifs remerciements à ma co-directrice de thèse Mme Virginie Fresse pour son soutien indéfectible dans mon projet de recherche, ses encouragements dans les moments les plus difficiles, sa confiance et sa disponibilité.

Je souhaite également remercier mon tuteur au sein de l'entreprise Adacsys, Pascal Remy, qui a toujours été disponible, bon conseillé et d'une aide inestimable dans ses méticuleuses relectures.

J'exprime mes remerciements aux autres membres de mon jury : Loic Lagadec, Amer Baghdadi et Camel Tanougast qui ont montré de l'intérêt pour mon travail et qui m'ont accordé de leur temps pour ma soutenance.

Je remercie tous les doctorants, stagiaires et ingénieurs que j'ai pu côtoyer dans les différents laboratoires où j'ai travaillé pour leurs aides, conseils et soutien, en particulier Adrien Prost-Boucle, Hatem Belhassen, Atef Dorai, Valentin Mena-Morales, Otavio Alcantara, Danmei Chen et Lina Shi.

Enfin, ce travail n'aurait pas été possible sans le soutien de l'entreprise Adacsys qui m'a permis de me consacrer à l'élaboration de ma thèse.

Table des figures

Liste des tableaux

Table des matières

Chapitre 1 : Introduction

L'art n'est pas d'arriver avec des idées neuves mais d'interpréter ces idées qui nous entourent depuis toujours.

George Lucas

Sommaire

1.1 Contexte général

De nombreux domaines d'application nécessitent des performances de calcul toujours plus accrues. Citons par exemple, le calcul financier (évaluation d'options), l'analyse de données (le « big data »), certaines applications scientifiques (simulations physiques, bio-informatique), le chiffrement et le traitement multimédia.

Afin de satisfaire ces besoins, des machines particulièrement performantes sont généralement utilisées. C'est ce qu'on appelle le calcul haute performance (HPC pour « High Performance Computing). Ces machines sont traditionnellement des systèmes à base de processeurs généralistes. Ces *super-calculateurs* permettent d'exécuter les mêmes programmes que sur un ordinateur standard

mais avec de meilleures performances, pour peu que le programmeur sache l'utiliser de manière efficiente.

Or la plupart des utilisateurs de ces machines sont des spécialistes dans leurs domaines de compétences respectifs (imagerie, finance, etc) mais pas des architectures matérielles qu'ils utilisent. Nous les appelons, dans la suite de ce mémoire de thèse, *les concepteurs d'algorithmes*. De ce fait, l'utilisation du matériel est souvent partielle et loin de l'optimum.

L'évolution des architectures matérielles a rencontré des limites physiques qui ont changé la manière d'améliorer les performances. En effet, la puissance dissipée (P_d) par les circuits est proportionnelle à leur fréquence (F) de fonctionnement (voir équation 1.1 pour une technologie CMOS) à tension d'alimentation constante (V_{dd}). Dans la poursuite de la loi de Moore, les dimensions des transistors ont été réduites jusqu'à atteindre des dimensions nanométriques. Jusqu'aux années 2005-2007, la réduction de la taille des transistors était associée à la réduction de la tension et du courant d'alimentation. Donc, à puissance égale, les circuits pouvaient fonctionner à des fréquences plus élevées, d'où des performances plus grandes à chaque génération de transistors (principe appelé « Dennard scaling »).

$$P_d = P_{stat} + \alpha \sum C_i \times V_{dd}^2 \times F \tag{1.1}$$

Bien que la puissance statique ait été négligée pendant longtemps, à l'échelle nanométrique des transistors, elle devient importante. En effet, les courants de fuite augmentent à mesure que la taille des transistors se réduit. Limiter la dissipation thermique nécessite de limiter la consommation dynamique en limitant la fréquence de fonctionnement des circuits.

C'est ce qu'on observe depuis plusieurs années. Les fréquences de fonctionnement des CPU atteignent des plafonds de fréquences (autour de 4GHz) [Moo11], voir figure 1.1. L'augmentation des performances passe désormais et nécessairement par l'exploitation du parallélisme disponible à la fois au niveau de l'*algorithme* et au niveau de l'*architecture matérielle*.

Bien avant d'atteindre ces limites, le parallélisme avait déjà été introduit de manière plus ou moins transparente dans les architectures de type Von Neuman (« pipelines », jeux d'instructions complexes, processeurs vectoriels, exécution des instructions dans le désordre, etc.). Il s'agissait du parallélisme au niveau des instructions. Les compilateurs ont alors été optimisés pour utiliser de manière efficiente ces architectures. Puis, le parallélisme au niveau des »threads« et l'utilisation d'accélérateurs matériels dédiés ont nécessité des descriptions qui permettent de rendre explicite le parallélisme dans les programmes.

La manière d'écrire les programmes influence directement les performances, notamment la manière d'accéder à la mémoire. L'ajout de directives (comme avec OpenMP) permet, par exemple, d'indiquer comment mieux exploiter le parallélisme d'un programme. Mais, surtout, de nouveaux modèles de programmation permettent à un utilisateur d'indiquer explicitement comment exploiter

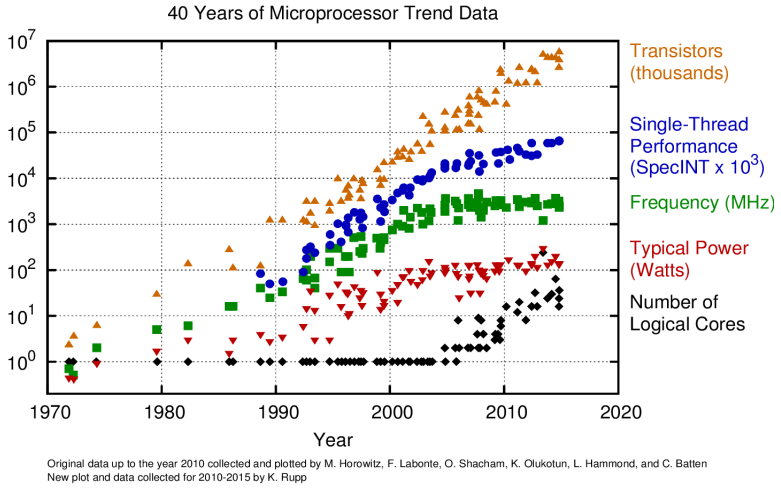


FIGURE 1.1 : 40 années de données sur les processeurs (mise à jour par K. Rupp)

une architecture pour une application donnée. C'est de cette manière que des architectures parallèles, dites super-scalaires, dédiés au traitement d'image, ont été utilisées avec succès, dans d'autres domaines d'application.

Les GPGPU (« general-purpose computing on graphics processing unit ») nécessitent des modèles de programmation différents (OpenCL, CUDA) qui permettent d'indiquer comment sont répartis les calculs, de synchroniser les opérations et de manipuler les données dans les différents types de mémoires disponibles.

Écrire un programme optimisé pour un processeur graphique nécessite, donc, de connaître son architecture. Cela est aussi vrai, de manière générale, pour toute architecture parallèle. Dans un premier temps, le concepteur d'algorithmes doit mettre en évidence le *parallélisme potentiel* de son programme. Dans un deuxième temps, il doit *répartir* les calculs sur les opérateurs disponibles de l'architecture matérielle (le GPU). Les concepteurs d'algorithmes n'ont pas forcement l'expertise nécessaire à l'utilisation de ces architectures. Elles nécessitent un apprentissage et les programmes développés doivent être adaptés pour chaque matériel.

Plus récemment, les architectures matérielles reconfigurables, comme les FPGA, ont montré qu'ils pouvaient être compétitifs dans le monde du HPC [PTD13, CA07, HVG+07, CA07, IBS20] de même que dans les systèmes embarqués.

Les FPGA sont des circuits reconfigurables qui permettent d'implémenter des architectures matérielles dédiées pour réaliser un calcul. Ils consomment relativement peu (de l'ordre du Watt) comparé aux GPU (plusieurs dizaines à centaines de watt). Sans compter que pour chaque Watt consommé par les architectures standards (CPU et GPU), 0,5W à 1W doivent être consommés en plus pour le refroidissement [PBS+03].

Les architectures matérielles peuvent être classées en 3 grands types, allant des plus performantes et spécialisées, aux moins performantes mais généralistes :

les ASIC, les FPGA et les processeurs. Le GPU est un processeur spécialisé dans le traitement graphique. Sa puissance de calcul s'est avérée intéressante pour l'accélération d'applications généralistes d'où la dénomination de calcul GPGPU pour « general-purpose processing on graphical processing unit ». Dans ce mémoire de thèse, nous étudierons les architectures matérielles basées sur des FPGA. Les FPGA combinent la flexibilité (reconfiguration) et la spécialisation (circuit dédié). C'est la raison pour laquelle son utilisation pour le calcul haute performance (HPRC pour « high performance reconfigurable computing ») se développe de plus en plus. En outre, comme le comportement des FPGA – et le langage HDL (Hardware Description Langage) utilisé – est proche de celui des ASIC, il est aussi utilisé pour faire du prototypage.

Un FPGA permet de réaliser des applications hautement parallèles (la seule limite étant la taille du FPGA) avec des opérateurs dédiés (calcul en virgule fixe par exemple). Pour des applications spécifiques, les FPGA ont montré qu'ils pouvaient être compétitifs par rapports aux GPU [OMVEC13, ZSJ+09].

Par ailleurs, un circuit dédié ne consomme que l'énergie dont il a besoin pour réaliser un calcul. Contrairement à un GPU, qui est généraliste, un FPGA consomme moins pour réaliser le même calcul [KDW10]. Mais les fréquences de fonctionnement des processeurs généralistes sont environs vingt fois plus grandes que celles des circuits FPGA. Ces derniers compensent leur lenteur par un niveau de parallélisme maximal : des mémoires distribuées auprès des nombreux opérateurs dédiés. En effet, les circuits dédiés bénéficient des trois types de parallélisme (de tâche, de données, de flux). Ils sont, par conséquent, plus efficients à faible fréquence [DD11].

Les FPGA peuvent être associés au CPU, de la même manière que les GPU, pour augmenter les performances de certaines applications (voir figure 1.2). Cette discipline est appelée « reconfigurable computing » [TCW+05] littéralement l'informatique reconfigurable.

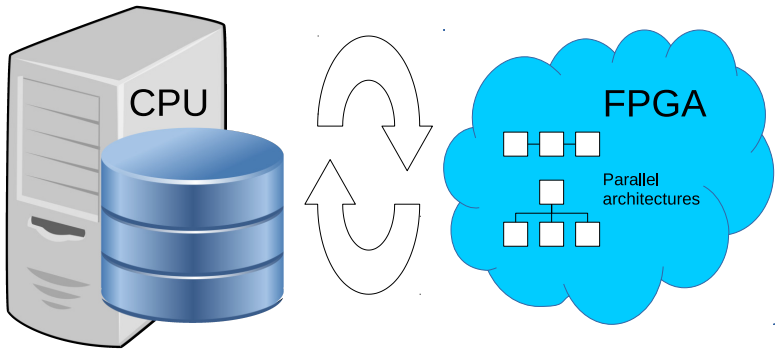


FIGURE 1.2 : HPRC : architectures parallèles sur circuits reconfigurable utilisés comme co-processeurs

Récemment, Intel a fait l'acquisition d'Altera (décembre 2015), ce qui ouvre des perspectives pour le HPC à base de FPGA. De manière générale, il y a une tendance qui vise à utiliser les FPGA pour accélérer des calculs à coté d'un

CPU principal. C'est ce qui « pousse les industriels concepteurs de CPU et de systèmes d'exploitation à définir des normes pour communiquer avec les accélérateurs sur FPGA » (Jon Masters - architecte ARM chez Red Hat) [Red16]. En effet, adopter les FPGA comme co-processeurs n'est pas une tâche aisée, que ce soit pour la conception des applications ou pour l'intégration du FPGA dans une plateforme dédiée à l'accélération de calcul.

1.2 Problématique

Malgré les avantages que présentent les architectures reconfigurables, la mise en œuvre d'applications sur FPGA est très coûteuse en temps de développement et de vérification. Typiquement, la validation d'une application FPGA (à partir d'un modèle VHDL) est de l'ordre de plusieurs mois.

De plus, en fonction du domaine d'application (finance, « big data », etc.), les algorithmes peuvent évoluer régulièrement selon les besoins de l'utilisateur. À chaque changement, le concepteur matériel doit, à nouveau, valider son architecture. Or, le cycle de développement d'une application à base de FPGA est trop long et le niveau d'expertise nécessaire est trop élevé pour démocratiser l'utilisation des FPGA dans le domaine de l'informatique reconfigurable. La conception sur FPGA est encore, aujourd'hui, réservée à des spécialistes. Il y a donc un fossé entre les utilisateurs et les concepteurs qu'il est nécessaire de combler.

Les outils actuels de synthèse sur FPGA permettent déjà de générer des accélérateurs par synthèse d'architecture au niveau système (HLS pour « High Level Synthesis »). Un concepteur d'algorithmes peut spécifier une version de son programme, mais doit d'abord l'optimiser pour la synthèse et ensuite faire communiquer le circuit généré avec le processeur hôte. Or, d'une part, notre objectif n'est pas de générer un accélérateur unique qui resterait dédié (un seul comportement). Nous cherchons plutôt à intégrer des accélérateurs disponibles en un système sur puce (SoC pour « System on Chip ») programmable et communicant avec un CPU (et son logiciel associé). D'autre part, l'obstacle majeur à l'adoption des FPGA reste la nécessaire connaissance de l'architecture matérielle à implémenter, pour l'optimisation du programme et pour l'intégration finale du FPGA.

Une des clés de la réutilisation et de l'intégration d'accélérateurs est d'utiliser des architectures centrées sur la communication. Le *réseau sur puce* (NoC pour « Network on Chip ») permet de faire passer la communication du niveau d'abstraction *physique* au niveau *système*. Un NoC n'a pas les défauts des infrastructures points à points, gourmands en ressources d'interconnexion, et des bus, limités en nombre de terminaux (non extensibles) et à communications non parallèles (un seul lien partagé).

Malgré l'adoption des outils HLS par des industriels de plus en plus nombreux, la conception des accélérateurs et des circuits de communication(que

nous appelons IP pour « Intellectual Property ») reste réservée aux experts de la conception de circuits. Nous les appelons *fournisseurs d'IP* dans la suite de ce mémoire. La coopération entre fournisseurs d'IP et concepteurs d'algorithmes est limitée, aujourd'hui, par trois facteurs :

Le problème de confidentialité : les algorithmes à synthétiser sont la propriété du concepteur d'algorithmes, son savoir faire.

Le temps de développement et de vérification : une faible variation de l'algorithme peut nécessiter des mois de développement et de validation matérielle.

L'absence d'adaptation : un circuit est de taille fixe. Un changement de FPGA nécessite un nouveau cycle de développement, de synthèse et de validation pour profiter des nouvelles ressources disponibles (un circuit optimisé).

La problématique générale est donc la suivante : **comment permettre à un concepteur d'algorithmes d'accélérer un calcul sur FPGA donné, de manière automatique, à partir d'une bibliothèque fournie par un fournisseur d'IP sans lui divulguer son algorithme ?**

Répondre à cette question c'est proposer une méthodologie de génération automatique, générer un circuit qui permette une accélération par rapport aux architectures traditionnelles et enfin concevoir un circuit à base d'accélérateurs réutilisables pour divers algorithmes. En d'autres termes, nous chercons à garantir une portabilité (indépendante du type de FPGA), une extensibilité (plus de ressource permettent plus de parallélisme) et une programmabilité (éviter les longues synthèses à chaque évolution de l'algorithme).

Le problème se situe donc à plusieurs niveaux. D'abord, la spécification de l'algorithme doit permettre le recueil des informations nécessaires à une implantation optimisée pour une architecture donnée sans nécessiter de connaissances architecturales (problématique développée en partie 1.2.1). Ensuite, le passage du niveau comportemental (programme) au niveau circuit nécessite une synthèse qui prend en compte les spécificités de l'architecture : optimisée, extensible (au sens de mise à l'échelle) et programmable (problématique développée en partie 1.2.2). Il s'agit de pouvoir dimensionner le circuit final, y intégrer des structures pour un contrôle distribué et générer un programme optimisé pour ce circuit (problématiques développées, respectivement, en parties 1.2.5, 1.2.3 et 1.2.4).

1.2.1 Problème de l'analyse

Le flot HLS classique commence par l'analyse du programme : la compilation. Le but de cette étape est d'extraire le parallélisme potentiel de l'algorithme. Les langages les plus populaires sont de type impératif (C/C++, Java, Python, etc). Dans cette mémoire, nous ne nous intéresserons pas à OpenCL (qui nécessite une connaissance architecturale du matériel ciblé), ni aux autres types de programmation moins populaires pour des raisons industrielles : ils seront plus

facilement acceptés par les concepteurs d'algorithmes.

Les langages de type impératif sont destinés à être interprétés de manière séquentielle (les premiers processeurs étaient purement séquentiels). Or, avec l'avènement des systèmes multiprocesseurs ou des processeurs multi-cœurs, l'exploitation du parallélisme de l'architecture passe par l'extraction du parallélisme potentiel du programme à exécuter. La tâche consistant à reconnaitre les instructions indépendantes d'un *programme séquentiel* peut être ardue sans l'aide de l'utilisateur notamment en ce qui concerne le traitement des boucles.

Les outils de synthèse actuels nécessitent l'ajout de directives de la part du développeur pour obtenir une synthèse optimisée. Comment éviter d'imposer cette contrainte à l'utilisateur ?

1.2.2 Problème de la synthèse de systèmes

La synthèse d'architecture comme son nom l'indique, aboutie à une architecture dédiée. Dans cette dernière, la communication des éléments du système repose sur des connexions points à points. Or, les connexions points à points sont connues pour ne pas être extensibles (mises à l'échelles), c'est-à-dire qu'elles ne permettent pas au système de garder les mêmes propriétés (notamment la tenue en fréquence) à grande échelle. Insérer une infrastructure extensible dans le flot de conception n'est pas simple. De nombreux paramètres rendent l'exploration de l'espace de conception difficile. Étant donné la grande influence d'un tel élément sur les performances du système et sa complexité, une nouvelle méthode doit être mise au point pour l'intégrer au système final. Comment intégrer la génération d'un réseau sur puce à la synthèse de systèmes ?

1.2.3 Problème de l'estimation de ressources

À la différence de la synthèse d'accélérateurs, la synthèse de systèmes nécessite un dimensionnement de l'infrastructure de communication, ici le NoC. Le réseau doit être pris en compte, notamment dans l'estimation de l'utilisation en ressources du système. Comment dimensionner le réseau pour un FPGA donné ?

1.2.4 Problème du contrôle distribué

Un système extensible doit pouvoir augmenter ses performances en fonction des ressources disponibles, c'est-à-dire, des FPGA différents. Or, actuellement, la synthèse d'accélérateurs aboutit à la génération d'une partie contrôle à base d'automate d'état. Un contrôle centralisé n'est pas extensible. Comment produire un contrôle réellement distribué ?

1.2.5 Problème du système programmable

À partir d'un code HDL (pour « hardware description level »), une synthèse de circuit sur FPGA peut varier de plusieurs dizaines de minutes à plusieurs heures. Aujourd'hui, les circuits générés ne sont pas programmables. Modifier, même légèrement, un algorithme nécessite une nouvelle synthèse. Comment permettre d'exécuter plusieurs algorithmes sur un même circuit si les même types d'opérateurs sont utilisés ?

1.3 Organisation de la thèse

Les travaux présentés ont été réalisés dans le cadre d'une thèse CIFRE pour l'entreprise Adacsys qui a mis fin à ses activités en décembre 2014. Cette entreprise n'avait pas d'expérience dans les domaines de la synthèse au niveau système, ni en utilisation des NoC. Cependant, son expérience dans la vérification fut centrale dans la mise en œuvre des premiers prototypes. La thèse s'est déroulée conjointement avec les laboratoires Hubert Curien et TIMA qui disposent respectivement d'une expérience de plusieurs années dans l'étude des NoC et des outils de synthèse au niveau système.

1.4 Objectifs et contributions

L'objectif des travaux présentés est de fournir un environnement de développement au concepteur d'algorithmes qui réponde aux critères suivants :

— Permettre à l'utilisateur de spécifier son algorithme sans connaissances architecturales.

— Générer automatiquement un circuit optimisé pour un FPGA à partir d'un programme.

— Ne pas nécessiter de nouvelles synthèses si un nouvel algorithme n'utilise pas de nouveaux opérateurs.

Le premier critère concerne la méthode d'analyse du programme. L'approche proposée, dans ces travaux de thèse, est originale au sens où, afin d'éviter les problèmes classiques de compilation, une analyse à l'exécution est proposée. D'une part, elle permet le passage de l'environnement logiciel à l'environnement matériel de manière transparente pour les concepteurs d'algorithmes. D'autre part, elle permet l'extraction du flot de données d'une manière plus fiable que les solutions existantes. Avec cette méthode, le concepteur d'algorithmes bénéficie des avantages natifs de l'environnement pour le développement et la vérification. La méthodologie proposée repose sur la création d'une bibliothèque d'accélérateurs matériels référencés avec leurs consommations en ressources matérielles – pour chaque FPGA supporté.

Pour répondre au deuxième critère, nous proposons une méthodologie de synthèse au niveau système et nous avons développé un outil qui l'implémente. Le flot de synthèse proposé intègre les réseaux sur puce au flot de synthèse d'architecture classique. Il réalise la synthèse sous contraintes de ressources d'un système à base de NoC. En plus de la génération de NoC adapté à la synthèse au niveau système, nous proposons une méthodologie d'évaluation des ressources du réseau utilisé afin d'estimer les ressources consommées sur FPGA par le système généré.

Nous utilisons, pour cela, une architecture flot de données paramétrable, à base de NoC, qui connecte les accélérateurs sélectionnés. Des composants permettant la gestion des tâches au niveau de chaque nœud du réseau et la gestion du contrôle distribué sont insérés dans le système généré. L'architecture finale est réutilisable pour plusieurs algorithmes car la re-programmation du chemin de données est rendue possible sans re-configuration des liens de communication. Pour le montrer, nous avons développé un gestionnaire de tâches programmable qui permet l'allocation de plusieurs tâches à un même opérateur. Le gestionnaire utilisé permet de distribuer le contrôle au niveau de chaque nœud du réseau, ce qui permet d'éviter les goulots d'étranglement d'un système centralisé.

L'ensemble des contributions est mis en œuvre dans un flot de conception automatisé. La figure 1.3 décrit ce flot et les contributions associées.

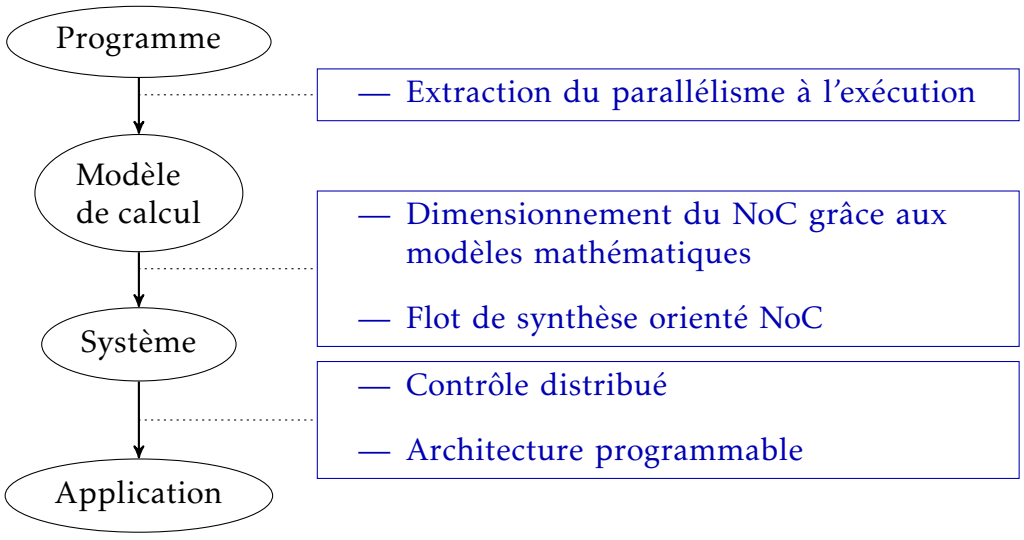


Figure 1.3 : Flot de synthèse proposé et contributions associées.

Ce mémoire de thèse présente une méthodologie, un outil de synthèse et une architecture à base de NoC permettant de générer automatiquement un système programmable à partir d'une bibliothèque d'IP et d'un programme écrit dans un langage de haut niveau, et ce, pour une cible FPGA donnée. La spécification d'entrée ne nécessite pas de connaissances architecturales. Le système généré repose sur une architecture au contrôle distribué de type flot de données.

1.5 Sommaire

Pour commencer, le **chapitre** 2 présente l'état de l'art dans le domaine de la synthèse d'applications pour FPGA, reposant sur une architecture à base de réseau.

La méthodologie de conception proposée est alors présentée dans le **chapitre** 3. Il s'agit d'une adaptation de la synthèse d'architecture classique en incorporant les étapes de génération du NoC.

Une proposition d'analyse utilisant les principes de l'exécution symbolique est développée dans le **chapitre** 4 pour analyser les dépendances de données d'un programme parallèle.

Dimensionner le réseau pour un FPGA donné fait partie du problème de synthèse sous contraintes de ressources. Le **chapitre** 5 présente une méthode pour caractériser la consommation en ressources FPGA d'un NoC 2D maillé.

Afin de distribuer le contrôle dans le réseau, une architecture est proposée dans le **chapitre** 6. Des gestionnaires programmables des tâches sont proposés pour implémenter le modèle de calcul construit après l'analyse du programme.

Enfin, le **chapitre** 7 présente des études de cas, des résultats d'expérimentations sur plusieurs algorithmes. Chacune des étapes du flot est détaillée pour chaque programme synthétisé.

Une conclusion et des perspectives terminent ce mémoire au **chapitre** 8.

Chapitre 2 : Synthèse au niveau système sur FPGA utilisant les réseaux sur puce

Le plus acceptable des systèmes est celui de n'en avoir par principe aucun.
Tristan Tzara

Sommaire

L'objectif de ce chapitre est de présenter les problématiques et les propositions dans le domaine de la synthèse d'applications pour FPGA et en particulier les architectures à base de réseau.

Dans un premier temps, il est essentiel de définir des concepts généraux (partie 2.1) et de présenter le domaine émergent qu'est l'informatique reconfigurable (partie 2.2), afin d'être en mesure de décrire comment la conception au niveau système engendre des problématiques allant de l'analyse d'un programme à la génération d'une architecture de circuit (partie 2.3).

Ensuite, afin d'exploiter le parallélisme disponible, des modèles de calcul parallèles permettant de caractériser une application doivent être choisis. La modélisation flot de données est alors présentée en partie 2.3.3.

Par la suite, en partie 2.4, la présentation des NoC ainsi que les méthodes et outils basés sur les NoC ouvriront la réflexion sur les raisons qui font des réseaux sur puce des supports particulièrement adaptés pour des systèmes extensibles malgré un certaines contraintes. Pour cette raison, les outils existants qui permettent la synthèse de systèmes à base de NoC sont présentés ainsi que leurs limites.

Enfin, cet état de l'art se termine, en abordant les architectures de type flots de données et leurs avantages par rapports aux architectures Von Neumann (partie 2.5).

2.1 Quelques définitions

Afin d'éviter toute ambiguïté concernant des concepts régulièrement utilisés comme « algorithmes », « modèles de calculs » ou « application », il convient d'en faire une brève définition.

2.1.1 L'algorithme

Un *algorithme* est le résultat d'une décomposition d'un processus de calcul en actions de moins en moins complexes. Il s'agit d'un ensemble ordonné ou partiellement ordonné de *tâches* permettant à partir de données d'entrée d'aboutir à un résultat par calculs successifs. En tant que processus, il s'agit d'une succession d'étapes avec un début et une fin.

Un algorithme est une conception abstraite qui peut éventuellement être représentée par un *programme* informatique (les tâches sont des fonctions ou des instructions) ou un graphe *flot de données* (les nœuds sont des tâches, les arcs des dépendances de données).

Un *algorithme* peut être caractérisé par son niveau de parallélisme. Celui-ci est défini par le nombre de tâches pouvant s'exécuter en même temps. Le parallélisme potentiel d'un algorithme est déterminé par ses dépendances de données. Pour extraire le parallélisme potentiel d'un algorithme, il est nécessaire,

par conséquent, de construire le graphe des dépendances entre les données produites lors de l'exécution de celui-ci.

2.1.2 Instruction, tâche et opération

Il faut faire la distinction entre ces trois termes : instruction, tâche et opération. Une instruction est un type de traitement (calcul, accès à la mémoire, etc.) exécutable sur un processeur et est définie par un code opératoire. L'ensemble de ces codes opératoires et la manière de les utiliser forment le « langage machine ».

Une tâche est un calcul à réaliser. Elle peut donc être associée à certaines instructions ou un ensemble d'instructions.

Enfin, une opération est l'exécution d'une tâche sur un opérateur donné.

2.1.3 Le modèle de calcul

Un *modèle de calcul (*MoC *pour « model of computation »)* est un modèle abstrait de l'implémentation d'un algorithme. On distingue deux types de *MoC* : les flots de contrôle et les flots de données, selon qu'ils décrivent une structure de contrôle ou de données. Une manière de les représenter est sous forme de graphe : CFG (pour « control flow graph ») ou DFG (pour « Data Flow Graph »). Il existe une grande variété de DFG en fonction de l'expressivité du modèle (Processus de Kahn, SDF, BDF, etc.) [LSV97].

Un programme informatique est conçu pour être traduit directement en MoC de type flot de contrôle. Dans les modèles de programmation classiques (les programmes traditionnels), le parallélisme n'est pas explicite (souvent à cause de la réutilisation de la mémoire). Dans le but de faire de l'exécution parallèle, il existe deux approches. La première est d'extraire le parallélisme des programmes. La deuxième est de changer de modèle de programmation. Dans ce mémoire, le programme à analyser est distingué du MoC qui est la représentation intermédiaire (modèle parallèle) résultant de son analyse.

2.1.4 Les modèles d'architecture, de performance et de structure

Un *modèle d'architecture* (*MoA* pour « model of architecture ») décrit ses fonctions de *coût* (en communication, énergie, surface, etc.) d'un système lorsqu'il exécute un MoC. Ce type de modèle provient de la tentative de séparation d'une description d'un calcul en *algorithme* d'une part et en *architecture* d'autre part, pour automatiser l'exploration de l'espace de conception des systèmes [PDM+15]. Elle est issue de la méthode nommée AAA pour adéquation algorithme-architecture.

Le but d'un MoA est d'offrir un moyen standard et reproductible d'évaluer l'efficience des décisions prises pendant la conception. Ces décisions concernent

des facteurs non fonctionnels tels que la consommation mémoire, la consommation d'énergie, le débit, la latence ou la surface (relative aux ressources utilisées). Un MoA est utilisé pour les systèmes dits "cyber-physiques", c'est-à-dire, où les calculs sont liés aux propriétés physiques du système.

Un *modèle de performance* (*MoP* pour « model of performance ») représente les contributions des éléments individuels d'un système à ses performances globales pour une implémentation spécifique (couple MoC/MoA).

Un *modèle de structure* (*MoS* pour « model of structure ») décrit l'implémentation spécifique d'un MoC sur un MoA (par exemple une description VHDL).

2.1.5 L'application

Nous appelons une *application* une implémentation qui réalise un calcul (un circuit, un exécutable, etc.). Ce calcul suit une méthode : un algorithme.

Contrairement à ce dernier, l'application est une conception concrète qui peut être logicielle (un exécutable), matérielle ou les deux. La figure 2.1 montre 3 niveaux d'abstraction :

— une spécification algorithmique,

— un modèle de calcul,

— et une implémentation (l'application).

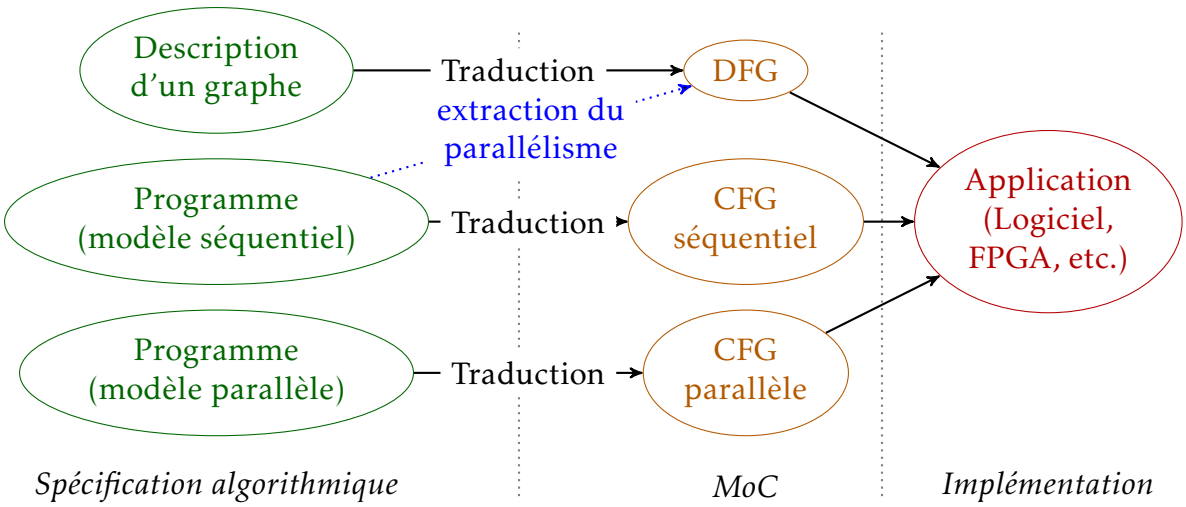


FIGURE 2.1 : Niveaux d'abstraction de la spécification d'un calcul. La spécification de l'algorithme (vert), le MoC (orange) et l'implémentation (rouge).

Dans ces travaux de thèse, nous cherchons à obtenir une forme d'*application* bien précise : un circuit à base de réseau s'exécutant sur un FPGA qui réalise un algorithme donné sous forme d'un programme informatique.

2.2 Vers une informatique reconfigurable

L'utilisation d'un FPGA comme co-processeur est dors-et-déjà possible pour peu que l'utilisateur soit aussi un concepteur matériel. Des acteurs majeurs de l'informatique comme Intel et ARM envisagent déjà le futur de l'informatique comme étant reconfigurable. Les principaux vendeurs de serveurs pour le HPC (Silicon Graphics Inc. (SGI), Cray et Linux Networx) ont déjà intégré des FPGA dans leurs produits.

Les entreprises et universités se tournent de plus en plus vers les nuages (« cloud ») pour bénéficier de ce qu'on appelle les infrastructures en tant que service ou IaaS, pour « infrastructure as a service ». L'intérêt est de permettre l'utilisation à distance des FPGA. En effet, les FPGA les plus performants ont un coût parfois prohibitif pour des petites structures (plusieurs milliers d'euros pour des FPGA Virtex de Xilinx ou Stratix d'Altera).

Malheureusement, le développement et la validation d'applications pour ce type de circuit sont bien plus longs que pour les processeurs. Quatre étapes sont indispensables afin d'exploiter une cible FPGA : concevoir l'application dans un langage HDL (1), vérifier son comportement en simulation (2) puis sur le FPGA (3) et enfin, la validation finale (4).

Le flot de conception FPGA / ASIC est décrit dans la figure 2.2. Le point de départ est la spécification d'une description du circuit et d'un banc de test. Ce dernier permet de générer les vecteurs de tests pour les différentes simulations qui jalonnent la conception d'un circuit. Trois étapes de validation par simulation suivent, dans l'ordre, la compilation de la description HDL, la synthèse logique et le placement/routage. Enfin, la phase de prototypage (validation finale) termine le processus de conception.

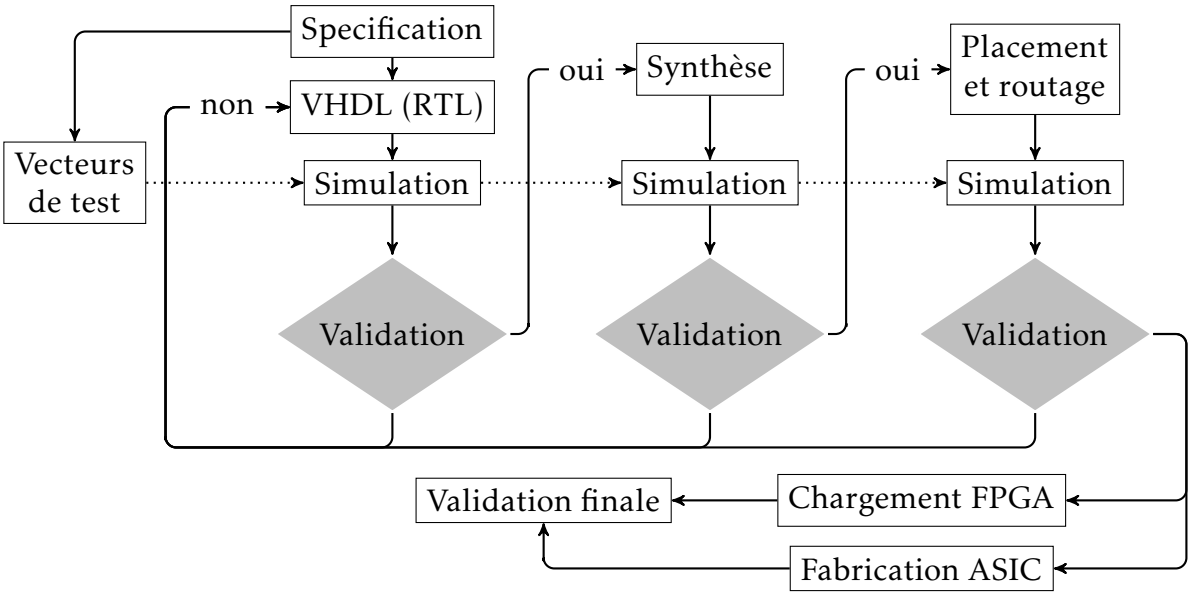


Figure 2.2 : Flot simplifié de conception FPGA et ASIC

Pour rendre la conception FPGA accessible à un public non averti, des outils d'aide à la conception sont développés (voir partie 2.3). Le but de ces outils est d'élever le niveau d'abstraction du système pour automatiser certaines tâches répétitives et diminuer le temps de développement/validation.

2.3 La synthèse de circuits au niveau système

2.3.1 Généralités

Avec l'intégration croissante des applications, le besoin de développer des outils au niveau système – ou ESL (pour Electronic System Level) – s'impose naturellement pour la conception, de la spécification à l'implémentation, de systèmes complexes. La plupart des méthodologies de synthèse au niveau système suivent une approche dite « Top-Down » [GHP+09].

Le modèle du toit à deux montants/faces (« double roof model »), figure 2.3 extrait de [GHP+09], montre bien l'analogie entre la synthèse matérielle et la synthèse logicielle. À chaque niveau d'abstraction, la spécification est transformée en implémentation (flèches verticales). Les flèches en pointillées représentent le passage au niveau d'abstraction inférieur. Le niveau système est le seul niveau commun entre les abstractions matérielles et logicielles.

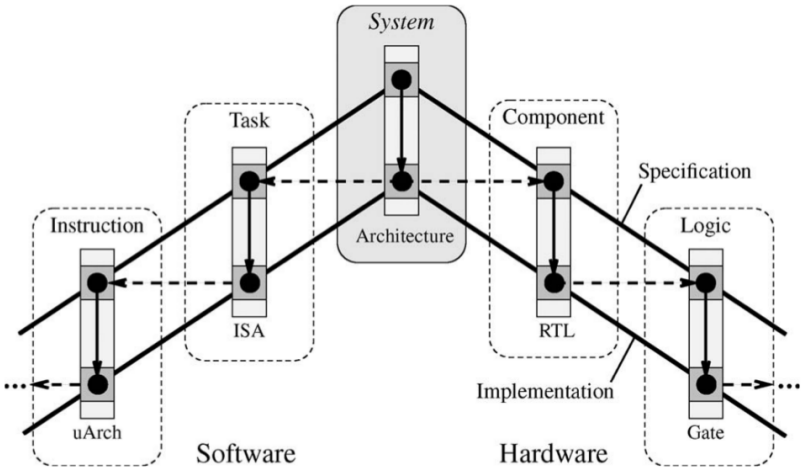


Figure 2.3 : Modèle du toit à deux montants

C'est au niveau système que sont données les spécifications comportementales et les contraintes (débit, latence, ressources, etc.). La synthèse ESL consiste au placement du modèle comportemental (MoC pour « Model of Computation ») sur le modèle architectural (MoA pour « Model of Architecture »). De même aux niveaux inférieurs, les synthèses se feront successivement par le passage de la spécification (modèle comportemental/fonctionnel et contraintes) à l'implémen-

tation structurelle. Les composants du modèle plus fin résultant servent d'entrée à la synthèse au niveau inférieur.

Chaque étape de synthèse aboutit à la génération d'un modèle structurel (MoS pour « Model of Structure ») et d'un modèle de performance (MoP pour « Model of Performance ») - voir figure 2.4.

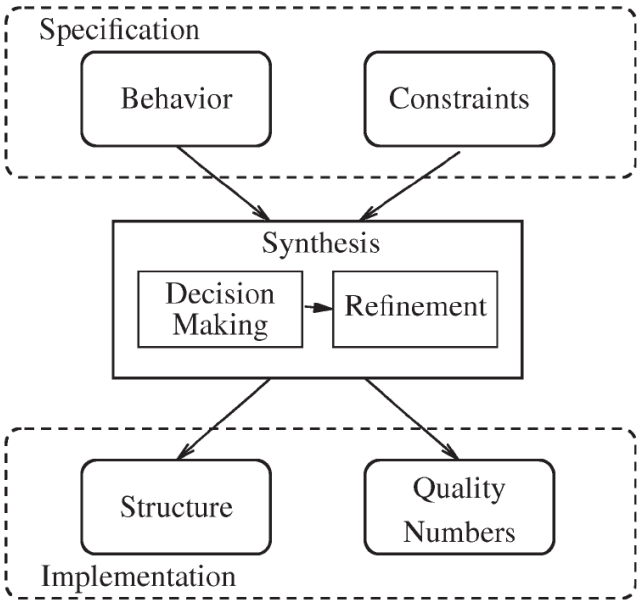


Figure 2.4 : Transformations lors d'une synthèse

De nombreux outils ont été développés pour concevoir et vérifier des circuits et les logiciels associés au niveau ESL [GHP+09]. En effet, 70 à 80% du temps et des dépenses sont consacrés à la vérification. La complexité de la conception optimisée d'applications basées sur des réseaux amène à développer des outils d'aide à la conception (CAD) poussant l'abstraction des modèles, donnés en entrée d'un flot de synthèse, au niveau système. En revanche, un effort important est à fournir pour concevoir l'interface entre le CPU et le FPGA. D'après Jon Masters - architecte ARM chez Red Hat - « Ninety percent of the effort is interface to the FPGA » [Red16].

Les outils ESL permettent de concevoir des systèmes complets (matériel et logiciel) à partir d'une spécification au niveau système (description structurelle ou comportementale). La plupart d'entre eux sont spécifiques à un domaine parce que les propriétés de certaines familles d'applications sont facilement exploitables pour spécialiser l'architecture. Ils nécessitent, pour obtenir des résultats satisfaisants, de fournir un MoC le plus riche possible en informations architecturales qui favorisent l'analyse et les optimisations. Il existe plus de 90 outils de modélisation ou de synthèse ESL [DPSV06] chacun avec ses avantages et ses inconvénients en termes d'effort d'apprentissage, de fiabilité des résultats

et de performances. Aucun ne s'est imposé comme outil de référence pour la conception de systèmes.

Les modèles d'entrée des flots de synthèse ESL sont en général des modèles flots de données (réseaux de processus (Kahn), flots de données synchrones, etc.), des automates d'état ou encore des programmes informatiques (un sous-ensemble du langage C étant le plus courant).

Les modèles de programmation développés pour les GPU, comme OpenCL, sont aussi de plus en plus utilisés car ils permettent d'expliciter le parallélisme d'un programme. Autrement dit, soit le MoC est spécifié directement, soit un programme est analysé pour le construire.

2.3.2 La synthèse d'architectures

L'une des disciplines ESL est la synthèse d'architectures aussi appelée HLS pour « High Level Synthesis ». Les outils HLS génèrent automatiquement un circuit au niveau transfert de registre (RTL pour « register transfert level ») à partir d'une spécification comportementale, de manière automatique. La spécification comportementale de haut niveau, partiellement ou non ordonnancée, est transformée en une implémentation entièrement ordonnancée (voir figure 2.5).

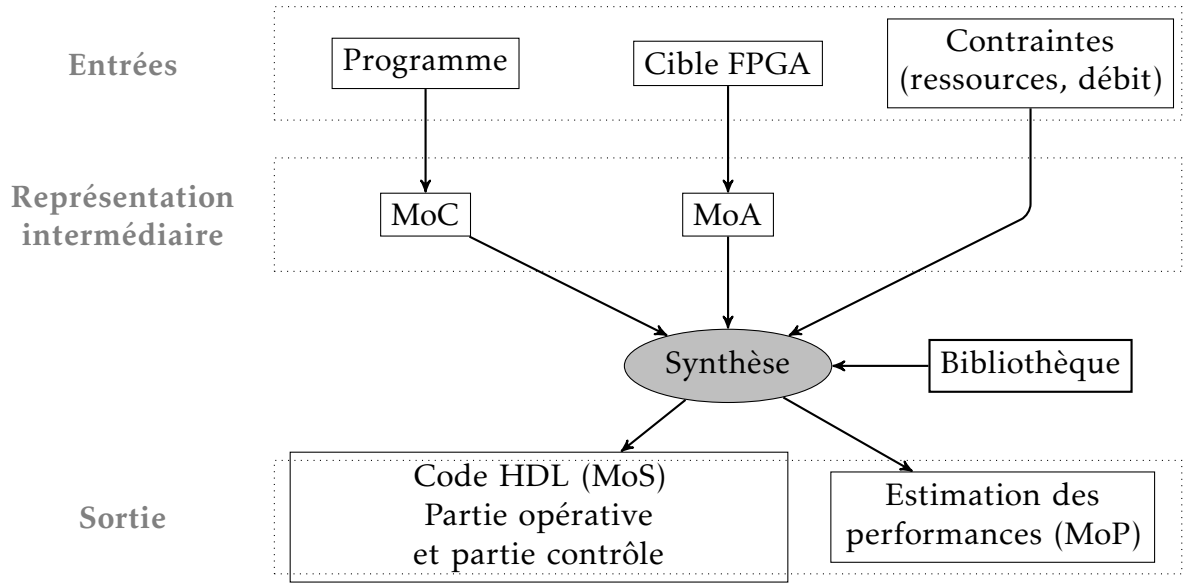


FIGURE 2.5 : Entrées/sorties de la synthèse d'architectures

Ils prennent en entrée :

Une spécification comportementale : un programme informatique, le plus souvent un sous-ensemble du langage C/C++ (langage impératif) auquel a été ajouté des annotations pour l'optimisation de la synthèse.

Des contraintes sur le circuit : coût, performance, consommation d'énergie, etc.

Une bibliothèque de modules : contient les composants matériels disponibles.

Le modèle structurel produit en sortie est un code HDL correspondant à :

La partie opérative : le chemin de données du circuit (la « netlist »).

La partie contrôle : un *automate d'état*.

Une large revue des outils HLS existants a été faite dans les travaux de [QB15]. Les langages utilisés sont le C, le C++, OpenCL et quelques langages fonctionnels. La HLS est déjà utilisée pour le traitement d'images [GSM+14, FWX14] et l'accélération des traitements sur les bases de données [AANS+14].

L'objectif de la synthèse d'architectures est de générer un circuit qui reproduise le *comportement spécifié* tout en satisfaisant les *contraintes données* et en optimisant une fonction de coût. Il s'agit de mettre en œuvre l'algorithme sur l'architecture de manière efficace sans forcément garantir que la solution obtenue soit optimale - les auteurs de [LS97] parlent d'implantation optimisée.

Les optimisations sont effectuées sous des contraintes de temps d'exécution maximale ou sous des contraintes de ressources (quantité d'unités matérielles - portes logiques, registres, LUTs ou autres élements de bibliothèque). L'exploration de l'espace de conception pour les optimisations multi-critères peut être très complexe et très longue. Toute autre contrainte spécifique doit être spécifiée au niveau de l'outil de synthèse ou par des annotations particulières.

Les améliorations attendues des outils HLS de prochaine génération, présentées dans [CGMT09], sont les suivantes :

- Le code d'entrée doit être lisible des spécialistes des algorithmes (sans connaissances des architectures matérielles).
- Les outils HLS doivent supporter des concepts avancés de programmation (orienté objet, programmation fonctionnelle, typage dynamique et programmation parallèle).
- Pour les algorithmes parallélisables, des architectures parallèles efficaces doivent être générées sans modifier le code d'entrée.
- Le processus d'optimisation doit être orienté conception. C'est-à-dire qu'il est possible d'augmenter les performances du circuit généré en modifiant successivement son code d'entrée puis en synthétisant à nouveau.
- Des circuits de performances au moins comparables, sinon meilleures que des circuits écrits manuellement doivent pouvoir être générés.
- Les outils doivent être ouverts et extensibles.
- L'exploration de l'espace de conception doit être améliorée.
- Les environnements de développement doivent être les plus transparents possibles. Idéalement, il n'est pas nécessaire de savoir que l'on fait de la synthèse.

2.3.2.1 Les étapes du flot HLS

Le flot de génération HLS (voir figure 2.6) est communément composé de cinq étapes [LRB11, CGMT09] :

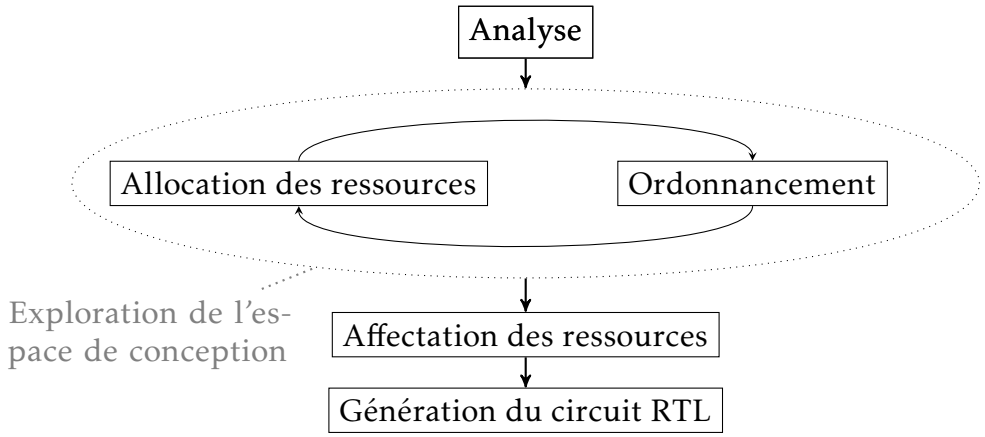


FIGURE 2.6 : Étapes de synthèse HLS

L'analyse : sélection des tâches et extraction du parallélisme par analyse des dépendances de données.

L'allocation des ressources : définit le type et le nombre d'unités matérielles à implanter dans le circuit (calcul, mémorisation, multiplexage) en fonction des tâches identifiées.

L'ordonnancement : décide de l'instant d'exécution de toutes les tâches, tout en garantissant le respect des dépendances de données. Cette étape ajoute une notion de temps (mise en « pipeline »).

L'affectation des ressources : associe à chaque type d'opérateur alloué une unité matérielle.

La génération du circuit RTL : génère un automate d'état respectant l'ordonnancement et le chemin de données avec les opérateurs sélectionnés lors de l'allocation des ressources.

Les étapes d'allocation et d'ordonnancement sont réalisées conjointement pour évaluer les différentes solutions qui satisfont les contraintes. C'est ici qu'est réalisée l'exploration de l'espace de conception.

Dans l'optique d'adapter ce flot pour la synthèse de systèmes basée sur les NoC, décrivons maintenant chaque étape.

2.3.2.2 Analyse et extraction du parallélisme

La première étape est l'analyse. La plupart des outils procèdent par compilation, c'est-à-dire qu'ils font l'analyse lexicale, syntaxique et sémantique du traitement à réaliser.

Cette étape prend en entrée une description comportementale (un programme) et produit un MoC en sortie (voir figure 2.7).

Comme présentée dans l'introduction, cette étape est critique car elle doit permettre de :

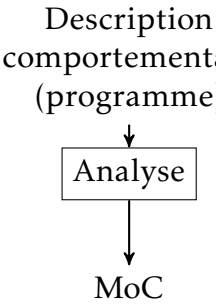


FIGURE 2.7 : la phase d'analyse

— minimiser la quantité d'informations nécessaire pour extraire le parallélisme potentiel,
— permettre une synthèse optimisée (en ressources ou en débit) pour une architecture matérielle cible, *un parallélisme disponible*.

L'analyse a pour objectif principal d'extraire le parallélisme intrinsèque du langage de spécification pour générer une représentation interne (MoC). Cette étape est généralement réalisée en utilisant des techniques d'analyse statique des dépendances de données. Elle doit identifier :

— les blocs de calcul,
— les dépendances de données,
— les flux et les formats (types) des données.

Durant cette phase d'analyse, des transformations structurelles sont souvent réalisées [GDGN03]. Les plus courantes sont :

— la transformation des boucles :
  — le déroulement séquentiel,
  — l'exécution parallèle,
  — la fusion ou le recouvrement des boucles ,
  — les transformations polyédriques,
— la transformation des conditions,
— le renommage de registres et l'insertion de registres temporaires.

La difficulté de l'analyse du parallélisme dans les programmes provient du fait que les dépendances de données ne sont pas toujours explicites. C'est d'ailleurs, la raison pour laquelle l'utilisation des pointeurs est très limitée dans les outils HLS acceptant le langage C.

La plupart des outils HLS font une analyse statique (compilation classique) du programme. Le parallélisme est extrait grâce à des méthodes d'analyse polyédrique [BJT99] lorsque les boucles présentent des dépendances de données inter-itération linéaires. Pour des types de dépendance plus complexes (non linéaires, imbrication intriquées de boucles, etc.), extraire le parallélisme est difficile sans indication de la part de l'utilisateur.

Certains outils permettent de transformer le code source d'un programme et l'optimiser pour des architectures parallèles [BBK+08] en insérant des primitives OpenMP. D'autres prennent directement en entrée des langages explicites comme OpenCL [CAD+12, CS14]. Cette dernière solution laisse à l'utilisateur la tâche de définir le parallélisme de son algorithme mais nécessite une connaissance de l'architecture ciblée.

Bien que les outils actuels proposent de plus en plus de fonctionnalités, il reste encore beaucoup à faire pour transposer sur le matériel des concepts avancés comme la programmation orientée objet, la programmation fonctionnelle, le typage dynamique et la programmation parallèle (« threads », OpenMP).

Toutefois, des techniques d'analyse dynamique peuvent aider à mieux identifier les sources de parallélisme. L'exécution ou analyse symbolique est une technique bien connue du monde du test de logiciel. Elle consiste à explorer tous les chemins possibles d'un programme pour une plage de données d'entrée. En outre, cette technique peut aussi être utile pour analyser les dépendances de données. Le problème majeur de cette technique est qu'elle n'est pas extensible (« scalable »). Ce problème est connu sous le nom d'explosion des chemins (« path explosion ») [CR85].

Une autre stratégie pour extraire le parallélisme est d'analyser les modèles CFG. En effet, des travaux [BJP91] ont montré qu'il était tout à fait possible de traduire des modèles de type flots de contrôle (un programme informatique décrit en langage impératif) en modèles de type flots de données. Ces derniers peuvent, ensuite, s'exécuter sur des architectures de type flots de données (voir partie 2.5).

La phase d'analyse produit, donc, en sortie, un modèle d'exécution parallèle. Le MoC généré est généralement un DFG à partir duquel il est possible de sélectionner des ressources matérielles, les ordonnancer et enfin générer le circuit final.

2.3.2.3 L'allocation des ressources

L'allocation des ressources consiste à sélectionner les opérateurs nécessaires à la réalisation des tâches de l'algorithme (voir figure 2.8). Cette étape est souvent associée à l'ordonnancement où les opérateurs sont partagés afin de respecter des contraintes de ressources ou de performances. Le nombre d'opérateurs à inclure dans l'implémentation finale est déterminé à cette étape. Le nombre de tâches à exécuter est souvent bien supérieur au nombre d'opérateurs disponibles pour un FPGA donné. Pour réaliser une sélection optimisée, une estimation est donc nécessaire à chaque étape d'exploration de l'espace des solutions.

Cependant, le besoin en mémoires, registres, multiplexeurs et logique de contrôle est en général implicite dans la description d'entrée. Il dépend, entre autres, de la manière de partager les opérateurs après une optimisation. C'est une des raisons pour laquelle les estimations à ce niveau ne sont souvent pas très bonnes.

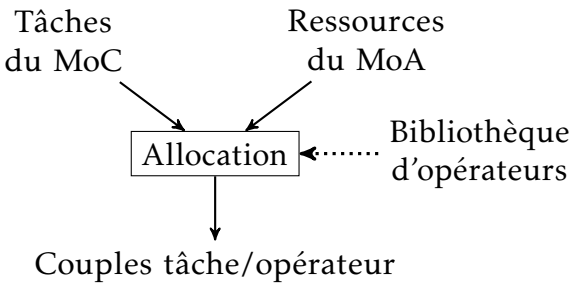


Figure 2.8 : la phase d'allocation des ressources

De plus, dans les FPGA les ressources sont diverses. En plus des LUT (« look-up tables »), il y a aussi des blocs de mémoire RAM, des DSP et autres composants dédiés à la communication (PCIe, Ethernet, etc.).

Les outils HLS traditionnels sont à grain fin. Ils composent avec les opérateurs de base (additionneurs, multiplieurs, etc.). Ils estiment la consommation des ressources pour ces opérateurs sans difficulté, mais sont toutefois moins précis quand il s'agit des ressources liées aux interconnexions (multiplexeurs) et à la partie contrôle, l'automate d'état. La raison est principalement due au comportement de l'outil de synthèse qui transforme le code HDL pour le FPGA ciblé. Celui-ci fait des optimisations qui ne sont pas toujours prévisibles. La simplification a un aspect positif quand les ressources sont limitées mais gènent les outils HLS qui opèrent sous contraintes de ressources et notamment la phase de dimensionnement du circuit.

Dans une synthèse à granularité plus importante (opérateurs complexes), la bibliothèque peut contenir l'information de consommation en ressources par opérateur pour un FPGA donné. Toutefois, les interconnexions et le contrôle posent toujours le même problème.

Après l'allocation des ressources, les tâches doivent être associées à un opérateur physique. Ensuite une date d'exécution doit être attribuée à chaque tâche. C'est l'étape d'ordonnancement. Sous contraintes de ressources, le nombre d'opérateurs alloués peut être inférieur au nombre de tâches, il faut alors partager le temps de calcul et augmenter la latence totale (pour les architectures « piplinées »).

2.3.2.4 L'ordonnancement

L'ordonnancement est généralement une étape d'optimisation. Elle consiste à affecter une date d'exécution et un des opérateurs sélectionnés à chaque opération de l'algorithme (voir figure 2.9). L'ordonnancement est un problème NP complet. De nombreuses méthodes heuristiques plus ou moins complexes (ILP, listes, chemins, etc.) existent et visent à satisfaire des contraintes diverses (ressources, latence, etc.). Dans le cas de la synthèse sur FPGA, il s'agit d'une optimisation sous contraintes de ressources.

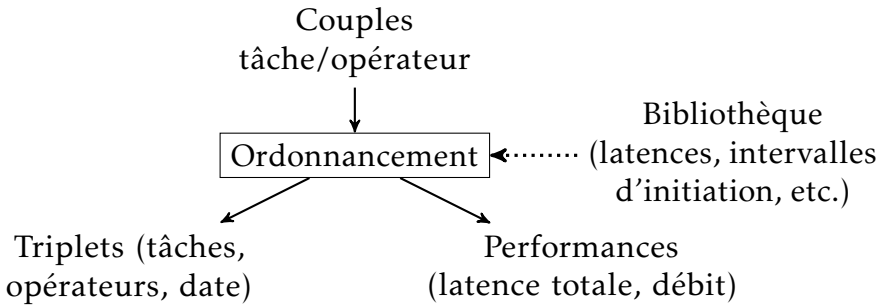


FIGURE 2.9 : la phase d'ordonnancement

Un bon ordonnancement doit maximiser les performances avec des ressources limitées. La solution obtenue doit être optimisée, elle doit être proche de la solution optimale au sens de Pareto. C'est-à-dire qu'il n'existe aucune autre solution ayant de meilleures caractéristiques en considérant tous les paramètres simultanément.

Dans une synthèse HLS classique, le circuit généré est constitué de deux parties : une partie contrôle et une partie opérative. La partie contrôle est un automate d'état et la partie opérative est un ensemble d'opérateurs connectés par des bus dédiés (connections point à point). Dans ce type de circuit, une date donnée correspond à une étape de contrôle de l'automate d'état.

Or, d'une part, afin de rendre le circuit extensible (« scalable »), les opérateurs peuvent communiquer entre eux à l'aide d'un réseau sur puce. Le problème, dans ce type d'architecture, est que les latences ne sont pas connues à la conception, simplement estimées. L'ordonnancement doit prendre en compte cette différence ou l'erreur d'estimation de la latence doit être à un niveau acceptable.

D'autre part, dans le cas où le contrôle est distribué à chaque nœud du réseau, l'étape de contrôle n'a plus le même sens que pour un automate d'état centralisé. Il faut donc adapter le concept d'étape à la nouvelle architecture.

Les différents parallélismes doivent être pris en compte lors de cette étape : le niveau de « pipeline« (parallélisme de flux), le parallélisme de tâches (indépendance entre opérations) et le parallélisme de données (un même calcul sur des données différentes).

2.3.2.5 L'affectation des ressources

Cette étape, aussi appelée « binding », consiste à affecter à chaque opérateur, mémoire et fonction de contrôle implicite alloué, un emplacement sur le circuit (partie opérative) (voir figure 2.10). C'est la dernière étape avant la génération du code HDL.

La difficulté de cette étape est de prendre en compte les contraintes physiques et temporelles des interconnexions entre les composants du circuit.

À cette étape, le partage des opérateurs implique l'utilisation de multiplexeurs et de registres supplémentaires.

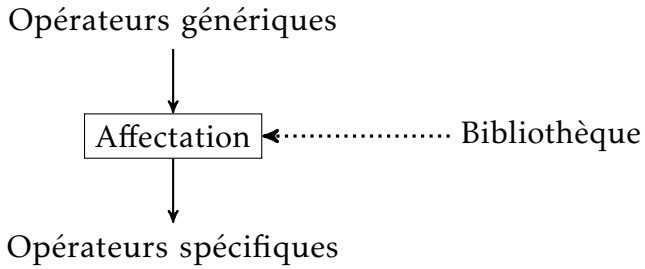


FIGURE 2.10 : la phase d'affectation des ressources

2.3.2.6 Génération du circuit RTL

La génération du circuit consiste, traditionnellement, à générer un automate d'état (partie contrôle) qui détermine l'exécution des tâches le long du chemin de données (partie opérative) (voir figure 2.11).

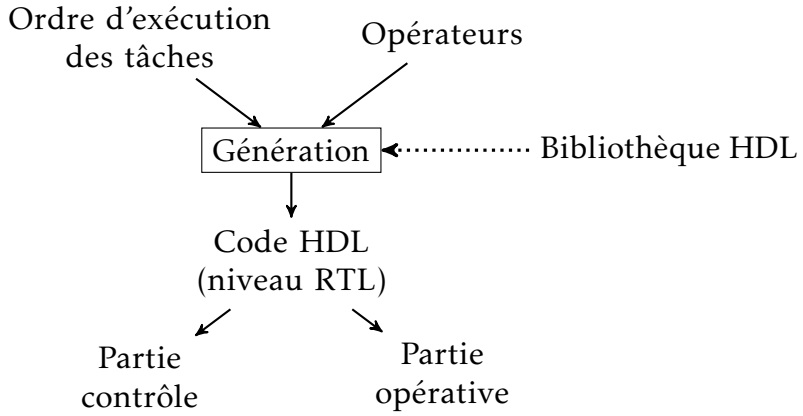


FIGURE 2.11 : la phase de génération du circuit RTL

2.3.3 La modélisation flots de données pour la synthèse ESL

Une des problématiques de la modélisation est de permettre la meilleure représentation possible du parallélisme d'un algorithme. Extraire de manière statique le parallélisme est le travail du compilateur. Une application peut avoir trois types de parallélisme : le parallélisme de données, de tâches et de flux. Les notions de parallélisme de tâches et de flux de données (« pipeline ») sont orthogonales par nature.

Dans les architectures CPU, le compilateur prend en compte le « pipeline » du programme pour ordonnancer les instructions afin de bénéficier du « pipeline » disponible. Ensuite, les instructions sont présentées en série au processeur qui doit gérer dynamiquement les aléas éventuels (dépendances structurelles, de contrôle ou de données).

Dans le cadre de la synthèse d'architectures classiques, les différents parallélismes doivent être gérés par un automate d'état. Il est donc nécessaire de modéliser les parallélismes potentiels avant de l'implémenter.

2.3.3.1 Contrôle centralisé : les automates d'état

Le outils HLS utilisent des automates d'état pour leur partie contrôle. Dans les systèmes extensibles, des centaines d'éléments de calcul peuvent communiquer entre eux. Dans ce cas, utiliser un mécanisme centralisé peut provoquer des phénomènes de congestion (points chauds).

Dans [HLY+06] les auteurs proposent de résoudre le *problème d'explosion des états* en utilisant un modèle appelé fFSM (pour « flexible FSM ») qui supporte la concurrence, la hiérarchie et un mécanisme interne de gestion d'événements. Mais ce modèle n'est pas satisfaisant pour les systèmes basés sur un NoC avec un schéma de communication basé sur la transmission de paquets.

Une autre approche intéressante est de convertir un programme C en un modèle de type flots de données [HOJH06]. Lorsque la communication est connue à la conception du circuit, la plus simple des solutions est de générer une architecture à base de FIFO (« first-in, first-out ») faisant communiquer les opérateurs entre eux [JMP+11].

Dans [AJR+05], les auteurs montrent que le système entier peut être généré à partir d'une analyse des modèles flots de données en appelant les outils HLS pour la génération des composants puis en les connectant à l'aide de FIFO. L'architecture résultante est entièrement matérielle et distribuée. Cependant, elle n'est pas extensible. Il n'y a pas de possibilité d'augmenter ou de réduire le parallélisme en fonction des ressources disponibles.

En règle générale, les circuits générés en HLS ne sont pas extensibles et sont dédiés à une seule application. La marge de manœuvre se situe à la génération du circuit et non après. L'utilisation des modèles flots de données est un choix pertinent, mais il n'existe pas d'outil HLS générant des architectures flots de données.

En plus d'expliciter les différents parallélismes, les avantages des flots de données sont :

Déterminisme du résultat : la propriété essentielle dont elle dispose est que l'ordre induit par la dépendance des données permet de déterminer le résultat d'un calcul.

Complétude déclarative : le résultat est dérivable à partir d'un ensemble d'« équations » initiales.

Dans l'optique d'utiliser ce type de modèle pour la synthèse, en voici quelques types.

2.3.3.2 Les modèles de type flots de données

Ils sont largement utilisés pour des applications multimédias et en traitement du signal afin de les implanter sous formes d'architectures parallèles. Il existe une grande variété de DFG (Processus de Kahn, SDF, BDF, CSDF, etc.) [LSV97]. Le compromis se faisant entre l'expressivité (ce qui peut être modélisé) et l'analysabilité (ce qui peut être calculé, et avec quelle efficacité) du modèle.

Il existe des outils permettant de construire et de simuler ces modèles. On peut citer Ptolemy (outil universitaire) [BHLM94], ADS de Agilent [Agi10] ou encore Simulink de MathWorks [sim].

Les flots de données sont des graphes où chaque nœud représente un acteur (une tâche) et chaque arc – de capacité illimitée – représente un canal (un chemin de données). Le lancement d'une tâche (acteur) est appelée tir car il provoque la sortie de *jetons* (données produites) à un certain taux (qui lui est propre). Les *jetons* produits et consommés ne sont souvent pas connus au moment de la compilation.

Ces modèles sont dits « data driven » (dirigée par les données) car c'est la disponibilité des données en entrée d'un bloc qui déclenche le calcul. L'idée générale est d'en produire aussi tôt que possible, c'est-à-dire dès le moment où il y a assez de jetons en entrée du bloc de calcul et qu'il y a assez d'espace disponible pour stocker la sortie.

Présentons, maintenant, trois de ces modèles en partant du plus déterministe puis en augmentant l'expressivité du modèle pour obtenir un comportement plus dynamique mais moins déterministe.

Les SDF ou flots de données synchrones : le diagramme de flot de données synchrone (SDF) est une manière naturelle de présenter des applications parallèles [LM87]. Introduit en 1987, il est un sous-ensemble des graphes flot de données. Le terme *synchrone* provient du fait que le nombre de données en sortie d'un nœud est indépendant de la valeur des données en entrée. Il peut donc être déterminé a priori, c'est-à-dire au moment de l'analyse.

La consistance d'un SDF est définie par la présence de vecteurs de répétition - fonction qui relie le taux de la sortie de jetons d'une tâche vers son successeur. Le débit d'une tâche étant le nombre de jetons produits par unité de temps.

Les avantages des SDF sont que l'ordonnancement peut être fait dès l'analyse, que la mémoire des canaux peut être bornée si un ordre parallèle périodique est trouvé et que des méthodes mathématiques de transformation existent.

Les BDF ou flots de données avec branchements conditionnels : les graphes flots de données à contrôle booléen (Boolean-controlled Dataflow – BDF) [BDT13] ajoutent aux graphes SDF deux acteurs dynamiques dont le comportement dépend de jetons de contrôle, consommés ou produits par les entrées/sorties conditionnelles de ces acteurs. Ces acteurs sont appelés *com-*

mutation (« switch ») et *sélection* (« select »). L'intérêt de ces acteurs est de permettre les branchements conditionnels et donc de rendre le modèle dynamique. La figure 2.12 montre un exemple de branchement conditionnel réalisé avec un modèle BDF.

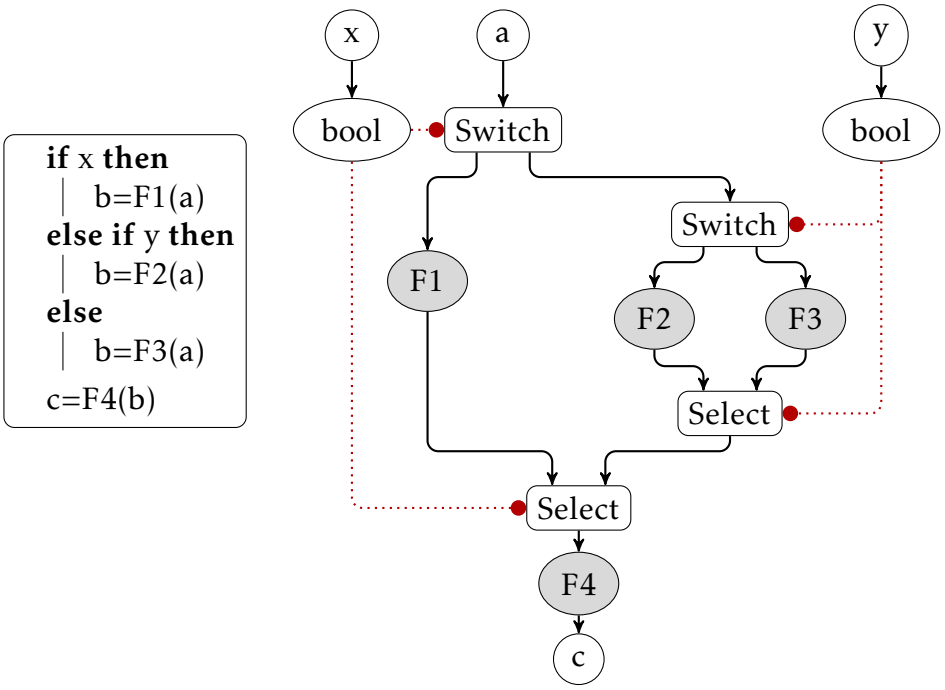


FIGURE 2.12 : modèle BDF supportant les branchements conditionnels

Les réseaux de processus de Kahn : un réseau de processus de Kahn (KPN pour « Kahn Process Networks ») est un modèle où les productions et consommations des nœuds de calcul ne sont pas connues à la compilation. L'ordonnancement statique (à l'analyse) des KPN n'est donc pas possible car les règles de tir ne permettent pas de construire un ordre tel que le système ne bloque pas en toutes circonstances [Kah74].

2.4 Le paradigme des réseaux sur puce et ses contraintes

La complexité du développement d'un système sur puce (SoC – System on Chip) croît avec la taille de celui-ci. Au début des années 2000, le paradigme des réseaux sur puce – ou NoC pour « Network on Chip » – est né [BDM02]. À un nouveau paradigme doit correspondre de nouvelles méthodes permettant d'accélérer le flot de conception [KJS+02]. C'est pourquoi il est indispensable de développer des outils qui se chargent de faire l'interconnexion optimisée des éléments du système de manière à permettre un bon acheminement des données et limiter les surcoûts liés à la communication (latences et débits). Dans cette partie, nous présentons les NoC, définissons des concepts fondamentaux et

décrivons les méthodes actuelles permettant l'optimisation des communications dans un NoC.

2.4.1 Qu'est-ce-qu'un réseau sur puce ?

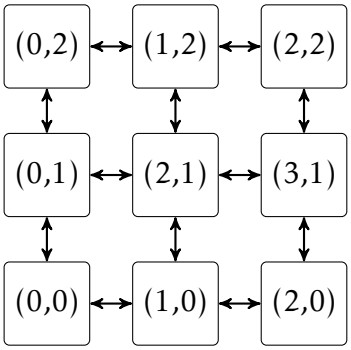
Un NoC est une infrastructure de communication alternative aux bus et aux connexions point à point qui a pour but de faciliter l'interconnexion à grande échelle d'IP. Le concept des NoC vient de l'idée d'utiliser les avantages des réseaux informatiques que sont :

— l'extensibilité (« scalability »),

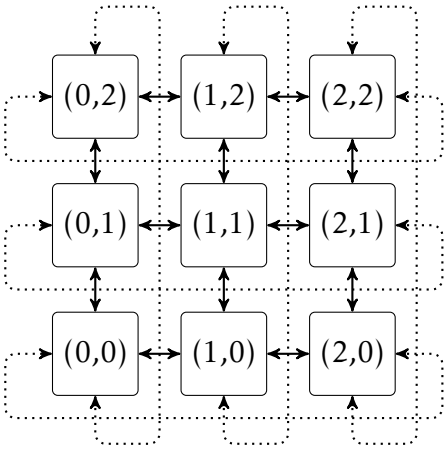
— le partage des chemins de communication,

— l'indépendance entre communication et le traitement des données.

Un réseau est un ensemble de routeurs connectés entre eux et à un, ou plusieurs, terminaux. Le terminal peut être un accélérateur dédié, un processeur, une mémoire ou encore un composant de communication. La manière dont est réalisée l'interconnexion des routeurs entre eux détermine la topologie logique du réseau. Il existe une multitude de topologies, qu'elles soient régulières ou non. Celles qui nous intéressent particulièrement sont les topologies maillées (2D ou 3D) et en tore – dite « Torus » (voir figure 2.13) – car elles sont plus proches de la topologie physique (2D planaire) d'un FPGA.



(a) Topologie maillée (« Manhattan like »)



(b) Topologie Torus

Figure 2.13 : deux topologies planes régulières de NoC

Comme dans les réseaux informatiques, l'information est structurée sous forme de paquets, avec un en-tête, une charge de données et éventuellement une queue. Comme le réseau postal, ils sont adressés. Chaque bureau de poste – notre routeur, auquel on a affecté une adresse – se charge d'acheminer ce paquet via le réseau, au bon destinataire. L'adresse de destination d'un paquet est placée au niveau de son en-tête. Un message correspond à un ensemble de paquets.

Comme les connexions points à points sont de largeurs déterminées, les paquets sont décomposés en unités élémentaires appelées flits pour « flow control unit (ou digits) ». Un paquet traverse un routeur flit par flit et est propagé dans le réseau de routeur en routeur jusqu'à sa destination (mode de routage « Wormhole »).

Un routeur connecte un port d'entrée à un ou plusieurs ports de sortie jusqu'à transmission complète d'un paquet, c'est ce qu'on appelle la commutation par paquet. C'est le type de la commutation le plus utilisé dans les NoC. Le chemin emprunté par un paquet est la succession des routeurs traversés entre la source et la destination. C'est un algorithme de routage implémenté dans le routeur qui détermine la route à suivre. Si la décision est toujours la même pour une destination donnée, elle est dite déterministe (ou statique) sinon elle est dite adaptative (ou dynamique).

Bien souvent, dans la littérature, le terme NoC est utilisé à la fois pour parler du système complet et pour parler du composant réseau seul. Dans ce mémoire, nous utilisons ce terme que dans sa deuxième acception. Pour la première, nous parlons de systèmes à base de NoC.

Un SoC à base de NoC peut être classé selon qu'il soit :

— un réseau de processeurs (MPSoC – pour Multi-Processor System on Chip), comme HEMPS par exemple [WBM06],

— un processeur dont les éléments de calcul (cœurs), mémoires et autres composants sont distribués dans un réseau (MultiCore ou ManyCore), comme le MPPA de Kalray [dDAB+09],

— un réseaux d'accélérateurs matériels,

— ou bien un système mixte avec processeurs et accélérateurs matériels.

Contrairement aux réseaux informatiques, les applications sur puce privilégient des NoC dédiés et optimisés plutôt que des NoC génériques plus gourmands en ressources matérielles.

Enfin notons que certains travaux ont montré la faisabilité de solutions multi-FPGA [CCL+10, TFR11]. Ces architectures sont appelées NoMC (pour « network on multi-chip ») [SLS10].

2.4.2 L'usage des métriques dans la conception

Les applications reposant sur un NoC peuvent présenter des caractéristiques variées [PGJ+05] (déterminisme et homogénéité des communications, contrôle centralisé ou non, sensibilité aux débits ou à la latence, etc.). Le choix des paramètres et des algorithmes internes au NoC (routage, arbitrage, etc.) peuvent impacter significativement les performances d'une application en termes de latence, de débit et de consommation [OHM05].

Dans le cas d'architectures MPSoC ou ManyCore, les caractéristiques des applications ne peuvent pas toujours être déterminées en amont de l'implantation sur matériel (applications variées ou fortement dépendantes des données).

La conception de telles applications repose donc sur des NoC génériques, indépendants des caractéristiques d'applications particulières. La solution courante étant de choisir pour le réseau soit des caractéristiques moyennes (telles que la taille des flits et des FIFO), soit des caractéristiques correspondant au pire cas (sur-dimensionner).

Dans l'étude [KBL+11] les auteurs proposent d'implanter deux types de NoC hétérogènes sur une même puce, l'un pour des applications plus sensibles aux débits, l'autre pour des applications plus sensibles à la latence. Un contrôle dynamique du débit et de la latence de chaque application en cours d'exécution permet de rediriger les données sur l'un des deux réseaux implantés sur la puce. Une telle approche aboutit à des applications plus efficientes d'un point de vue consommation. Il est, dans ce cas, nécessaire de dresser le profil des applications afin de les classer dans une des deux catégories.

Nous nous interessons, dans cette partie, à trois métriques que sont : la latence, le débit et la charge.

2.4.2.1 La latence

La latence d'une communication dans un NoC correspond au temps entre l'émission du premier flit d'un paquet par un nœud source et la réception du dernier flit de ce même paquet par le nœud destinataire [PGJ+05]. Elle est d'abord, proportionnelle à la *distance réseau* séparant deux nœuds. La distance réseau entre deux nœuds correspond au nombre de sauts (« hop ») sur le chemin à parcourir par un paquet pour arriver à destination, c'est-à-dire le nombre de routeurs à traverser.

Mais dans des conditions réelles de fonctionnement, la latence dépend aussi :

— du traffic dans le réseau (phénomènes de congestion),

— de la politiques de routage des paquets (statique ou dynamique),

— des arbitrages d'accès aux ports d'un routeur (gestion de priorité),

— de l'utilisation ou non de canaux virtuels (débit local disponible),

— de la profondeur des mémoires tampons en entrées des ports de chaque routeur.

Ces phénomènes induisent des surcoûts :

$$Latence_{Connexion} = Surcoût_{Source} + Durée_{Transport} + Surcoût_{Destination}$$

La latence totale pour un paquet est donc la somme des latences le long du chemin qu'il emprunte entre sa source et sa destination finale. Comme la latence n'est pas garantie dans le NoC, on parle souvent de latence moyenne pour l'envoi de *plusieurs paquets* :

$$Latence_{Moyenne} = \frac{\sum_{p=0}^{N_{Paquets}} Latence_p}{N_{Paquets}}$$

2.4.2.2 Le débit

La mesure de la performance dans les réseaux de communication est le débit en bit/s. On distingue en général le débit maximal supporté par l'infrastructure de communication ou bande passante (« bandwidth » en anglais) et le débit observé pour un certain nombre de données (« throughput » en anglais). Le deuxième est mesuré comme une fraction du premier.

Au niveau d'un lien physique entre deux routeurs, le débit maximal est proportionnel à la taille du flit (en bits), au mécanisme de contrôle de flux (poignée de main, basée sur les crédits, etc.) et à la fréquence de fonctionnement pour les systèmes synchrones.

$$Débit_{Local} = \frac{Nb_{Flit} \times Taille_{Flit}}{Latence_{Totale}}$$

Certaines mesures de bande passante prennent en compte le parallélisme des communications. Le débit total fait la somme des débits maximaux de chaque lien de communication.

Pour une application donnée, le débit qui nous intéresse est le nombre de résultats par seconde, par exemple le nombre de frame/s en traitement d'images ou en nombre d'options évaluées par seconde pour la finance. Lorsque le temps de calcul est négligeable, le débit dépend exclusivement des latences accumulées entre les dépendances de données.

2.4.2.3 La charge

La charge est une mesure de l'utilisation d'un canal physique. Il s'agit du temps utile de transport des données dans ce canal, exprimée en pourcentage de l'utilisation maximale du canal (débit maximal).

Dans un NoC, une charge élevée signifie un débit local plus important, mais des latence globales aussi plus importantes à cause des requètes de routage plus nombreuses au niveau des routeurs. En effet, l'arbitrage pour l'allocation des différents ports est un goulot d'étranglement. La *saturation* correspond au niveau de charge au delà duquel la latence augmente de manière exponentielle (voir figure 2.14). Le point de fonctionnement idéal doit donc se situer en dessous du point de saturation pour éviter les congestions.

2.4.3 Génération de l'infrastructure réseau

De nombreux outils de génération de NoC ont été développés. Ils se limitent à générer le réseau seul et non l'application complète. C'est le cas d'AT-

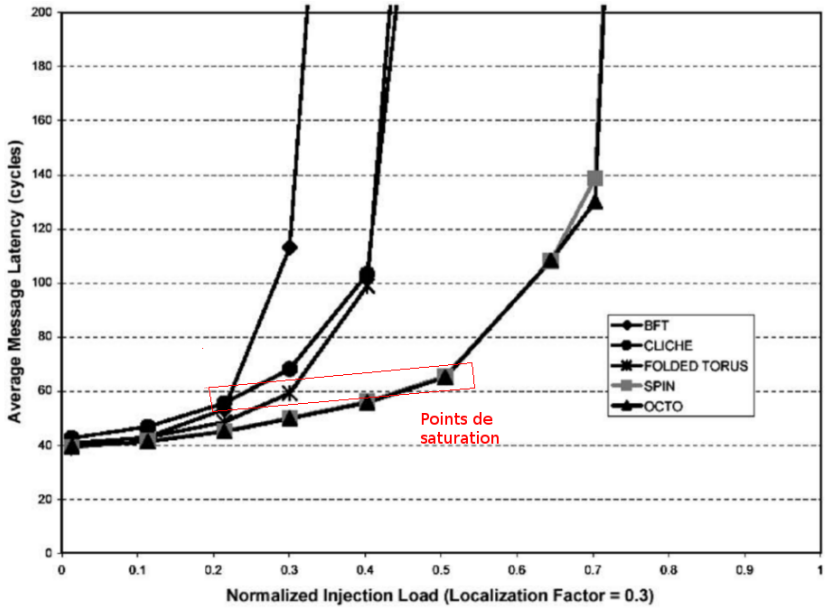


FIGURE 2.14 : influence de la charge sur la latence

LAS [MCM+04] qui génère le code RTL pour un NoC spécifique (Hermes) à partir des choix du concepteur. Il permet de simuler des trafics et estimer la consommation du réseau. C'est le cas aussi pour d'autres outils comme µSPIDER CAD TOOL [EDD+07] qui accepte des contraintes de latence et de débit, comme optiMap [SV06] et comme XpipesCompiler [JMBDM04].

Dans le cas des MPSoC, le système est déjà disponible. Le problème est d'y implanter de manière optimisée une application. Dans ce cas, les tâches doivent être placées sur les cœurs et leurs dates d'exécution doivent être définies. Ces décisions, interdépendantes, sont prises lors des phases de placement et d'ordonnancement.

2.4.4 Le placement et l'ordonnancement de tâches

Les outils d'aide à la conception de systèmes à base de NoC ne considèrent que des ensembles de composants matériels (IP) à placer sur chaque nœud. Certains d'entre eux, comme ATLAS, acceptent des spécifications comportementales, comme les scénarii de traffic, pour estimer les performances d'une application. En effet, ces dernières dépendent des latences entre chaque calcul ainsi que du partage des tâches par les opérateurs. C'est la raison pour laquelle il est nécessaire de choisir de manière optimisée l'emplacement du réseau (adresse) où s'exécutent les tâches d'un MoC (placement) et à quel moment elles sont exécutées (ordonnancement).

Dans la conception de systèmes entièrement matériels, le placement ne peut être que statique, c'est-à-dire décidé pendant la phase de conception du

circuit. Dans ce cas l'ordonnancement précède le placement car il détermine le nombre d'opérateurs matériels (il est couplé à l'allocation de ressources - voir partie 2.3.2.6).

Sur les architectures permettant l'exécution de plusieurs algorithmes, comme les MPSoC, on peut distinguer le placement statique (avant l'exécution de l'application) [LK03, HM05, WYJL09] du placement dynamique (pendant son exécution) [COM08, CM08].

Le placement d'un opérateur sur le réseau consiste à lui affecter une adresse réseau, tandis que le placement d'une tâche consiste à lui affecter un processeur sur lequel elle va s'exécuter. Dans ce deuxième cas, le placement influence l'ordonnancement (et réciproquement). Coupler ces deux optimisations est d'autant plus difficile que les deux problèmes sont NP-difficiles [HM05].

Pour une même application, des placements différents peuvent aboutir à des performances différentes. Les stratégies de placement visent en général à minimiser la moyenne des latences, la somme totale des latences ou la consommation. Afin d'évaluer les solutions de placement de tâches, un modèle qui décrit les communications entre tâches doit être fourni.

2.4.4.1 Le graphe des tâches de communication

Un graphe des tâches de communication (CTG – communication task graph) décrit les communications entre des tâches pour une application donnée. Il est défini à partir d'un modèle de calcul, parfois à l'aide d'outils statistiques d'analyse d'un programme. C'est le point d'entrée du placement des tâches sur le réseau.

Définition 2.1. Un CTG est décrit comme un graphe orienté. CTG = (T, C) avec T, l'ensemble des tâches à placer sur un NoC (les nœuds du graphe) et C, l'ensemble des liens de communication entre les tâches (les arcs du graphe). Le poids des arcs correspond au besoin de communication (en nombre de flits, par exemple).

Le CTG est le modèle de communication le plus répandu pour la conception d'applications basées sur les NoC. Le placement consiste à affecter à chaque nœud du graphe, une adresse réseau. La seule condition est qu'il y ait au moins autant d'adresses disponibles que de nœuds à placer.

2.4.4.2 Les méthodes heuristiques

Plusieurs travaux de revues des méthodes de placement et ordonnancement de tâches existent [SC13, PK04]. De manière générale, les meilleurs résultats sont obtenus avec les algorithmes qui parcourent le plus l'espace de conception, ce qui se fait au détriment de leurs temps d'exécution.

Parmi les méthodes statiques [PFYDL15], les algorithmes basés sur des listes sont les plus rapides. Mais les meilleures performances sont produites par les algorithmes « branch and bound » malgré un temps de calcul qui peut être prohibitif pour des systèmes de grande taille [PFYDL15].

Pour les méthodes dynamiques, la tâche est plus complexe car il faut gérer l'interaction entre les applications pour maximiser l'utilisation du système. D'après [CM08], c'est l'algorithme « Path Load » (prenant en compte la congestion) qui produit de meilleurs résultats en termes de temps total d'exécution, d'utilisation des canaux, de congestion et de latence des paquets.

En vue de la synthèse de systèmes, nous nous focalisons sur les méthodes de placement statiques [LK03, HM05, WYJL09, PFYDL15].

2.4.5 Flots de synthèse orientés NoC

Polaris [SEW+07] et Xpipes [BB04] offrent des chaînes d'outils pour la modélisation, l'exploration de l'espace de conception, la génération et la vérification des systèmes à base de NoC. Ces outils ne permettent cependant pas l'analyse d'une spécification comportementale.

Un flot de synthèse ESL basé sur IP-XACT [VPCM09] a été développé pour l'architecture FAUST2 [MPCVG04], basée sur un NoC. La chaîne d'outils proposée a pour but d'optimiser les paramètres du contrôleur de configuration de l'architecture FAUST2. Cependant, la plateforme générée est dédiée aux applications de télécommunication et le mécanisme de contrôle proposé est centralisé.

Plusieurs plateformes de génération de systèmes intègrent des CPU pour le contrôle des calculs. C'est le cas pour He-P2012 [CPMB14] où des blocs matériels sont couplés aux CPU sous forme d'îlots avec un réseau sur puce asynchrone nommé ANoC. Avec cette architecture, un modèle de programmation est proposé. Il permet de distribuer des tâches logicielles et matérielles sur les éléments du système.

G-MPSoC [KBA+15] est une architecture similaire pour FPGA basée sur un NoC avec deux types d'exécution : séquentielle et parallèle. Chacune est contrôlée par des structures dédiées, à l'aide d'instructions séquentielles ou parallèles. Le contrôle de l'exécution séquentielle est centralisé autour d'un processeur embarqué sur le FPGA. Le contrôle de l'exécution parallèle est distribué à chaque nœud du réseau, un îlot de calcul qui dispose de 16 élements de calcul (logiciels ou matériels - voir HoMade [PCD15]). G-MPSoC utilise un NoC nommé Xnet qui peut changer sa topologie logique pour s'adapter aux besoins de l'algorithme. Cependant, le programme exécuté doit être écrit et optimisé en fonction des éléments présents dans un système précédemment défini.

2.5 Architectures flots de données

L'architecture des systèmes à base de NoC doit avoir un contrôle distribué pour rester extensible et coupler le parallélisme de la communication avec celui des calculs. Une manière naturelle d'exploiter le parallélisme dans la communication est d'organiser le système en architecture flots de données.

2.5.1 Généralités

Les architectures de type flots de données [Vee86] existent depuis 1976 [Dav99]. Plusieurs machines ont été proposées jusqu'au début des années 90. Elles ont progressivement été abandonnées au bénéfice des architectures de type Von Neumann.

Les architectures de type flot de données sont des ordinateurs programmables où le matériel est optimisé pour le traitement de données à grain fin dit « data driven », c'est-à-dire dirigée par les données. Elles sont tolérantes aux latences non prévisibles de l'infrastructure de communication et supportent naturellement l'exécution parallèle [I+88]. Elles sont conçues pour exécuter directement des modèles de type flots de données. La spécification d'entrée de ces machines est donc un graphe flot de données.

2.5.2 Description et typologie

Une machine flot de données consiste en un ensemble d'élements de calcul qui peuvent communiquer les uns avec les autres [Vee86]. Les nœuds du MoC, représentant une instruction, sont souvent stockés avec une description du nœud du MoA, le *code de l'instruction*, une liste des *adresses de sortie* et de l'espace pour les *données d'entrée*.

L'unité élémentaire échangée entre les nœuds du graphe est appelée *jeton*. Dans une architecture flot de données, les jetons sont envoyés aux unités fonctionnelles avec un code opérande et une liste de destinations. Le tout forme un *paquet exécutable* [Vee86]. Une *unité d'activation* permet de générer un *paquet exécutable* sur la disponibilité des jetons.

Ces architectures sont de type MIMD (pour « multiple instructions multiple data »). Dans ce type d'architectures, lorsque les calculs sont indépendants, il est difficile de prédire la date d'arrivée des données sur deux opérateurs différents. Un mécanisme de synchronisation est nécessaire. Celui-ci induit un surcoût important en termes de latence.

Un nœud est dit *fonctionnel* si sa sortie est déterminée par ses entrées et sa description. Un nœud qui produit plus de données qu'il n'en consomme, augmente le degré de parallélisme. Un nœud exécute une instruction. Chaque instruction est considérée comme un processus séparé des autres. Une instruction qui produit une donnée possède une référence vers chacune des instructions qui consomme cette donnée.

Les architectures de type flots de données peuvent être classées selon qu'elles sont statiques, dynamique ou à sauvegarde explicite des jetons. Celles qui implémentent des mécanismes d'aquittement ou de verrouillage sont dites statiques. Elles n'autorisent qu'un seul jeton par arc. Celles qui préfèrent utiliser des jetons marqués et stockés dans une seule mémoire sont dites dynamiques. Les machines statiques sont plus simples mais permettent moins de parallélisme effectif. Le marquage de jetons permet d'isoler les contextes d'exécution mais

cela est gourmand en ressources nécessaires au stockage et à la communication des marqueurs de chaque jeton.

Les architectures flot de données de type « tagged token » (jetons marqués) utilisent des mécanismes complexes pour distribuer le calcul sur un ensemble de processeurs, dès que les opérandes sont disponibles [Cul86, SHNS86, GKW85].

Certaines machines sont totalement distribuées. Une machine distribuée est définie telle que :

— Aucun des modules du système ne peut déterminer l'état du système en entier.

— Aucun des modules du système ne peut forcer la simultanéité d'exécution avec d'autres modules.

2.5.3 Quelle performances obtenir avec ces architectures ?

Une bonne mesure de la performance des architectures est le niveau d'utilisation effective du matériel. C'est-à-dire analyser comment des techniques d'ordonnancement et de communication peuvent mener à une sous utilisation des ressources disponibles pour une application donnée.

La performance d'une architecture dépend beaucoup de comment elle accède aux structures de données. La plupart des architectures sont parfaitement adaptées pour un type d'algorithme et ont des performances moins bonnes pour d'autres types d'algorithme.

Dans les architectures de type Von Neumann « piplinées », les performances sont limitées par les aléas (structurelles, dépendances de contrôle et de données) et la latence des accès à la mémoire.

Les dépendances de noms (de registres) peuvent être supprimées par le compilateur ou par des mécanismes matériels. Il est aussi possible de limiter les aléas dûs aux vraies dépendances de données et aux dépendances de contrôle, en séparant les contextes d'exécution (threads) des instructions du « pipeline ». C'est naturellement ce que font les architectures de type flots de données.

Dans les architectures du type flots de données, les données ne sont affectées qu'une seule fois, évitant ainsi les dépendances de noms.

L'exécution d'une instruction est contrôlée par la dépendance des données, il n'y a pas de compteur de programme ni de tableau d'affichage centrale à mettre à jour. Lors de son exécution, les données en entrées sont consommées et des résultats sont produits.

2.5.4 Architectures modernes

Avec les limites (fréquence de fonctionnement et de ressources de routage) des technologies et donc les architectures dites « clock driven » qui s'appuient sur la rapidité d'exécution des instructions, des mécanismes flots de données on été réintroduits progressivement au sein des systèmes. Tout d'abord de manière

transparente pour l'utilisateur (exécution des instructions dans le désordre - Tomasulo [Tom67]), elle apparaissent plus tard de manière explicite à mesure que la granularité des tâches parallèles augmentait (au niveau thread et processus - GPU, MPSoC, ManyCore).

Les architectures modernes sont donc le plus souvent hybrides (flots de données et Von Neumann). Toutefois, quelques systèmes réellement distribués ont été développé depuis 1997 pour s'affranchir des limites des contrôles centralisés. On peut citer Raw [WTS+97], TRIPS [SNL+03], WaveScalar[SMSO03], Tilera [WGH+07] et Kalray [BDDDD13].

Les concepteurs de *WaveScalar* prétendent faire 2 à 7 fois mieux que les meilleures architectures superscalaires de la même génération pour les benchmarks SPEC et Mediabench. Cependant, dans [KS05] les auteurs ont utilisé l'architecture *WaveScalar* [SMSO03] pour implémenter l'algorithme de décodage H264. Cette architecture a un jeu d'instructions de type flot de données. Il s'agit d'une grille d'unités arithmétiques et logiques associée à de la mémoire cache. Ils arrivent ainsi à diminuer de 25% les communications et au final de 13% le temps total d'exécution pour l'application de décodage vidéo H264.

Tilera est un réseau maillé qui supporte un débit total de 1,28 Tbps pour chaque brique de base du système et de 2,56 Tbps en débit de bissection pour un réseau maillé de taille 8x8.

TRIPS consiste en un ensemble d'ALU (unités arithmétiques et logiques) connectés par un réseau d'opérandes de type scalaire. Il utilise un mode de transport dynamique pour ses opérandes. Pour cela il a besoin de l'aide d'un compilateur et d'un nouveau jeu d'instructions permettant d'encoder les dépendances de données directement.

Kalray est une architecture manycore qui dispose de 256 coeurs de calcul (MPPA-256) et d'un potentiel de 230 GFLOPS. Son infrastructure de communication est un NoC hiérarchique 4x4 grappes de 4x4 cœurs VLIW (Very Long Instruction Word). Le parallélisme est géré au niveau des grappes pour les processus, entre les cœurs pour les threads et au sein des cœurs VLIW pour les instructions.

Les architectures qui ne visent qu'à exploiter le parallélisme au niveau instruction (ILP pour « Instruction Level Parallelism ») doivent prendre en compte le fait que celui-ci varie entre 4 et 10, en moyenne, pour les programmes (3 à 4 en moyenne à l'intérieur des « basic blocks » de 10 instructions environ) du benchmark SPEC92 [PGTM99]. Les applications qui peuvent le plus bénéficier de ces architectures sont celles qui sont gourmandes en données (aussi dites « data-intensive » ou « data dominated ») telles que les applications de traitement vidéo, de big data, etc.

Les auteurs de [K+14] mettent en évidence que l'utilisation d'architectures flots de données avec les FPGA et un NoC peuvent aider à obtenir de meilleures performances globales (vitesse d'exécution).

2.5.5 Limitations des architectures flots de données

La granularité des données : la première limitation connue de ces architectures est la granularité trop fine des données. D'une part, à cause du surcoût de l'exécution de chaque cycle d'instruction (tailles des jetons notamment). D'autre part, à cause des faibles performances des applications avec un niveau de parallélisme bas.

La structuration des données : la plupart des architectures flots de données gèrent mal la représentation et l'utilisation des données structurées (tableaux de données). Elles sont traitées comme des ensembles de jetons.

Le stockage des jetons : les applications de type flots de données ont un impact important sur la mémoire. La raison principale est le stockage des jetons pour l'ensemble des instructions en attente d'exécution.

Le partage des opérateurs : contrairement aux architectures Von Neumann où la difficulté est de paralléliser un programme, un problème majeur des architectures flots de données est de restreindre le parallélisme disponible dans le modèle lorsque les ressources du matériel sont insuffisantes.

Le débogage : le fait de ne pas avoir de contrôle sur le moment où un calcul a lieu gène les programmeurs. Les applications pour ce type d'architecture sont difficiles à déboguer.

Le modèle de programmation : enfin, la manière de programmer des machines flots de données demande un effort d'apprentissage aux utilisateurs qui sont habitués aux langages de programmation non parallèles. De ce fait, même si elles ont prouvé leur potentiel, dix ans après leur lancement, leur adoption est loin d'être généralisée.

Chapitre 3 : Flot de synthèse pour la génération d'applications à base de NoC

Le but de l'informatique est l'émulation de nos facultés de synthèse, pas la compréhension des facultés analytiques
Alan Jay Perlis - Epigrams on Programming (1982)

Sommaire

Dans ce chapitre, la contribution qui concerne un flot de synthèse optimisée d'applications construites autour d'un NoC est développée en considérant l'utilisation partagée d'IP. Pour cela, nous avons développé un outil nommé *SyntheSys*, qui réalise automatiquement toutes les étapes du flot proposé. Tout d'abord, nous présentons la méthode générale en partie 3.1. Puis nous détaillons les étapes du flot de conception proposé dans la partie 3.2. Enfin, nous décrivons la structure des bibliothèques dans la partie 3.3.

3.1 Présentation de la méthode

Nous prennons comme spécification de départ un algorithme contenant certaines parties calculatoires (un ensemble de fonctions) à accélérer. Ces fonctions à accélérer sont clairement identifiées. Nous partons du principe que le concepteur d'algorithme a besoin d'accélérer une partie d'un *programme*. Pour cela, il indique quelles fonctions ont besoin d'être accélérées. C'est la spécification de l'algorithme.

Plusieurs hypothèses sont posées sur la spécification du programme à analyser :

1. Le programme est décrit dans un langage séquentiel (voir les contraintes en chapitre 4)
2. Le programme fait appel :
   — soit aux fonctions de la bibliothèque logicielle,
   — soit aux opérateurs standard du langage.
3. Les fonctions de la bibliothèque logicielle et les opérateurs standard sont associées à un *service*, c'est-à-dire une fonctionnalité qui peut avoir une implémentation (IP) synthétisable sur le FPGA cible.

La figure 3.1 montre le lien entre la spécification utilisateur, la bibliothèque logicielle et les IP sélectionnées.

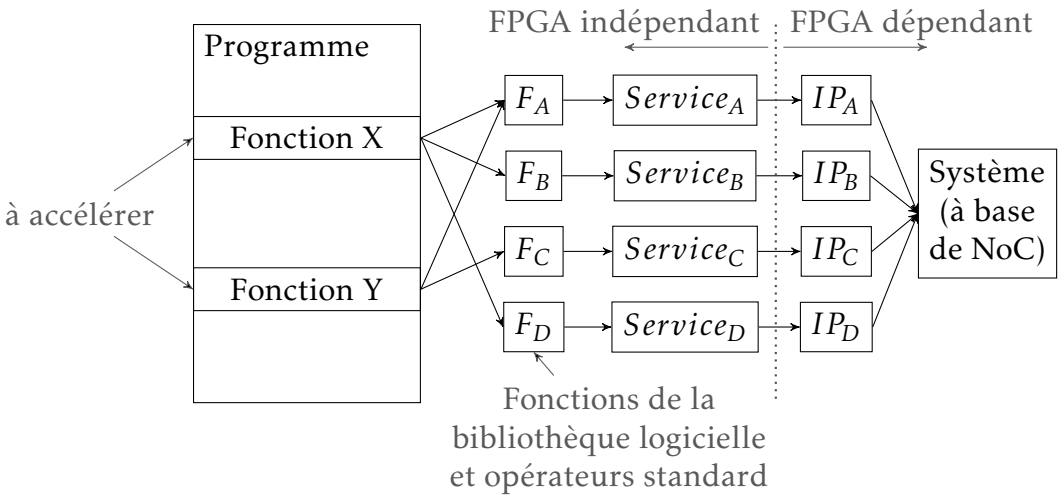


FIGURE 3.1 : Suite des sélections de la fonction utilisateur au réseau d'IP

Le concepteur d'algorithme doit cependant aussi indiquer la carte à base de FPGA ciblée. Celle-ci doit être référencée dans la bibliothèque des cartes FPGA supportées. La sélection des opérateurs (IP) est faite lors de la synthèse en considérant :

— le type des arguments de la fonction à accélérer (entiers, flottants, vecteurs, etc.),

— et les IP disponibles dans la bibliothèque et compatibles avec le FPGA ciblé.

Les hypothèses sur les bibliothèques (logicielle, matérielle et d'IP HDL) sont les suivantes :

1. Tous les arguments des fonctions de la bibliothèque logicielle doivent être explicitement associés aux interfaces d'un *service*.
2. Une carte FPGA référencée en bibliothèque doit être associée à :
   — un des flots de synthèse logique supporté,
   — un fichier de configuration contenant les entrées/sorties et les horloges disponibles,
   — une liste des ressources (LUT, Registres et Blocs RAM) disponibles sur le FPGA.
3. Les IP référencées doivent respecter les critères suivants :
   — elles doivent être synthétisables sur le FPGA avec lequel elles sont associées.
   — la quantité de données produites et consommées à chaque calcul est constante et déterminée à la synthèse,
   — leurs entrées et sorties peuvent être de type *scalaire* (taille inférieure ou égale à un *flit*) ou de type *vecteur* (nombre entier de flits),
   — leurs interfaces doivent disposer des signaux décrits en figure 3.2,

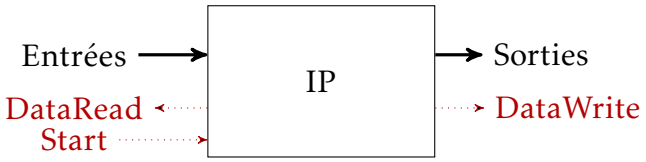


Figure 3.2 : Interface d'une IP (en pointillés les signaux de contrôle pour la sérialisation des données)

La figure 3.3 montre un cas d'utilisation du flot de synthèse. D'un coté, un expert matériel (fournisseur d'IP) ajoute à la bibliothèque un certain nombre d'IP. Celles-ci peuvent être compatibles avec plusieurs FPGA. Un service doit être associé à l'IP et il permet de faire le lien entre une fonction logicielle et son implémentation sur FPGA (l'IP). Les paramètres des cartes FPGA supportées sont aussi ajoutées par un expert matériel (les entrées/sorties et leurs fonctions - PCIe, horloges, leds, etc). Des fichiers de configuration de la bibliothèque permettent de faire le lien entre une IP et les FPGA avec lesquels elles sont compatibles.

De l'autre coté, le concepteur d'algorithme fournit les fonctions qu'il veut accélérer et le FPGA qu'il souhaite utiliser. La synthèse permet alors de sélectionner les IP en fonction du FPGA choisi et de les instancier dans une architecture flot de données à base de NoC.

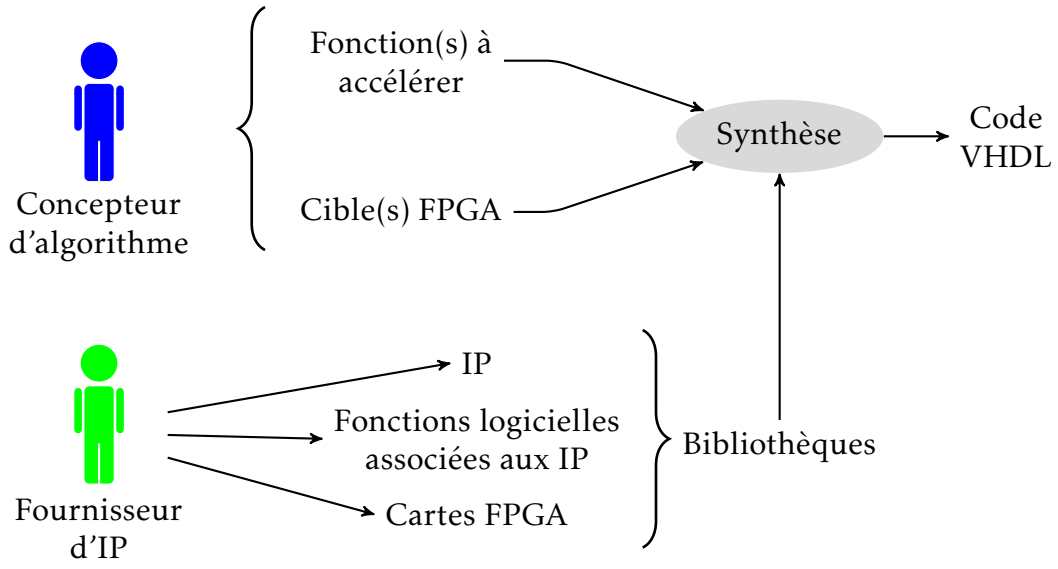


FIGURE 3.3 : Mode d'utilisation du flot de synthèse proposé

L'objectif de la synthèse est de sélectionner le jeu d'IP et les tâches devant s'exécuter sur ces IP de manière à optimiser les performances (débit) sous contraintes de ressources (un FPGA donné).

Notre approche repose sur une analyse des dépendances de donnée à l'exécution du programme qui produit un modèle flot de données qui supporte les branchements conditionnels et les boucles. Ce modèle permet de générer un système basé sur un NoC reliant une sélection d'IP qui réalise les calculs spécifiés.

Les IP sélectionnées peuvent être partagées pour réaliser plusieurs tâches grâce à des composants de contrôle programmables. Assemblés, ces composants additionnels (voir chapitre 6) gèrent la communication avec le NoC et forment l'*adaptateur réseau* (NA pour « Network adapter »). Il est généré en fonction de l'interface de l'IP (largeurs des entrées et sorties) et de la granularité des données (scalaires, vecteurs).

Une IP de communication (CE pour « communication elements ») est ajoutée au système en tant qu'interface d'entrée/sortie entre le PC hôte et le NoC.

Une vue simplifiée de la plateforme finale est présentée en figure 3.4

Les fichiers de configuration sont générés avec le code du circuit RTL. Il contient les données de programmation des chemins de données entre les nœuds du réseau. Le chargement de ces fichiers de configuration est une étape préalable à tout calcul utilisant les IP sur l'architecture générée.

Nous pouvons maintenant détailler, étape par étape, comment notre méthode permet, à partir d'un programme à analyser, de sélectionner un jeu d'IP et de les faire communiquer au moyen d'une architecture à base de NoC.

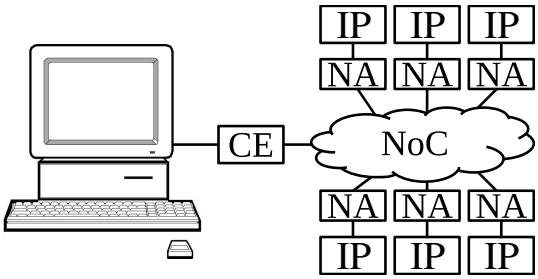


FIGURE 3.4 : Vue simplifiée de la plateforme finale

3.2 Les étapes du flot de synthèse

Le flot proposé est présenté en figure 3.5. Il commence par l'étape d'analyse du programme qui produit en sortie un modèle de type graphe flot de données (DFG). Ensuite, ce modèle est traduit en CTG (graphe de tâches) afin de pouvoir être ordonnancé.

Si un circuit a déjà été généré auparavant et dispose des opérateurs nécessaires, le CTG peut être ordonné directement sur celui-ci. Autrement, une étape d'exploration de l'espace de conception est effectuée. Elle consiste en une boucle alternant allocation et ordonnancement qui produit un graphe caractéristique de l'application nommé APCG. Le NoC est alors dimensionné et les IP de l'APCG sont placées judicieusement sur le réseau.

Une étape facultative d'ordonnancement prennant en compte les résultats du placement est alors effectuée et le circuit au niveau RTL peut être généré. Les fichiers de contraintes nécessaires à l'implantation sur le FPGA ciblé sont alors générés (horloges, entrées/sorties, etc).

Enfin, la synthèse logique utilisant les outils du vendeur de FPGA peut être effectuée et le fichier de configuration du FPGA généré peut être chargé sur le FPGA.

Une fois la synthèse terminée, les fonctions de communication fournies dans la bibliothèque logicielle permettent de communiquer avec le FPGA en utilisant l'IP de communication allouée durant la synthèse.

3.2.1 Analyse du programme

Le but de la synthèse étant d'exploiter le parallélisme potentiel de l'algorithme, la première étape est l'analyse du programme. Nous utilisons, pour cela, une technique d'analyse symbolique qui constitue une contribution importante de cette thèse, décrite en détail dans le chapitre 4.

Cette étape prend en entrée une fonction à accélérer, le type et la taille (si ce sont des tableaux) de tous ses arguments et un ensemble de contraintes. Ces contraintes comprennent la carte FPGA ciblée (disponible en bibliothèque) et éventuellement, le placement des opérateurs sur une carte FPGA précédemment configurée.

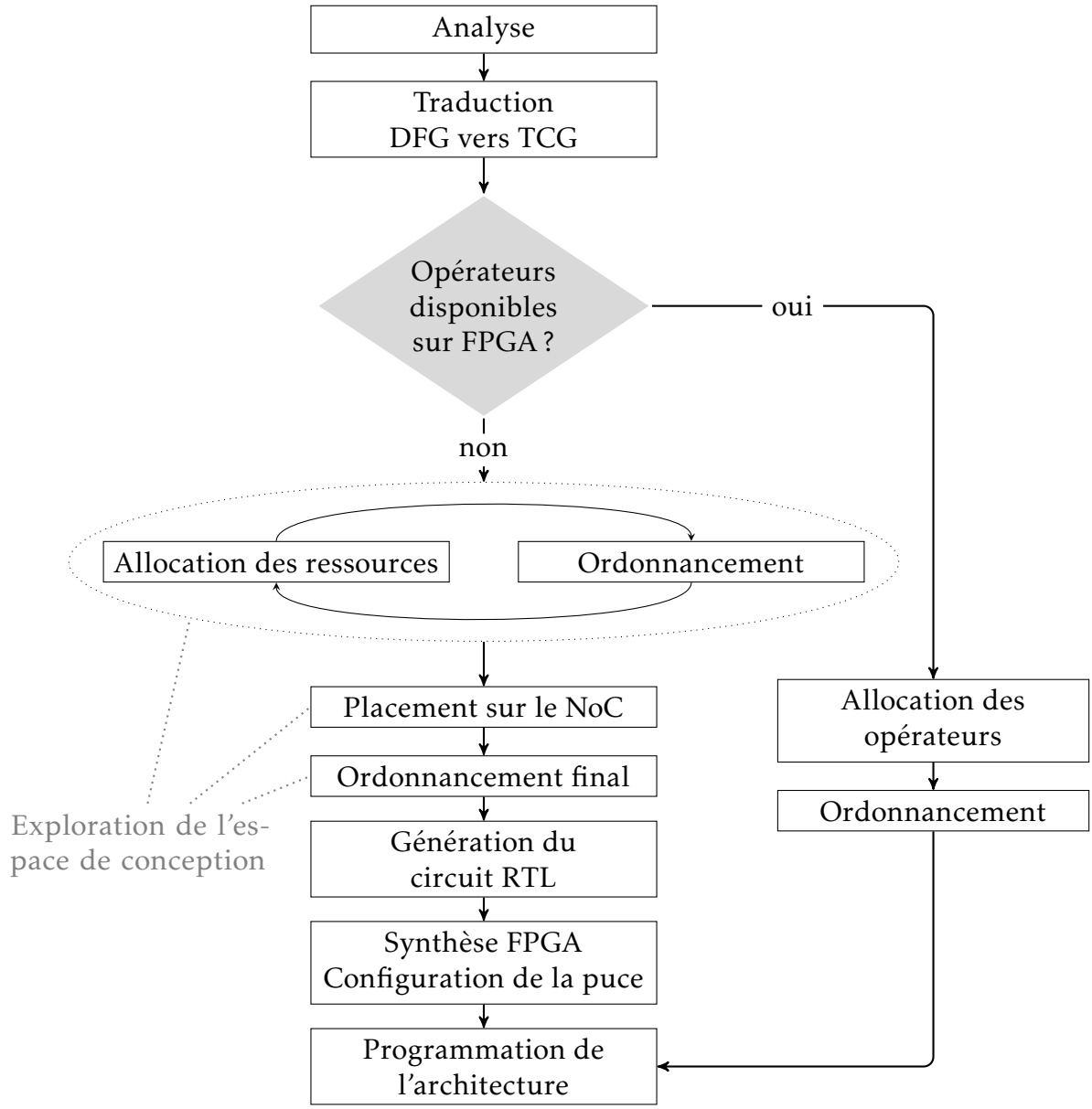


FIGURE 3.5 : Étapes du flot de synthèse HLS implémenté

Nous avons utilisé le langage Python pour montrer la faisabilité de la méthode. Tout code Python mono-processus qui ne fait pas appel à des extensions C/C++ (ou autres langages compilés) peut être analysé. La fonction à accélérer est marquée par l'utilisateur. Les limitations se font au niveau des fonctions de la bibliothèque et des opérateurs standard qui doivent accepter les arguments passés en paramètres (la bibliothèque étant bien sûr ouverte et extensible). Dans notre implémentation, la récursivité n'est pas gérée, sauf pour les cas où le nombre de récursions est indépendant des données d'entrée).

L'analyse produit un modèle de calcul (MoC pour « model of computation ») de type flot de données à partir d'un programme. La génération du MoC est

faite lors de l'exécution de la fonction spécifiée (voir chapitre 4). La figure 3.6 illustre cette génération et présente sur la gauche un morceau de programme et, sur la droite, le MoC généré après son analyse. Les quatre fonctions (F1, F2, F3 et F4) sont représentées par des nœuds grisés et les branchements sont rendus possibles par des nœuds de controle (« switch » et « select »).

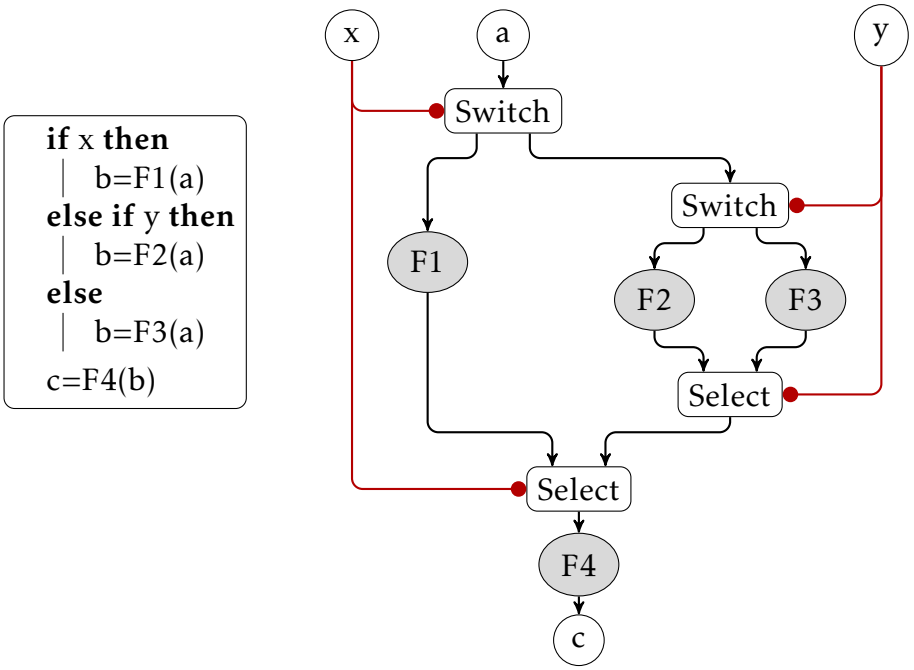


Figure 3.6 : Exemple de DFG produit après analyse

Le MoC produit est un sous-ensemble de processus de Kahn. Plus précisément, il s'agit d'un graphe de type BDF (« boolean dataflow graph ») [BDT13], c'est-à-dire un graphe flot de données synchrone (SDF) enrichi avec les nœuds *switch* et *select* (pour les branchements conditionnels), mais dans lequel les boucles sont acceptées. Dans ce type de modèle, les tâches ont une production de jetons indépendante des données consommées. De plus, pour faciliter l'ordonnancement, les latences et intervalles d'initialisation, ou d'introduction des données (DII) doivent être constantes. Ce sont des contraintes fortes sur les accélérateurs de la bibliothèque d'IP. Ces contraintes sont nécessaires pour qu'un ordonnancement statique optimisé soit possible. Autrement, dans le cas de modèles plus expressifs, il est nécessaire d'implémenter un mécanisme d'ordonnancement dynamique.

3.2.2 Traduction du DFG en graphe de tâches

Le DFG produit après l'analyse est ensuite traduit en un CTG. Cette transformation fait correspondre à certains élements du DFG (map, reduce, switch, select, tests booléens et entrées/sorties) des tâches de contrôle (map, reduce,

Route, select et communication). Cette étape de transformation est spécifique à l'architecture flot de donnée générée et repose sur la gestion du contrôle par des IP dédiées (voir chapitre 5).

Les figures 3.7 et 3.8 présentent les correspondances entre les nœuds du DFG et les tâches du CTG. L'intérêt principal est que chaque tâche du CTG peut être associée à une IP de la bibliothèque. Ces IP, qu'elles soient de type accélérateur ou IP de contrôle, sont partageables entre plusieurs tâches, ce qui augmente par la suite, les possibilités d'ordonnancement.

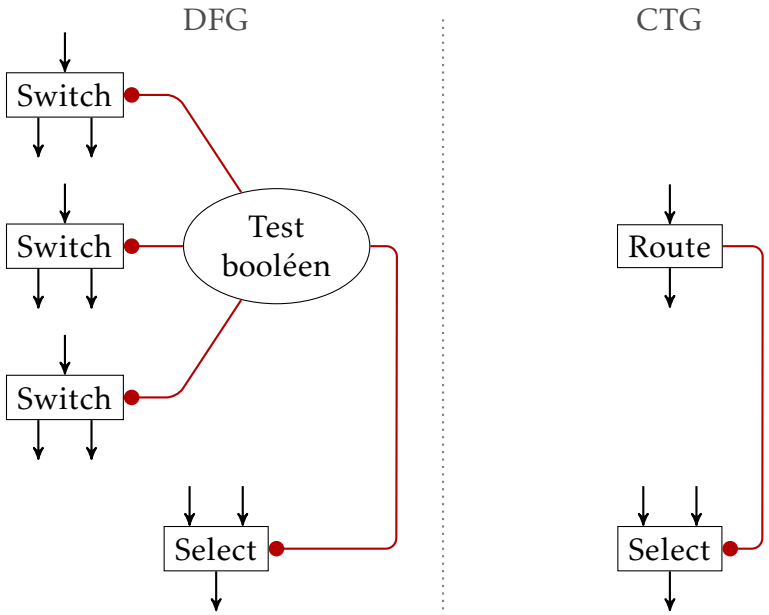


FIGURE 3.7 : Traduction du DFG vers CTG - branchements.

Lors d'un branchement conditionnel, les nœuds de type *switch* et les tests qui leur sont associés sont systématiquement fusionnés en une tâche nommée *Route*. Cette tâche reçoit donc, les données à tester, et celles à rediriger en fonction du résultat du test. Le choix d'associer un *test* et les *switchs* qui en dépendent, se justifie parce que, d'une part, les tests extraits du DFG sont toujours booléens (voir chapitre 4) et que, d'autre part, plusieurs *switchs* sont très souvent associés à un même test. Autrement dit, un même branchement concerne plusieurs données. Par conséquent, la granularité du test est toujours très faible et le traffic entre un *test* et les *switchs* est souvent important. La tâche *Route* a donc la fonctionnalité d'un routeur avec une table de routage à remplir durant la phase de programmation de l'architecture.

De même, les entrées et sorties sont respectivement produites et consommées par des tâches de communication d'entrée et de sortie. La figure 3.8 montre la fusion des nœuds d'entrée et de sortie en une seule tâche de communication.

Enfin, les tâches identifiées sont associées à des *services* disponibles en bibliothèque.

Figure 3.8 : Traduction du DFG vers CTG - entrées/sorties.

3.2.3 Allocation et ordonnancement

Le CTG va être transformé en APCG (« application characterization graph ») [Jo04] lors des étapes d'*allocation* et d'*ordonnancement*. L'allocation sélectionne pour chaque *service* une liste de modules compatibles avec le FPGA ciblé. Puis, il détermine le nombre maximum de modules implantables sur le FPGA en fonction des ressources disponibles. Pour estimer les ressources utilisées, nous utilisons un modèle mathématique évaluant la quantité de chaque ressource du FPGA (LUT, Registres, RAM) pour une taille de NoC donnée (voir chapitre 5).

Les possibilités d'allocation doivent être explorées pour choisir celle qui maximise les performances après ordonnancement. L'ordonnancement sélectionne un ordre optimisé d'exécution pour un jeu d'opérateurs donné. Cette étape ne prend pas en compte les temps de communication dans le NoC (transmission des paquets). Pendant l'allocation, une IP de communication est sélectionnée pour effectuer les échanges entre le PC hôte et le FPGA.

L'ordonnancement est un problème NP-difficile [M+09]. Nous utilisons, pour cela, l'algorithme HOIMS (pour « hardware oriented iterative modulo scheduling ») [SWN02] qui alterne l'allocation des opérateurs et l'ordonnancement des tâches en prenant en compte le « pipeline ». Cet étape produit en sortie un graphe de caractérisation de l'application ou APCG. L'APCG est un modèle largement utilisé dans la génération de systèmes basés sur les NoC où un nœud représente une IP et un arc (pondéré) représente les communications entre deux IP.

L'algorithme HOIMS (décrit en annexe B) a été modifiée pour intégrer l'estimation des ressources du NoC et des composants associés au contrôle. Comme présenté dans la fonction *EstimateRscUsage* (voir l'algorithme 1), les ressources correspondantes au gestionnaire de tâches (« TaskManager »), à l'adaptateur réseau et au module sélectionné sont ajoutés pour chacun des nœuds aux ressources estimées du NoC.

Les ressources du NoC sont estimées à partir d'un modèle mathématique alors que les ressources des autres composants sont disponibles en bibliothèque, c'est-à-dire préalablement déterminées lors d'une synthèse logique (réalisée une fois par FPGA compatible).

```
Require: FPGAModel, ModList, APCG
Ensure: UsedRsc, NoCParams
 1: function EstimateRscUsage(FPGAModel, ModList, APCG)
 2:     UsedRsc ← HwResources(FPGAModel)
 3:     TaskManagerRsc ← GetModuleResources("TaskManager", FPGAModel)
 4:     NetAdaptRsc ← GetModuleResources("NetworkAdapter", FPGAModel)
 5:     for Mod in ModList do
 6:         UsedRsc+=Mod.GetUsedResources(FPGAModel)
 7:         UsedRsc+=TaskManagerRsc
 8:         UsedRsc+=NetAdaptRsc
 9:     NoC=GetModule("NoC", FPGAModel)
10:     NoCRsc, NoCParams = GetResourcesFromModel(NoC, FPGAModel,
    APCG)
11:     UsedRsc+=NoCRsc
12:     return UsedRsc, NoCParams
```

Algorithme 1 : Fonction d'estimation de ressources

3.2.4 Placement des IP sur le réseau

Ensuite, l'APCG est placé sur le NoC. La topologie du NoC est fixée (pas d'exploration des topologies). Le nombre de nœuds de l'APCG détermine la taille du NoC. Seuls les paramètres comme la largeur des liens (taille des flits), la profondeur des FIFO est choisie en fonction des IP sélectionnées. Si une architecture a déjà été générée, les caractéristiques du système existant sont utilisées. Sinon, un nouveau NoC est dimensionné et paramétré.

Le placement consiste à affecter une adresse à chaque IP. Il prend en entrée un APCG et un ARCG (« architecture characterization graph ») pour produire un APCG placé. L'ARCG [Jo04] est un graphe complet dont les nœuds représentent les nœuds du NoC et les arcs représentent les distances de type Manhattan (nombre de sauts de routeur à routeur) entre les nœuds. Le placement des tâches est aussi un problème NP-difficile. Nous utilisons deux méthodes heuristiques :

— l'algorithme BB (« branch and bound »),

— l'algorithme basé sur des listes (conçu pour les besoins de la thèse).

Le premier est utilisé lorsque le nombre d'IP est réduit. En effet, il se rapproche beaucoup des méthodes exactes. Son temps d'exécution est prohibitif lorsque le nombre d'IP à placer dépasse 16 (plusieurs heures). Dans ce cas, c'est le deuxième, moins efficace mais plus rapide, qui est utilisé. Ces deux algorithmes ont pour objectif de minimiser l'énergie globale, c'est-à-dire, le nombre de cycles dédiés à la communication (la somme totale des latences).

La figure 3.9 récapitule les transformations depuis le CTG jusqu'à l'APCG placé sur un NoC.

Les IP étant placées sur le réseau, une étape facultative d'ordonnancement peut être effectuée afin de prendre en compte les délais dûs aux latences de

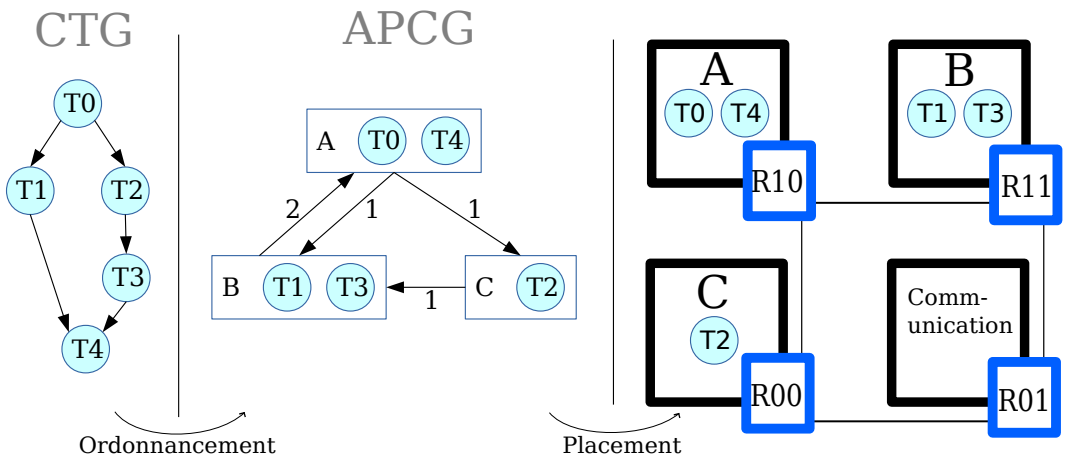


FIGURE 3.9 : Allocation, ordonnancement et placement des IP

communication dans le réseau. Bien sûr, les latence ne sont pas déterministes et seules les latences théoriques (sans aucune congestion) sont utilisées.

3.2.5 Génération du code RTL

Enfin, la génération du circuit repose sur l'instanciation des IP de la bibliothèque et de circuits de contrôle configurables dynamiquement. Le circuit final est une architecture flot de données avec un contrôle distribué décrit au chapitre 6.

Si le FPGA cible est non configuré, le code HDL du circuit est généré avec les fichiers de configuration des chemins de données pour l'application, puis synthétisé pour ce FPGA. Autrement, seuls les fichiers de configuration sont produits.

Plutôt que de générer un automate d'état (contrôle centralisé) pour ordonner les calculs sur les opérateurs, nous avons choisi d'instancier des composants permettant le déclenchement des calculs sur la disponibilité des données en entrée (architecture dite « data driven »). L'architecture proposée est décrite en détail dans le chapitre 6. L'ensemble de ces composants forme l'adaptateur d'interface entre l'IP et le réseau.

La granularité des IP a un impact fort sur les performances de l'application générée. En effet, une granularité faible implique de nombreuses tâches et donc un trafic important dans le réseau. Par conséquent, la part de la latence qui est due à la communication dans le NoC est importante. Privilégier une grosse granularité permet de limiter le trafic réseau et permet de bénéficier de l'optimisation des IP.

En revanche, dans le cas où la faible granularité permet de mettre en évidence un plus grand parallélisme, la latence qui en découle peut être compensée par l'exécution en parallèle des tâches. Sans compter que le NoC permet les communications en parallèle si les tâches s'exécutent sur des nœuds différents.

Dans le flot proposé, c'est l'utilisateur qui décide de la granularité des fonctions appelées et donc des IP sélectionnées.

Les outils HLS classiques s'arrêtent à la génération et à la simulation du code RTL. Or, l'exploitation du FPGA comme co-processeur demande un effort de développement supplémentaire. En effet, l'interface du système doit être connectée aux entrée/sortie de la carte FPGA ciblée. Grâce à la bibliothèque matérielle, l'outil *SyntheSys* est capable de générer les fichiers de contraintes propres à un FPGA et à une carte donnée. Il dispose pour cela des informations sur les horloges disponibles, leurs fréquences, les pattes du FPGA connectées aux oscillateurs et le code VHDL pour instancier les gestionnaires d'horloge.

SyntheSys permet aussi de générer le fichier de programmation du FPGA en utilisant les flots de synthèse logique des constructeurs (Xilinx, Altera, etc.). Pour libérer l'utilisateur des outils constructeurs (licence, maitrise des outils), *SyntheSys* permet d'exécuter la synthèse logique à distance en utilisant le protocol SSH.

3.2.6 Programmation de l'architecture

Le FPGA configuré avec le circuit généré doit d'abord être programmé pour fonctionner. La dernière étape est donc de configurer le circuit et de programmer l'application sur l'architecture. C'est-à-dire que les chemins de données sont écrits dans des mémoires distribuées au niveaux des nœuds du réseau.

La notion de programme diffère de l'acception classique (suite d'instructions). Il s'agit ici des dépendances de données, identifiées par une tâche (un calcul réalisé sur une IP) et le chemin que parcourt le paquet transportant le résultat de ce calcul dans le réseau.

En effet, dans le système généré les données sont échangées sous la forme de paquets. L'adresse de destination du paquet est contenue dans son en-tête. Les chemins des données sont déterminés par les en-têtes des paquets émis. L'étape de programmation consiste donc à charger en mémoire de chaque nœud, les en-têtes qui seront utilisés pour envoyer les données produites par chaque tâche exécutée. Le mécanisme utilisé est expliqué en détail dans le chapitre 6.

3.2.7 Vérification du système et prototypage

Comme beaucoup d'outils de synthèse HLS, les données récupérées lors de l'exécution du banc de test logiciel peuvent être chargées en entrée de l'application FPGA en respectant le format des paquets du NoC, pour la simulation ou pour l'exécution finale sur FPGA.

Dans le cas de la simulation, l'outil *SyntheSys* génère automatiquement un banc de test en VHDL pour tout le système. Le banc de test logiciel est exécuté afin de recueillir l'ensemble des données d'entrée à envoyer au système. Ces données sont ensuite converties en paquets puis en signaux pour respecter l'interface du NoC. En effet, lors de la simulation, aucune IP de communication

n'est sélectionnée. Les paquets sont directement injectés dans le réseau au niveau du port local du routeur à l'adresse du nœud réservé à la communication. L'outil développé supporte plusieurs simulateurs (Modelsim, Isim, Vivado Simulator).

Dans le cadre du prototypage, les données recueillies à l'exécution du banc de test logiciel sont aussi converties en paquets puis directement sauvegardées dans un fichier qui peut être envoyé au FPGA en utilisant les pilotes de communication fournis. Aujourd'hui seul le PCI express est supporté.

3.3 Les bibliothèques

3.3.1 Gestion du logiciel, du code HDL et du matériel

Il existe 3 bibliothèques dans l'outil développé : une bibliothèque d'IP, une bibliothèque de fonctions logicielles et une bibliothèque de cartes FPGA. La bibliothèque d'IP est conçue pour permettre :

1. la sélection des opérateurs en fonction de l'architecture FPGA ciblée,
2. la description des propriétés des opérateurs (latence, intervalle d'initialisation, fréquence, etc.),
3. la spécification des interfaces pour la génération automatique du code HDL.

La bibliothèque de fonctions logicielles permet de fournir un ensemble de fonctions dont les accélérateurs sont disponibles en bibliothèque d'IP. Le concepteur d'algorithme doit composer avec elles ou demander au fournisseur d'IP d'ajouter celles dont il a besoin. Le choix de la taille des fonctions à accélérer détermine le niveau de granularité des IP et des données échangées.

Nous avons choisi un référencement des opérateurs utilisant des fichiers au format XML de la même manière que [VPCM09]. Notre logiciel de gestion de bibliothèque est indépendant de la notion de réseau sur puce. Le NoC étant lui même référencé en tant que composant HDL paramétrable. Ce qui permet de changer de NoC sans modifier profondément le logiciel de génération automatique, pour peu que l'interface de communication et le mode d'adressage des paquets reste le même.

Comme un des objectifs est que le concepteur d'algorithme ne spécifie aucun détail architectural, les informations concernant les opérateurs (IP/description HDL) et propriétés des FPGA (ressources, interfaces, compatibilités) doivent être ajoutées au préalable en bibliothèque par le fournisseur d'IP. L'organisation de la bibliothèque est présentée sur la figure 3.10.

Les fonctions logicielles sont définies et liées aux composants matériels (IP) grâce à une organisation en trois niveaux d'abstraction de la bibliothèque logicielle : fonction logicielle, service et modules. Les modules sont référencés comme compatibles avec certaines familles de FPGA. Ces dernières utilisent un flot de synthèse FPGA spécifique (Vivado, ISE, Quartus, etc.). L'outil de synthèse

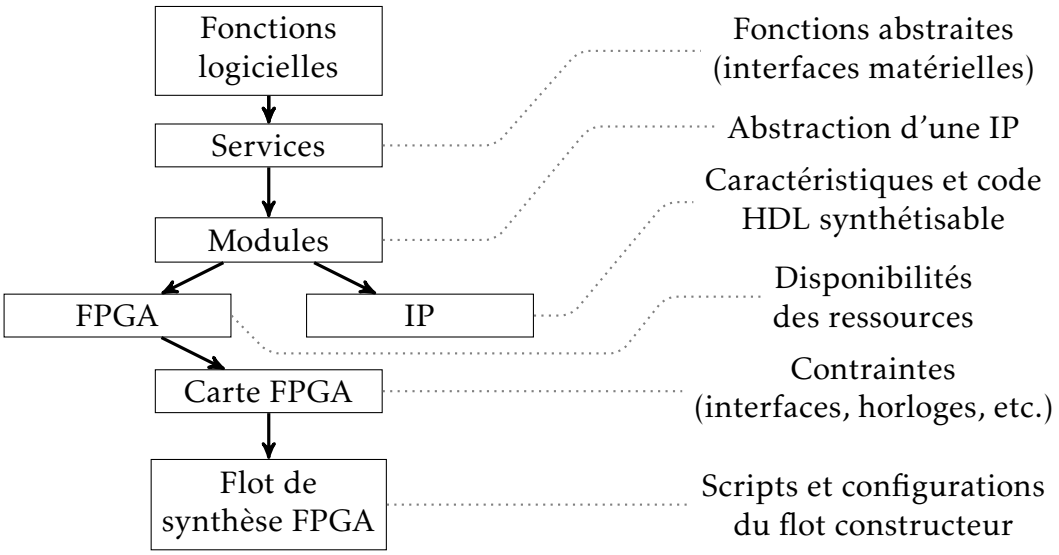


FIGURE 3.10 : Organisation des bibliothèques

est assez générique pour supporter tout type de flot constructeur. Aujourd'hui, les flots ISE, Vivado et Quartus sont supportés.

La bibliothèque de FPGA consiste à associer à chaque carte FPGA supportée, ses caractéristiques (entrées/sorties, fréquences d'horloge, ressources disponibles et flot de synthèse logique utilisé).

3.3.2 Sélection des opérateurs

La sélection des opérateurs repose sur les informations d'implantation de chaque opérateur pour un FPGA donné : la consommation en ressources FPGA (LUT, registres, blocs RAM), la latence, la fréquence de fonctionnement et l'intervalle d'injection des entrées. Les opérateurs dédiés au FPGA ciblé sont choisis en priorité. Les informations de consommation en ressources matérielles du FPGA sont utilisées pour dimensionner le réseau (voir chapitre 5) c'est-à-dire combien d'opérateurs peuvent être synthétisés – avec un seuil de 80% d'utilisation de ressources pour limiter les temps de synthèse logique.

Dans un langage de programmation dynamiquement typé, les fonctions peuvent être appelées avec des arguments de types différents. De la même manière, un accélérateur matériel peut être implémenté différemment pour chaque FPGA pour lequel il est compatible (des IP différentes). Pour un même FPGA, il peut aussi y avoir plusieurs implémentations résultant de compromis différents (ressources/performances). C'est pourquoi le choix d'une IP lors de l'appel d'une fonction logicielle suit deux étapes de sélection : celle du *service* puis celle du *module*. Cette sélection est décrite par la figure 3.11.

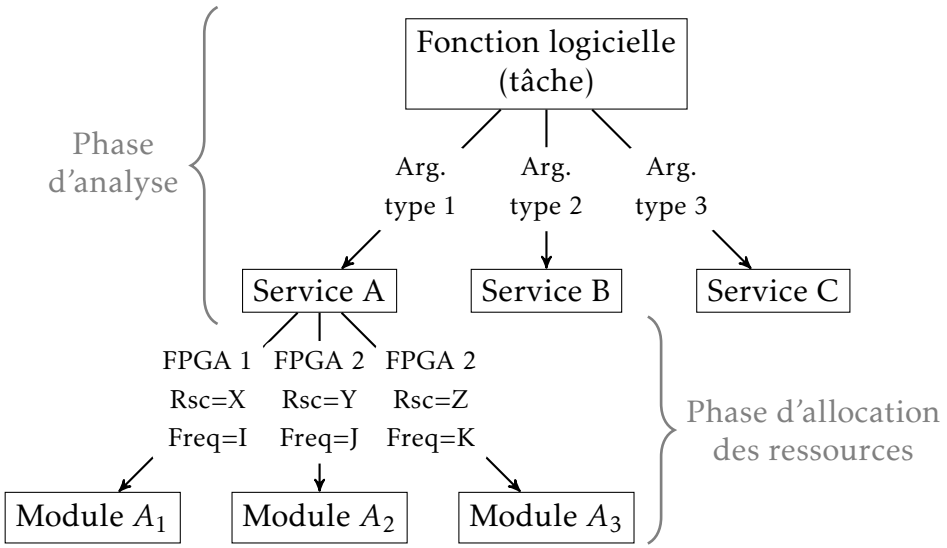


FIGURE 3.11 : Sélection d'un module HDL de la bibliothèque (une IP) lors de l'appel d'une fonction logicielle

3.3.3 Le NoC utilisé

Le NoC utilisé (AdOCNet pour « Adjustable On-Chip-Network ») a été développé spécifiquement pour ces travaux de thèse afin de faciliter son intégration dans le flot de synthèse. En outre, l'entreprise Adacsys préferait concevoir, utiliser et fournir son propre NoC au sein de ses produits.

3.3.3.1 Structure du NoC

Conçu pour consommer peu de ressources matérielles sur le FPGA, il est très proche structurellement du NoC *Hermes* [MCM+04].

Il a une topologie maillée 2D et utilise une technique de commutation par paquets (« wormhole »). Le routage des paquets suit un algorithme déterministe XY (dit « X first ») pour garantir que les données arrivent toujours dans l'ordre dans lequel elles ont été envoyées. Le contrôle de flux supporté suit le protocole poignée de main synchrone.

Les mémoires tampons sont disposées en entrée des ports des routeurs et sont paramétrables individuellement.

Au niveau qualité de service, le réseau AdOCNet ne propose qu'une gestion de type « best effort » pour la transmission des paquets. La taille du NoC est alors minimale (pas de mécanisme de priorisation des paquets, ni de canaux virtuels) mais la latence des paquets transmis est très sensible à la congestion dans le réseau. Nous estimons que pour une application donnée, un placement optimisé permet de limiter les phénomènes de congestion.

3.3.3.2 Utilisation pour la HLS

AdOCNet est un réseau modulable généré automatiquement par un logiciel nommé YANGO (pour « Yet Another NoC Generator and Optimizer »). Tous ses composants HDL et leurs instanciations sont paramétrables par l'interface de programmation de YANGO. Celle-ci est utilisé par l'outil de synthèse de haut niveau développé pour cette thèse (*SyntheSys*) afin d'automatiser l'optimisation des paramètres du NoC.

Trois aspects du flot de synthèse présenté font l'objet d'une contribution particulière. L'analyse est présentée dans le chapitre 4, l'estimation des ressources est présentée au chapitre 5 et l'architecture flot de données générée est présentée au chapitre 6.

Chapitre 4 : Analyse symbolique

Le signe premier de la certitude scientifique, c'est qu'elle peut être revécue aussi bien dans son analyse que dans sa synthèse.

Gaston Bachelard

Sommaire

Dans ce chapitre, nous présentons en détail la méthode d'analyse utilisée pour extraire les dépendances de données d'un programme donné afin de produire un modèle de calcul utilisable pour la synthèse de circuit. Nous commençons par identifier ce qui est de l'ordre de l'algorithme et ce qui est de l'ordre de l'architecture dans un programme. Nous montrons alors comment un langage avec un haut niveau d'abstraction peut faciliter l'analyse des dépendances de données. Ensuite, nous détaillons la méthode appliquée et les choix techniques mis en œuvre dans l'outil *SyntheSys* développé. Enfin nous nous focalisons sur l'expressivité du modèle flot de données généré et son impact sur le comportement des accélérateurs référencés en bibliothèque.

4.1 Contexte et motivations

4.1.1 Entre algorithme et architecture

Les premiers programmes étaient écrits dans un langage compréhensible pour une machine donnée. Puis, étant donné la complexité croissante et la diversité de ces machines, des langages de plus en plus abstraits ont été conçus pour des raisons de portabilité et d'optimisation mais ils nécessitent des compilateurs performants. L'objectif est toujours le même : exploiter au mieux le matériel. Un programme optimisé fait donc un usage optimisé des spécificités architecturales.

Par conséquent, dans un programme, cohabitent les notions d'algorithme (ordre nécessaire des calculs) et d'architecture (potentiel en calcul, en mémoire, etc).

Dans le cadre de la synthèse d'architecture, une difficulté est d'extraire l'ordre nécessaire (la signification de l'algorithme) de l'ordre effectif imposé par des contraintes architecturales. Allen et Kennedy [AK02] nous disent que :

> « Any reordering transformation that preserves every dependency in a program preserves the meaning of that program. »

Ce qui signifie que le concept d'algorithme réside dans les *dépendences de données* d'un programme. Autrement dit, l'analyse des dépendances des données suffit pour reproduire un algorithme sur une nouvelle architecture disposant des opérateurs nécessaires.

4.1.2 Dépendances de données

Les dépendances de données représentent des contraintes temporelles de production et de consommation qui imposent un ordonnancement partiel (parfois total) des tâches.

Lorsque les dépendances de données sont telles que les données se consomment au même rythme que de nouvelles sont produites, elles sont de type « flot ». La même zone mémoire peut donc être réutilisée par un programme pour stocker plusieurs résultats de calcul au cours du temps.

Or la mémoire est une contrainte architecturale. Les dépendances entre les données sont indépendantes de la manière dont ces dernières sont stockées. Pourtant, la manière avec laquelle le concepteur d'algorithme utilise la mémoire gène parfois les outils d'analyse. C'est pourquoi nous proposons une méthode basée sur une *analyse symbolique* qui est indépendante de la manière dont le programmeur utilise la mémoire.

Outre les dépendances de données, certaines informations architecturales sont cependant utiles pour la synthèse. Le type d'une donnée permet de contraindre la sélection des opérateurs, par exemple.

4.1.3 Intérêt de l'approche pour la HLS

Dans les langages impératifs, les dépendances de données sont implicites. Il est donc difficile, même pour celui qui connait son algorithme, de choisir le meilleur moyen de paralléliser son programme. L'encapsulation des données dans des objets rend la tâche encore plus ardue.

Les outils de HLS actuels comptent sur les indications des utilisateurs pour faciliter les décisions qui concernent la parallélisation. Or, ces décisions sont dépendantes de l'architecture utilisée. Par exemple, le déroulage de boucle ne peut se faire que si l'architecture cible dispose des ressources matérielles pour son exécution parallèle. Certaines techniques, comme l'analyse polyhédrique, permettent de paralléliser efficacement des boucles dont les bornes sont linéaires. Autrement, la plupart des outils de parallélisation échouent à produire un résultat efficace.

Nous avons besoin de technique plus efficace et éviter à l'utilisateur de devoir indiquer comment considérer les boucles pour le parallélisme (déroulement partiel ou total, fusions, etc).

Comme présenté en partie 2.3.2, les outils d'analyse doivent être capables d'intégrer les améliorations suivantes :

1. Le code d'entrée doit être lisible par les concepteurs d'algorithmes (sans connaissances des architectures matérielles).
2. Les outils HLS doivent supporter des concepts avancés de programmation (orienté objet, programmation fonctionnelle, typage dynamique et programmation parallèle).
3. Pour les algorithmes parallélisables, des architectures parallèles efficaces doivent être générées sans modifier le code d'entrée.
4. Il doit être possible d'augmenter les performances du circuit généré en modifiant et en synthétisant successivement son code d'entrée (processus d'optimisation orienté conception).
5. Les circuits générés doivent avoir des performances au moins comparables, sinon meilleures que celles des circuits RTL écrits manuellement.
6. Les outils doivent être ouverts et extensibles.
7. Les environnements de développement doivent être transparents. Idéalement, il n'est pas nécessaire de savoir que l'on fait de la synthèse.

Le premier point nécessite une extraction du parallélisme sans l'aide de l'utilisateur. L'analyse symbolique permet d'exposer l'ensemble des dépendances de données, donc le parallélisme potentiel, sans intervention de l'utilisateur.

Le deuxième point concerne les concepts avancés des langages de programmation. Ces derniers permettent d'exprimer de manière plus abstraite un algorithme (indépendance avec le matériel). Par exemple, il est possible de grouper des données de types différents dans des objets abstraits (programmation orientée objet) ou encore d'exécuter une même fonction avec différents types de données (polymorphisme). L'analyse symbolique permet de s'affranchir de la manière dont le

programmeur encapsule ses données. De plus la gestion du polymorphisme est facilitée car la sélection des opérateurs est faite à l'exécution.

Si de plus, les utilisateurs utilisent des processus légers (« threads ») dans leurs programmes, il est nécessaire de prendre en compte les mécanismes de synchronisation qui, de surcroît, sont souvent propres au système d'exploitation utilisé. Une analyse symbolique est indépendante des mécanismes de synchronisation.

Le troisième point concerne la parallélisation pour plusieurs architectures. Notre approche permet d'abord d'exposer le parallélisme puis d'ordonner les tâches sur les opérateurs (partage des ressources) en fonction des contraintes architecturales.

Pour le quatrième point, l'utilisateur peut influer sur les performances du circuit final en jouant sur le niveau de granularité de son application à l'aide des fonctions de bibliothèque fournies.

Pour répondre aux besoins des points cinq et six, les bibliothèques fournies sont extensibles, ouvertes et écrites par des experts matériels. Mais cela ne suffit pas à garantir de meilleurs performances. La synthèse basée sur les réseaux doit ensuite minimiser l'impact de la communication sur la latence total de l'application.

Le septième point est rendu possible en associant l'analyse symbolique à l'utilisation de gestionnaires de contextes de certains langages interprétés.

Nous montrons donc, dans ce chapitre, qu'il est tout à fait possible de résoudre la plupart de ces problèmes en utilisant un procédé d'analyse symbolique reposant sur les propriétés d'un langage de programmation de haut niveau d'abstraction.

4.2 Description de la méthode proposée

Dans cette partie, nous commençons par décrire l'approche générale et les caractéristiques dont nous avons besoin afin de choisir un langage de programmation approprié. Ensuite, nous décrivons les techniques utilisées pour réaliser l'analyse symbolique, les problèmes rencontrés et les solutions que nous proposons.

4.2.1 Analyse symbolique

L'*analyse symbolique* du programme consiste en son exécution, ou plus exactement son interprétation, après substitution de ses arguments d'entrée par des *symboles* qui, reliés entre eux par des relations de type producteurs/consommateurs, définissent un graphe de dépendance de données.

L'interprétation du programme est altérée, c'est-à-dire que du code supplémentaire est exécuté, afin d'extraire les informations dont nous avons besoin. Les altérations principales sont l'enrichissement d'un graphe pendant l'initialisation

des variables (les *symboles*) comme au moment de l'appel des fonctions de la bibliothèque et l'exécution de chaque branchement conditionnel du programme.

Nous proposons d'analyser les dépendances de données d'un programme en s'appuyant sur les possibilités :

— d'encapsulation (programmation orientée objet),

— de typage dynamique,

— et d'introspection.

La figure 4.1 montre l'algorithme exécuté lors de l'analyse d'une addition de deux variables *a* et *b*. La fonction *Addition* (appelée lors de l'exécution de la fonction à analyser) :

— teste le type des arguments,

— ajoute les acteurs appropriés au graphe flot de données,

— ajoute éventuellement des acteurs pour la conversion de type,

— et retourne un symbole à la place du résultat.

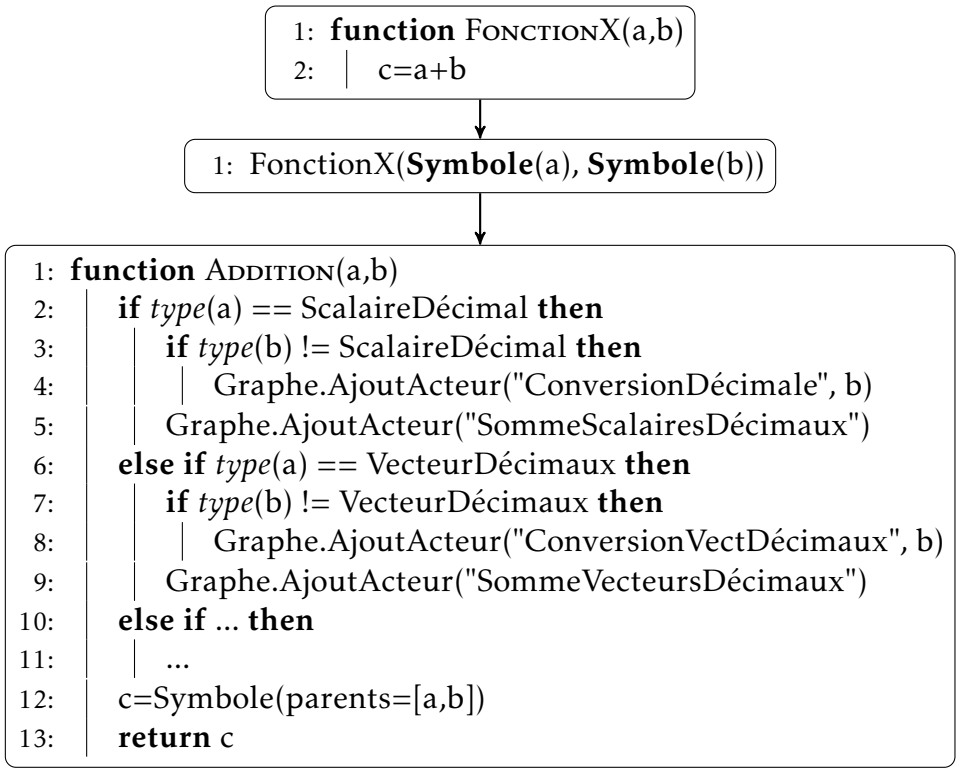


Figure 4.1 : Analyse symbolique

4.2.2 Choix du langage

Pour implémenter notre méthode, le langage utilisé doit supporter les fonctionnalités suivantes :

— être orientée objet,

— être dynamiquement typé,

— supporter l'introspection.

L'introspection n'est possible que pour les langages interprétés. Parmi les quelques langages compatibles avec ces contraintes, nous avons choisi Python pour la richesse de sa bibliothèque, sa popularité et le fait qu'il soit une alternative à Matlab pour le calcul scientifique.

Ce langage a un haut niveau d'abstraction ce qui lui permet de s'abstraire de notions architecturales des CPU comme les pointeurs (C, C++). Le typage dynamique est largement utilisé durant l'analyse. Pour cette raison, notre méthode ne permet pas l'utilisation d'extensions C/C++.

Les symboles peuvent être insérés en utilisant des gestionnaires de contexte nommés *décorateurs* [DH14] de la même manière que [SAK12] (voir figure 4.2). Ils permettent de changer le contexte d'appel des fonctions décorées.

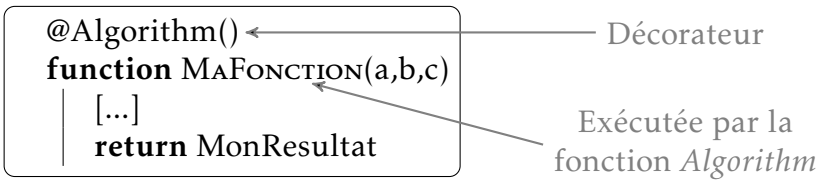


FIGURE 4.2 : Utilisation de gestionnaires de contextes pour spécifier la fonction à accélérer

Dans ce langage orienté objet, les opérations standard sont implémentées en tant que *méthodes* des opérandes. Les symboles utilisés pour l'analyse sont des objets dont les *méthodes* enrichissent un graphe de dépendances de données lorsqu'elles sont appelées. C'est à ce moment que ces fonctions sauvegardent la tâche à réaliser et le type des données qu'elles consomment et produisent.

La figure 4.3 montre les méthodes exécutées lors de l'utilisation des opérateurs standard. Par exemple, la fonction *__pow__*, appelée pour l'opération *puissance* (b**2), rajoute une tâche « POW » au modèle flot de données, puis rajoute les relations de dépendances entre les symboles d'entrée (b et 2) et la tâche « POW », puis retourne un nouveau *symbole* (le résultat) qui pourra ensuite être utilisé, à son tour, comme entrée d'une autre fonction (ici *__sub__*).

Dans le cas où ce sont des fonctions disponibles en bibliothèque qui sont appelées, elles réalisent les mêmes actions que pour les opérateurs standard. Les fonctions correspondantes à tous les opérateurs standard de Python pour les *symboles* ont été surchargées pour l'analyse.

4.2.3 Utilisation du typage dynamique

Pour pouvoir remplacer les arguments d'une fonction par ces symboles, notre méthode repose sur le typage dynamique du langage utilisé. L'algorithme 2

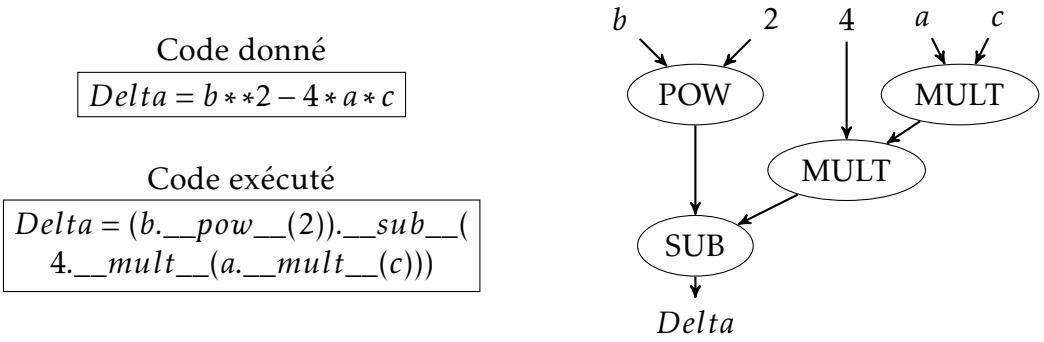


Figure 4.3 : Exemple d'analyse symbolique avec les opérateurs standard

montre l'utilisation de cette propriété où la méthode *Ajout* de la classe *Coût* est appelée successivement avec un objet *Pain*, un entier, un flottant et une liste d'autres objets.

```
1: Total=Coût()
2: for Elmt in [Pain(), 9, 1.02, list(Jambon(), Fromage(), Salade(), Sauce() )] do
3: |   Total += Elmt
```

Algorithme 2 : Exemple d'utilisation du typage dynamique

La figure 4.4 montre comment peut être analysé l'algorithme 2. Étant donné que l'opérateur « += » est appelé avec des arguments de types différents, l'analyse doit prendre en compte la conversion du type (nœuds *Décimal* et *Vecteur*) et l'identification de tâches differentes (addition de scalaires ou sommes des éléments d'un vecteur).

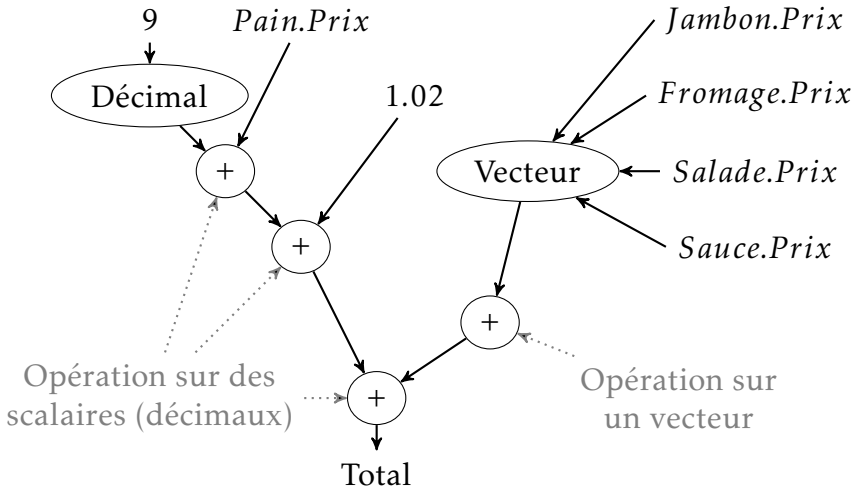


Figure 4.4 : Graphe flot de données produit après analyse de l'algorithme 2

Toute variable modifiée par une fonction est soit locale, soit globale. Les résultats sont soit retournés, soit passés en paramètres (référence). Les variables

locales sont réinitialisées à chaque appel de la fonction. Par conséquent, et contrairement à certains circuits, une fonction ne dispose pas de registres internes pouvant modifier son taux de production des données entre deux opérations successives.

Pour l'analyse proposée, une donnée peut aussi bien être un vecteur qu'un objet quelconque. Donc une tâche donnée, identifiée par l'appel d'une fonction avec des données d'entrée, a une production de données constante. En effet, un vecteur peut avoir une taille variable mais restera toujours une donnée (un objet vecteur).

Autrement dit, les acteurs (ou tâches) associés au graphe flot de données généré satisfont les contraintes des SDF. L'analyse d'un programme sans branchement conditionnel produit donc un graphe de type flot de données synchrone (SDF).

Mais la plupart des programmes contiennent des branchement conditionnels. Utiliser la technique proposée pose alors trois difficultés. La première est la détection des branchements qui sont mutuellement exclusifs. La deuxième est l'analyse de chaque branche dans le cas des boucles et des fonctions récursives. La troisième est comment modifier le modèle pour gérer ces branchements.

4.2.4 Gestion des branchements conditionnels

La méthode proposée ne permet d'exécuter qu'une partie du programme : les branches du code prises lors de l'exécution. Il est donc nécessaire de détecter les branchements pour ensuite considérer chacune des branches.

Pour y arriver, nous profitons de la propriété suivante de Python : tous les tests consistent en l'appel de la méthode « __bool__ » de l'objet testé (version 3 de Python, « __nonzero__ » dans la version 2). La figure 4.5 montre un branchement conditionnel et le code réellement exécuté.

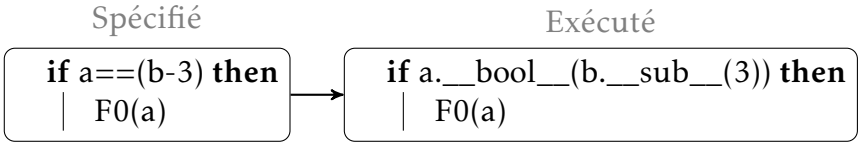


FIGURE 4.5 : Code exécuté lors d'un branchement conditionnel

Comme la méthode « __bool__ » est toujours celle d'un symbole, nous choisissons d'utiliser les possibilités d'introspection pour détecter s'il s'agit d'un « if », un « elsif » ou une boucle (« while » ou « for »).

La figure 4.6 présente la procédure adoptée pour gérer des branchements conditionnels. À l'exécution de la fonction « __bool__ », la première étape est de récupérer le code du bloc exécuté si le test est vrai. Nous utilisons, pour ça encore, l'introspection.

Ensuite, le code du bloc est exécuté grâce à la fonction standard « *exec* », ce qui met à jour une copie des variables locales et globales. Puis les nouveaux

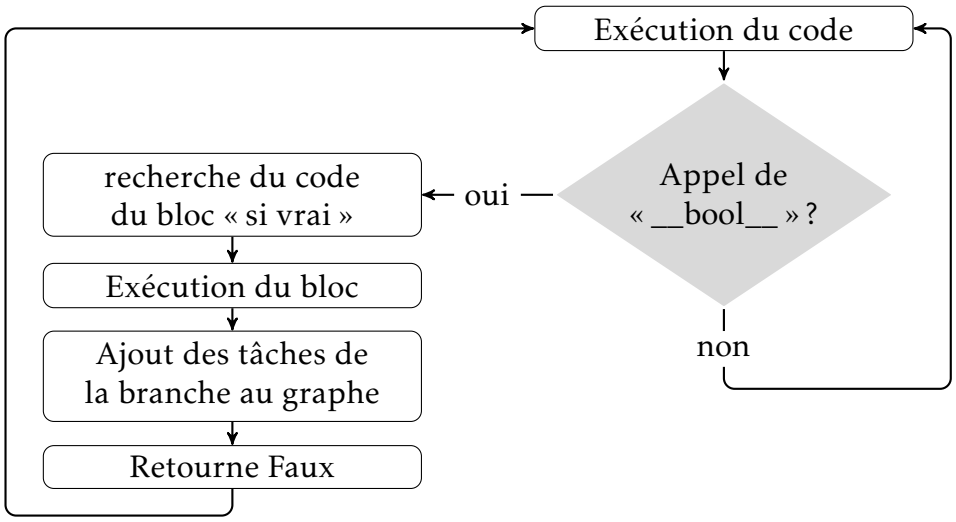


FIGURE 4.6 : Gestion des branchements conditionnels

acteurs sont ajoutés au graphe.

La fonction « __bool__ » doit retourner toujours faux pour continuer avec le reste du programme. Ces étapes sont répétées pour tous les « elif » suivants jusqu'au « else » final (s'il existe) qui est d'abord exécuté de manière isolée pour fusionner les données avec les précedentes dans le modèle. Puis ses instructions sont ignorées dans la poursuite normale de l'exécution.

Afin de prendre en compte les branchements dans le modèle flot de données, les acteurs *switch* et *select* sont ajoutés, respectivement lors des tests booléens et après l'ajout des nouveaux acteurs (fusion des variables de branche).

Un exemple de modèle BDF généré est présenté en figure 4.7.

Dans cet exemple, les tâches associées aux fonctions F1, F2, F3 et F4 sont représentées par des nœuds grisés. Les acteurs *bool*, *switch* et *select* ont des ports d'entrée ou de sortie conditionnels. Les arcs en pointillés représentent les dépendances de contrôle. Les paires de *switch* et *select* permettent chacune de sélectionner une des branches (if/elif/else), c'est-à-dire d'exécuter soit F1, soit F2, soit F3.

4.2.5 Cas des boucles

Deux types de boucles sont analysées différemment selon que leur nombre d'itérations dépende ou non des données.

4.2.5.1 Boucles à nombre d'itérations fixe

Le premier type de boucle est celui dont le nombre d'itérations est indépendant des données. Comme aucun test n'est effectué pour déterminer le nombre d'itérations de ces boucles, elles sont totalement déroulées, dévoilant tout leur

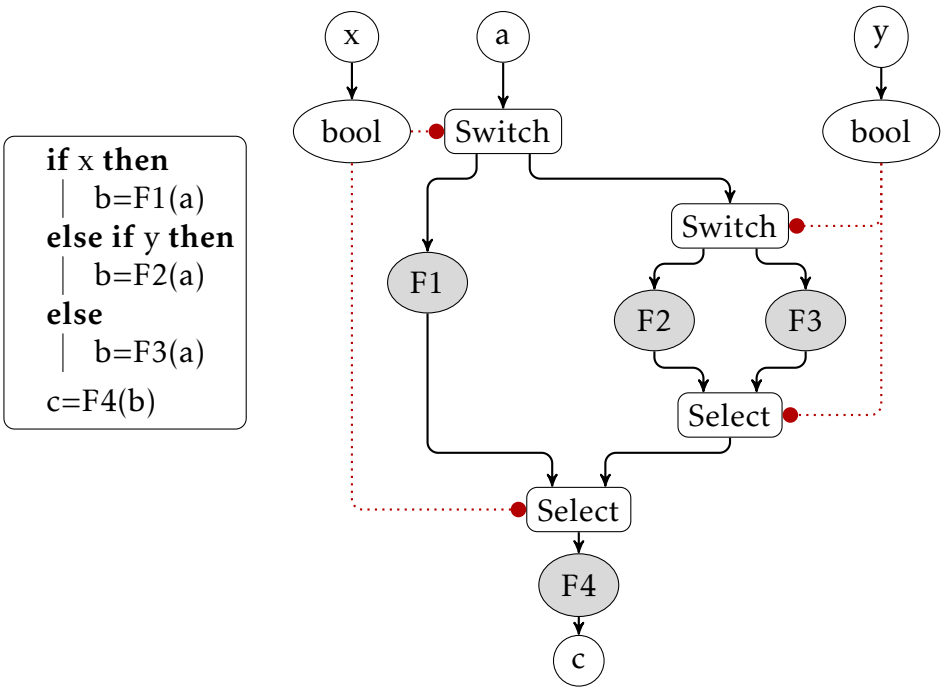


FIGURE 4.7 : Modèle BDF supportant les branchements conditionnels

parallélisme potentiel. La figure 4.8 montre un exemple de modèle généré à l'analyse d'une boucle *for* décrivant un arbre binomial.

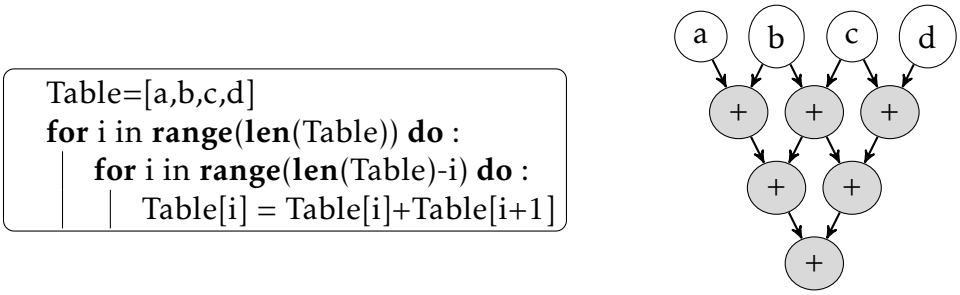


FIGURE 4.8 : Modèle généré après analyse d'une boucle *for* (nombre d'itérations fixe)

Dans cette approche, nous exposons donc d'abord le parallélisme et ensuite nous décidons de partager les tâches si nécessaire et non l'inverse. Le nombre de tâches du graphe est dans ce cas proportionnel au nombre d'itérations de la boucle. Ce dernier peut être très élevé dans certaines applications, ce qui nécessite dans certains cas d'utiliser les mémoires de masse (problème d'extensibilité).

4.2.5.2 Boucles à nombre d'itérations variable

Dans le cas des boucles à nombre d'itérations dépendant des données, un appel à la fonction __bool__ est réalisé pour déterminer le résultat du test et

donc le nombre d'itérations.

La détection du *while* requiert une autre procédure. Comme le test doit être inclus dans le corps de boucle, il est exécuté en plus du corps de boucle avec la fonction « *exec* » (contexte d'exécution isolé). Les acteurs *bool*, *switch* et *select* sont insérés de manière différente par rapport à un simple branchement.

La figure 4.9 montre un exemple de modèle généré après l'analyse d'une boucle. L'acteur *select* est inséré de manière à sélectionner l'entrée ou le retour de boucle. L'acteur *switch* est utilisé soit pour ré-itérer la boucle, soit pour en sortir.

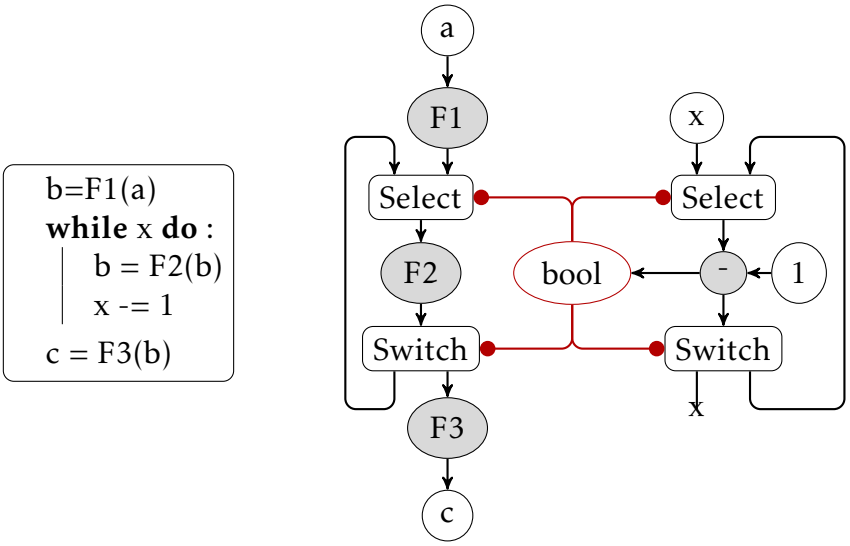


FIGURE 4.9 : Modèle généré après analyse d'une boucle (nombre d'itérations variable)

4.2.6 Retour dans les branchements et fonctions récursives

Deux cas particuliers posent des difficultés : le retour de valeurs dans le code d'un sous bloc après un branchement et les fonctions récursives.

Dans le premier cas, les retours sont détectés lors de la récupération du code des blocs après un test. Puis, les symboles retournés sont fusionnées aux autres valeurs de retour pour chaque branche. La fusion avec les valeurs retournées en dehors des branches n'est possible que lors de la consommation éventuelle du symbole retourné après l'exécution de la fonction.

Dans le deuxième cas, la mise à jour d'une pile d'appel des fonctions est nécessaire pour détecter la récursivité d'une fonction. Nous proposons de faire l'ajout à la pile lors de la création d'une tâche (acteur du modèle) en sauvegardant (ou mettant à jour) la fonction et la ligne de code d'où est appelée cette tâche.

Lorsqu'une tâche, associée à une fonction A est créée depuis une fonction B déjà présente dans la pile, cette dernière doit être mise à jour. Si le numéro de ligne à l'appel de la fonction A est supérieure à celui marqué dans la pile, les

fonctions de la pile au dessus de la fonction B sont supprimées (leur exécution est terminée).

Si au contraire, le numéro de ligne à l'appel de la fonction A est inférieure ou égal à celui marqué dans la pile, alors il y a récursion. Aujourd'hui, l'analyse de fonctions récursives n'est pas supportée mais reste théoriquement possible en utilisant l'introspection pour retrouver la ligne d'appel récursif à ne pas exécuter, lors du prochain branchement.

Si la récursion est indépendante des données d'entrée, alors elle est exécutée sans traitement particulier comme pour le déroulement des boucles à nombre d'itérations fixe.

Cette partie clôt le chapitre concernant l'analyse du programme d'entrée. Le modèle généré peut être traduit en graphe de tâches qui permettra la sélection des IP lors de l'allocation des ressources. Dans le prochain chapitre, nous présentons la méthode d'estimation de ressources qui nous permet d'allouer les IP sous contraintes de ressources.

Chapitre 5 : Estimation des ressources et modélisation du NoC

Tous les modèles sont faux, mais certains sont utiles.

George Box

Sommaire

Ce chapitre présente comment nous avons construit et utilisé des modèles mathématiques afin d'estimer les ressources qu'utilise un NoC sur FPGA. Tout d'abord, la partie 5.1 décrit la méthode employée pour modéliser l'utilisation en ressources d'un NoC. Puis, la partie 5.2 discute des limites des modèles obtenus.

5.1 Présentation de la méthode de modélisation

Les flots de conception existants ne font pas l'exploration de l'espace de conception (DSE pour « design space exploration ») du réseau. La difficulté de la DSE repose sur le nombre de paramètres à définir. En effet, il peut y avoir des milliers de combinaisons des paramètres possibles pour un NoC. Le concepteur doit sélectionner une topologie, un nombre de nœuds, la taille des mémoires tampons, la taille des connexions (flit), etc. Ces paramètres doivent être déterminés avant l'étape de placement (voir figure 3.5).

Les FPGA sont des circuits au nombre de ressources limité mais dont la densité d'intégration suit aussi la loi de Moore. Une DSE peut nécessiter de nombreuses évaluations des ressources utilisées avant de trouver une configuration adéquate. Étant donné que la durée des synthèses logiques dépend de la taille du système (nombre de portes logiques) et du FPGA, effectuer une synthèse par configuration pendant la DSE n'est pas raisonnable en termes de temps.

Pour éviter ces temps d'exploration trop importants, il est nécessaire d'effectuer une estimation rapide des ressources. Un modèle mathématique permet une évaluation rapide de l'impact du paramétrage du NoC sur l'utilisation des ressources.

Notre contribution consiste à proposer un modèle mathématique pour les NoC afin d'accélérer la DSE et estimer la consommation en ressources sur FPGA. Les explorations sont contraintes, d'une part par la cible FPGA choisie et d'autre part par le graphe de tâches de l'application. L'objectif est de dimensionner le système pour la phase d'allocation des ressources du flot de synthèse HLS proposé au chapitre 3.

La méthode que nous proposons est illustrée en figure 5.1 et s'effectue en trois étapes :

1. Définition du NoC : sélection d'une structure appropriée de NoC (routage, topologie, etc.).
2. Collecte des données requises pour l'analyse.
3. Analyse des données afin d'identifier le type de modèle mathématique le plus adéquat pour différentes configurations du NoC.

Le principe de notre approche est d'abord de sélectionner une structure de NoC pour connecter les IP d'une application. Ensuite, les données (les ressources FPGA) sont collectées en réalisant un certain nombre de synthèses logiques, avec différentes combinaisons de paramètres (en faisant varier un seul paramètre à la fois). L'analyse des résultats permet de trouver les modèles les plus appropriés et leurs variables associées. Ensuite, le concepteur peut construire les modèles pour chaque type de ressource en fonction des paramètres du NoC choisi. Enfin, les modèles sont validés à la fin de l'étape 3.

Cette étape de modélisation est effectuée une fois pour toute pour un NoC et un FPGA donné. Les modèles mathématiques obtenus peuvent ensuite être réutilisés par un outil de synthèse pour estimer les ressources utilisées par le NoC choisi. La DSE peut ainsi être accélérée car la phase d'évaluation des ressources par synthèse logique (chronophage) n'est plus nécessaire.

Détaillons maintenant chacune des étapes.

5.1.1 Étape 1 : sélection du NoC

La première étape est la sélection d'une structure appropriée de NoC. Toute structure de NoC décrite en langage HDL peut être utilisée. Dans nos expérimentations, nous avons utilisé deux NoC différents. L'un a été spécialement

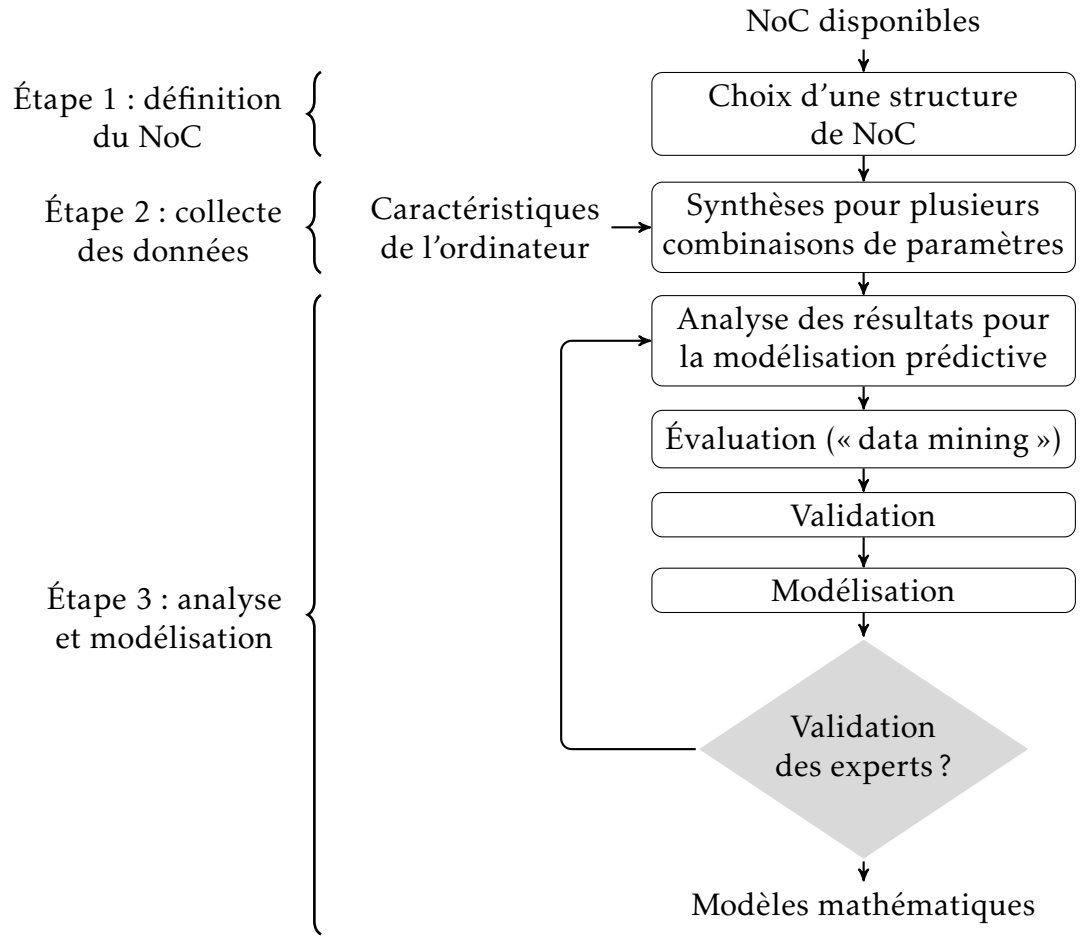


Figure 5.1 : Méthode de construction des modèles

	Topologie	Contrôle de flux	Canaux virtuels	Algorithme de routage
Hermes	Maillé 2D	Par crédit	2	XY statique
AdOCNet	Maillé 2D	Poignée de main synchrone	-	XY statique

Table 5.1 : Comparaison des deux structures de NoC utilisées

conçu pour l'outil de synthèse (AdOCNet - « adjustable on-chip network ») et l'autre souvent utilisé pour la recherche universitaire (Hermes [MCM+04]). La structure de NoC (topologie, contrôle de flux, canaux virtuels, mode d'arbitrage et algorithme de routage) est fixée pour un modèle. Le tableau 5.1 montre les caractéristiques des deux NoC utilisés pour la modélisation.

Nous avons choisi deux NoC de même topologie (maillée 2D) car celle-ci est proche de la topologie physique du FPGA. Les deux réseaux présentés diffèrent par leur flot de contrôle et leurs nombres de canaux virtuels. Ces différences ont un impact significatif sur la consommation en ressources du réseau.

5.1.2 Étape 2 : collecte des données

Les données récoltées sont extraites des rapports de synthèse logique de l'outil Vivado [Viv] pour les FPGA Xilinx, pour différents paramètres du NoC choisi. Les ressources utilisées peuvent différer en fonction des FPGA, mais elles sont toujours de même type : LUT (« look-up tables »), registres (basés sur des bascules - ou « flip flop ») et mémoire RAM (distribuée ou en blocs). Les mémoires distribuées utilisent des LUT et sont appelées MLUT (« Memory LUT ») tandis que les mémoires en blocs sont appelées BRAM (pour « Block RAM »).

Les résultats sont donnés pour les ressources suivantes :

— n_1 : le nombre de routeurs selon l'axe des x
— n_2 : le nombre de routeurs selon l'axe des y
— n_3 : la profondeur des mémoires tampon
— n_4 : la taille du *flit*
— LUT : le nombre de LUT (« look-up table ») utilisés
— MLUT : le nombre de LUT utilisés pour la mémoire
— FF : le nombre de bascules (« flip flop »)

Les options de synthèses sont fixes et l'ordinateur utilisé est le même pour toutes les synthèses.

5.1.3 Étape 3 : analyse et modélisation

Le travail d'analyse et de modélisation est le fruit d'une collaboration avec Mme Catherine Combes de l'équipe « Machine learning » du laboratoire Hubert Curien de Saint-Étienne.

5.1.3.1 Analyse des données

La base de données des résultats observés est analysée afin d'identifier les liens entre les variables et les LUT, MLUT, FF en fonction des variables n1, n2, n3 et n4. Le but final est de modéliser mathématiquement les relations entre la configuration d'entrée du NoC et les ressources matérielles. L'utilisation finale des modèles permetterait d'estimer les ressources consommées sans passer par une étape de synthèse logique.

Nous utilisons les techniques de fouille de données (« data mining ») pour hiérarchiser les liens entre les variables, dans une quantité importante de données. La fouille de données peut être à la fois prédictive et descriptive. Lorsqu'elle est prédictive, son but est de prévoir la valeur d'un attribut particulier étant donné un jeu de valeurs. Lorsqu'elle est descriptive, son but est de reconnaître des motifs qui décrivent de manière synthétique les relations entre les données. C'est donc une technique à part entière de découverte de connaissances, qui est le processus global de conversion des données brutes en connaissances en sélectionnant des informations utiles, à partir des données.

Cinq méthodes de fouille de données ont été identifiées [DHS12, RRS11] :

— modélisation prédictive,

— sélection des attributs,

— analyse d'associations,

— analyse des groupes (« cluster »),

— et détection d'anomalies.

Dans notre approche, la fouille de données la plus appropriée est la modélisation prédictive car notre but est la prédiction des consommations en ressources. L'objectif, ici, est donc de construire un modèle prédictif pour une variable cible, basée sur des variables explicatives.

Pour atteindre cet objectif, la classification et la régression sont des moyens efficaces de prédire une finalité discrète ou continue, c'est-à-dire une extrapolation d'une sortie (ce que la valeur future d'une mesure va être). L'analyse de régression est un outil statistique pour l'investigation des relations entre variables. En général, l'investigateur recherche une relation de cause à effet d'une variable sur une autre.

D'abord, nous vérifions les relations linéaires entre les couples de variables en utilisant la corrélation de Pearson. C'est la méthode la plus utilisée en statistique car elle mesure la linéarité alors que la méthode de Spearman mesure la monotonie.

Il est possible d'isoler des variables en termes de corrélation. Pour deux variables, nous avons une paire de valeurs par échantillon. Nous pouvons alors mesurer la distance et la dissimilarité entre deux vecteurs de données pour chaque variable.

La mesure de la similarité entre variables peut être sous forme de coefficients de corrélation ou en utilisant d'autres mesures d'association. Nous avons choisi les coefficients de corrélation. Le résultat de l'analyse d'un groupe est un arbre binomial (ou dendrogramme) avec $n-1$ nœuds, n étant la taille du groupe.

Les branches de cet arbre sont sélectionnées à partir d'un niveau de similarité satisfaisant.

Ensuite, pour la modélisation prédictive, lorsque la valeur finale (ou classe) est discrète et que les attributs sont discrets, la régression linéaire est un premier choix logique. La manière standard d'effectuer la prédiction est d'écrire la valeur finale comme une somme linéaire des valeurs d'attribut avec les poids appropriés tels que :

$$y_{Prédite} = w_0 + w_1.x_1 + w_2.x_2 + ... + w_n.x_n \tag{5.1}$$

où

— $y_{Prédite}$ est la classe,

— $x_1, x_2, ... x_n$ sont les valeurs des attributs de la variable $X_1, X_2, ... X_n$,

— et $w_1, w_2, ... w_n$ sont les poids de chaque variable $X_1, X_2, ... X_n$.

	n_3	n_4	n_2	n_1	FF	LUT	MLUT
n_4	0.033						
n_2	-0.045	-0.045					
n_1	-0.071	-0.071	0.551				
FF	-0.122	-0.124	0.731	0.954			
LUT	-0.093	-0.127	0.721	0.959	0.998		
MLUT	-0.340	-0.292	0.596	0.789	0.876	0.850	
$n_1 \times n_2$	-0.089	-0.089	0.733	0.959	0.997	0.999	0.834

TABLE 5.2 : Tableau des corrélations de Pearson pour AdOCNet

	n_4	n_2	n_1	FF	LUT	MLUT
n_2	0.170					
n_1	-0.157	-0.237				
FF	0.364	0.509	0.531			
LUT	0.460	0.503	0.467	0.992		
MLUT	0.627	0.463	0.339	0.932	0.971	
$n_1 \times n_2$	-0.056	0.468	0.680	0.880	0.812	0.649

TABLE 5.3 : Tableau des corrélations de Pearson pour le NoC Hermes

De cette manière, nous obtenons une équation de régression à utiliser pour déterminer les poids correspondants de chaque variable. C'est une procédure bien connue en statistique. Les poids sont calculés à partir de données résiduelles. Le modèle minimise cette somme des carrés des écarts en choisissant les coefficients appropriés. La régression linéaire est un moyen simple et efficace pour la prédiction discrète et elle est très répandue dans les applications statistiques. Cependant, elle est très sensible aux données aberrantes.

La différence entre la valeur observée ($y_{Observée}$) et la valeur prédite ($y_{Prédite}$) est appelée la valeur résiduelle. Chaque point est associé à une valeur résiduelle.

$$V_{Residuelle} = V_{Observée} - V_{Prédite} \tag{5.2}$$

La somme, et par conséquent la moyenne des valeurs résiduelles, sont égales à 0. C'est-à-dire : $\sum e = 0$ et $\bar{e} = 0$.

Les tableaux 5.2 et 5.3 listent les coefficients de corrélation de Pearson pour AdOcNet 5.2 (29 synthèses) et Hermes 5.3 (152 synthèses) [Pea20], respectivement. Le choix du nombre de synthèses incombe à l'expert qui évalue le nombre de points nécessaires pour avoir un modèle précis.

Lorsque le coefficient de corrélation de Pearson est supérieur à $0,7$ ou inférieur à $-0,7$ les points sont considérés comme proches. Une corrélation parfaite est de 1 ou -1 : les points sont confondus.

Concernant les deux NoCs, il y a une corrélation positive forte entre les ressources observées. Lorsque les similarités entre groupes sont analysées, la

plus forte corrélation est trouvée entre les *flip flop* et les *LUT* (respectivement 0,992 pour Hermes et 0,998 pour AdOCNet). De fortes corrélations sont aussi trouvées entre les MLUT et les LUT et entre les FF et les MLUT. Il y a aussi une forte corrélation (moins forte, mais significative) entre $n_1 \times n_2$ et le nombre de FF et de LUT. Une des différences principales entre les deux NoC est le lien entre l'utilisation des ressources et le paramètre n_3 (profondeur des mémoires tampon).

AdOCNet n'utilise que des MLUT, LUT et FF alors que le NoC Hermes utilise des BRAM pour les buffers. n_3 est donc considérée comme une constante pour AdOCNet et une variable pour Hermes. Par conséquent, n_3 n'est pas considérée dans la corrélation de Pearson et l'analyse hiérarchique d'AdOCNet.

Ensuite, nous identifions les groupes de variables fortement corrélées en réalisant une analyse hiérarchique [DHS12]. La figure 5.2 présente les dendrogrammes pour AdOCNet et Hermes. Une corrélation forte indique un haut niveau de similarité.

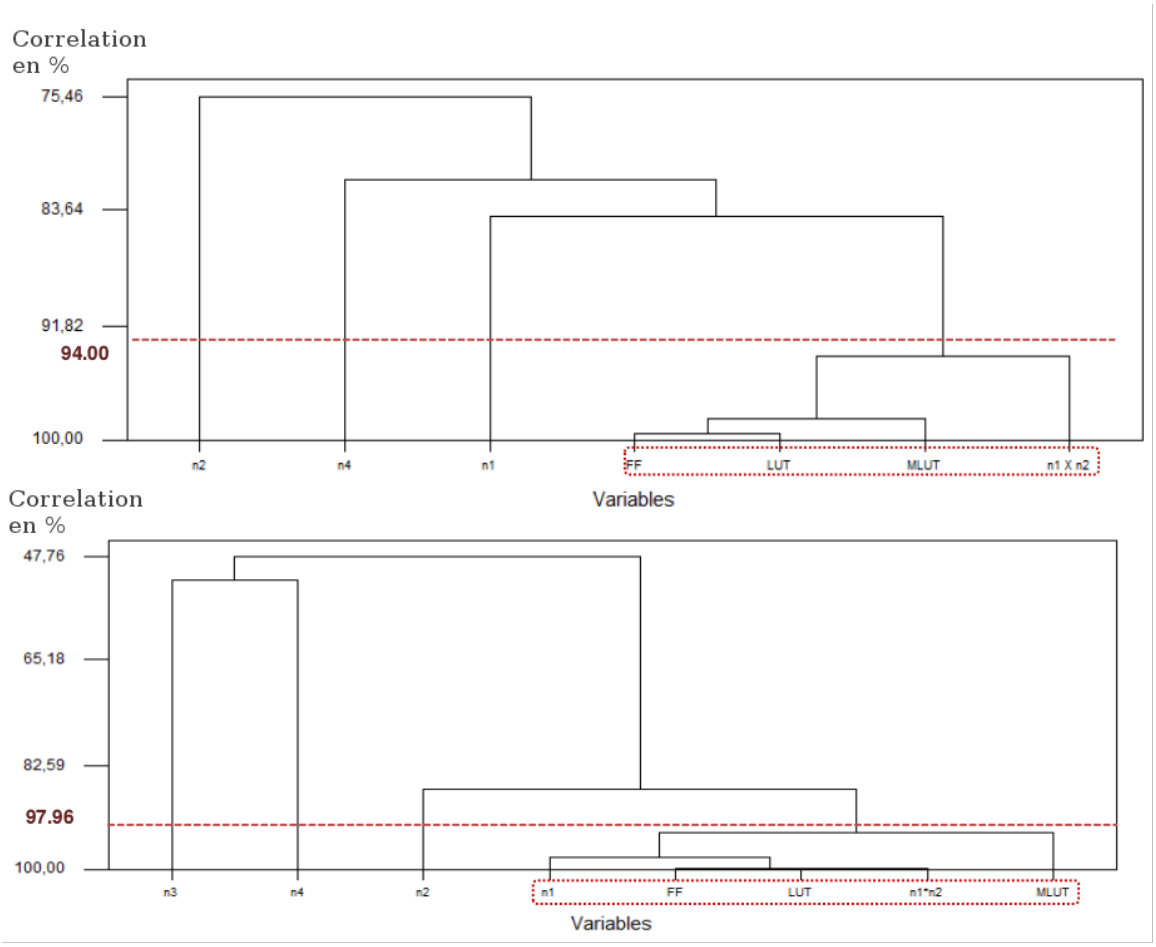


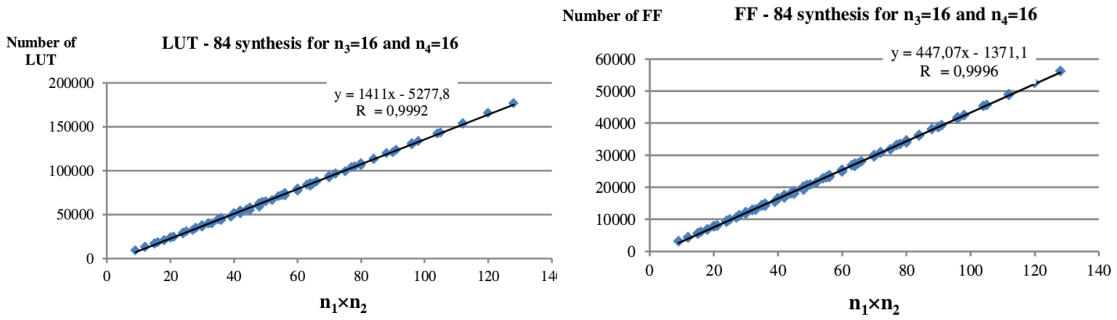
FIGURE 5.2 : Analyse hiérarchique des similarités entre variables pour Hermes (en haut) et AdOCNet (en bas)

Pour AdOCNet, un groupe contient cinq variables en prenant un seuil de similarité de 97,96% (n_1 , FF, LUT, MLUT, $n_1 \times n_2$). Pour Hermes, un groupe contient quatre variables en prenant un seuil de similarité de 94% (FF, LUT, MLUT, $n_1 \times n_2$). En conclusion, il y a un lien fortement linéaire entre les variables comme cela est résumé dans le tableau 5.4).

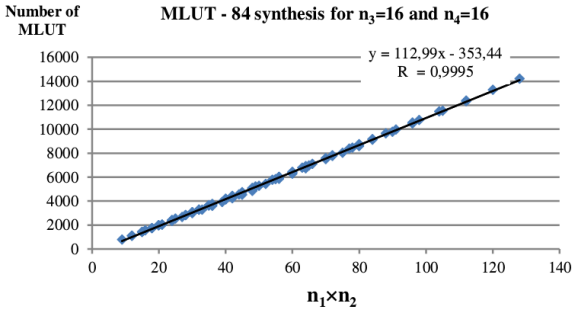
NoC	**Variables hautement corrélées**	**Niveau de similarité**
AdOcNet	FF, LUT, $n_1 \times n_2$, n_1, MLUT	97.96%
Hermes	FF, LUT, MLUT, $n_1 \times n_2$	94.00%

TABLE 5.4 : Identification des variables hautement corrélées pour chacun des NoC

Pour chaque NoC, les variables FF, LUT et MLUT sont fortement corrélées avec $n_1 \times n_2$. Notre objectif était d'identifier (et mesurer) les relations potentiellement linéaires entre les ressources et $n_1 \times n_2$. Les figures 5.3c, 5.3b et 5.3a valident les relations linéaires respectivement entre le nombre de MLUT, de FF et de LUT et $n_1 \times n_2$ pour Hermes. Nous avons ensuite vérifié cette linéarité pour chaque NoC.



(a) Nombre de LUT en fonction de $n_1 \times n_2$ (b) Nombre de flip flop en fonction de $n_1 \times n_2$



(c) Nombre de MLUT en fonction de $n_1 \times n_2$

FIGURE 5.3 : Résultats des synthèses pour le NoC Hermes.

5.1.3.2 Évaluation du modèle

L'analyse graphique est un moyen très efficace pour investiguer si un modèle de régression est approprié et vérifier les hypothèses sous-jacentes. La courbe des résidus nous permet d'examiner la pertinence du modèle.

Le test d'Henry (courbe de probabilité normale) et l'histogramme des valeurs résiduelles pour les FF sont présentées en figure 5.4 et 5.5. Les valeurs résiduelles pour les LUT et les MLUT sont similaires à celles des résidus pour les FF. La droite de Henry (figure 5.4) est une méthode graphique pour ajuster une distribution gaussienne avec celle d'une série d'observations.

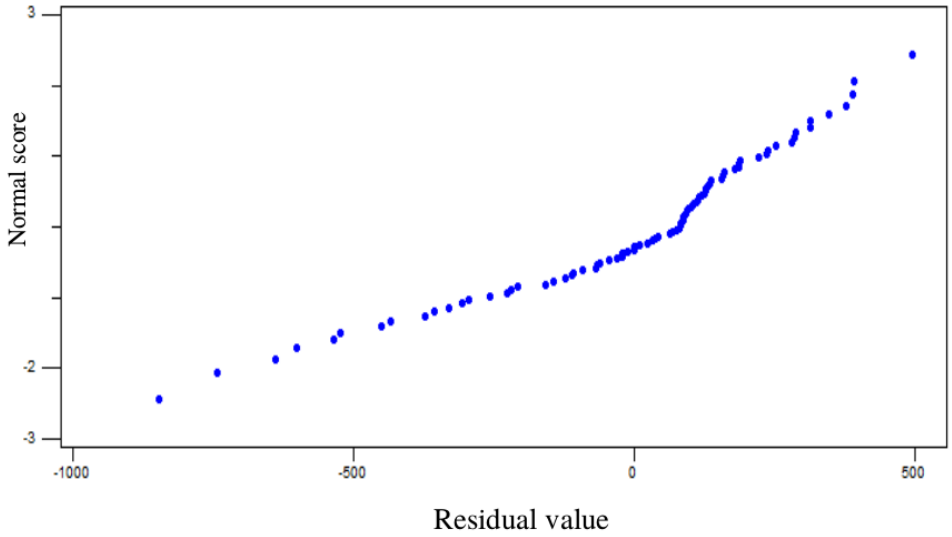


FIGURE 5.4 : Test de Henry

Nous observons que les résultats normaux ne sont pas alignés, il y a des écarts. L'histogramme des valeurs résiduelles (figure 5.5) utilisé pour vérifier les variances, n'est pas normalement distribuée autour de zéro (un histogramme symétrique bien formé et éventuellement centré autour de zéro indique que l'hypothèse de normalité est très probablement vraie).

Nous observons que le modèle de régression ne correspond pas parfaitement, mais les résultats sont tout de même satisfaisants et le modèle linéaire peut être approuvé (voir partie 5.1.3.4 pour la validation).

Par conséquent, le choix d'un modèle linéaire avec une variable $(n_1 \times n_2)$ semble être une bonne solution. Le concepteur peut maintenant utiliser ces résultats pour construire des modèles pour différentes configurations du NoC.

5.1.3.3 Modélisation : cas général

Le type de modèle linéaire est adopté. La construction d'un modèle peut donc être faite en ne réalisant que 3 synthèses logiques, c'est-à-dire 3 points pour établir le modèle : la droite. Pour une configuration donnée du NoC (topologie,

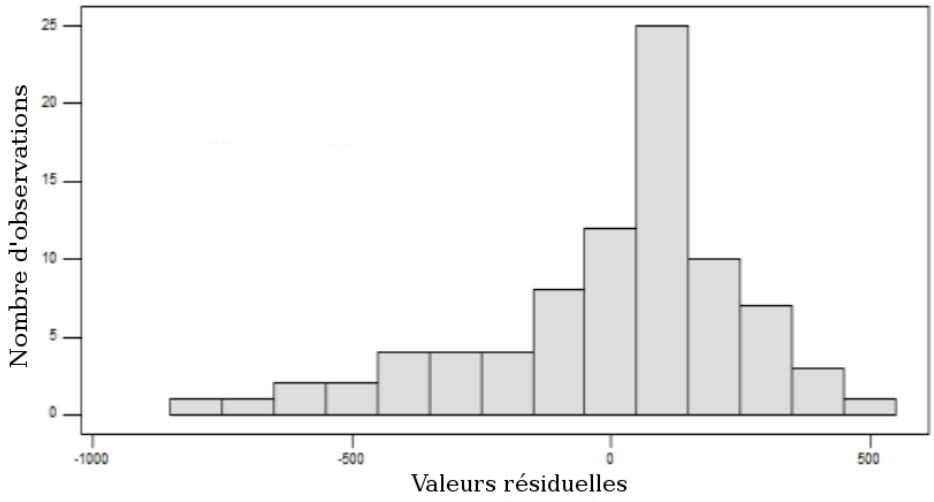
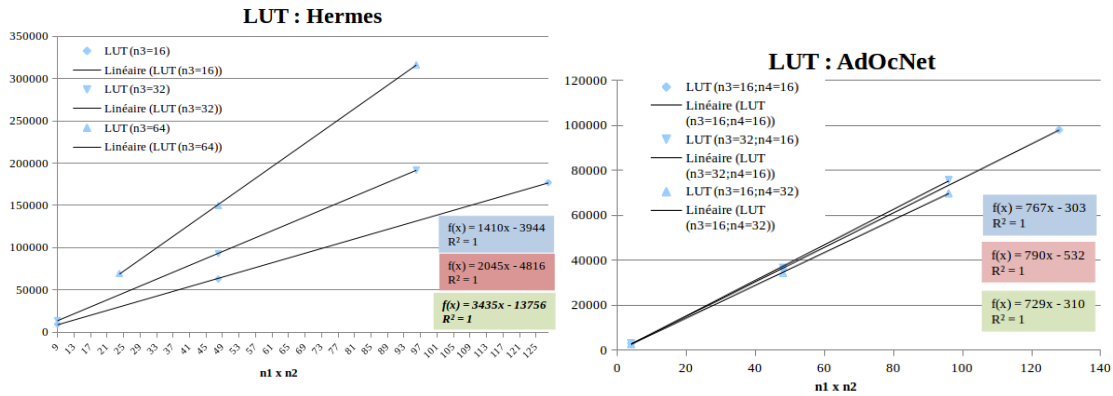


FIGURE 5.5 : Histogramme des valeurs résiduelles

la taille des mémoires tampons, la taille des connexions (flit), etc.), on obtient un modèle (une droite affine) pour chaque type de ressource.

Les figures 5.6a et 5.6b montrent respectivement les modèles de régression linéaire pour l'utilisation des LUT avec Hermes et AdOCNet.



(a) NoC Hermes (b) AdOCNet

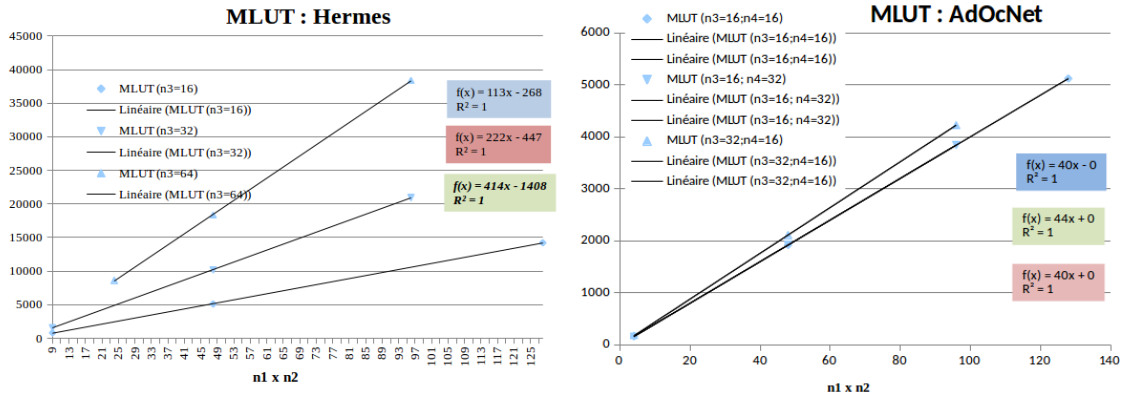
FIGURE 5.6 : Modèles de régression pour la consommation des LUT

Les figures 5.7a et 5.7b montrent le modèle de régression pour l'utilisation des MLUT avec Hermes et AdOCNet.

Les figures 5.8a et 5.8b montrent le modèle de régression pour l'utilisation des FF avec Hermes et AdOCNet.

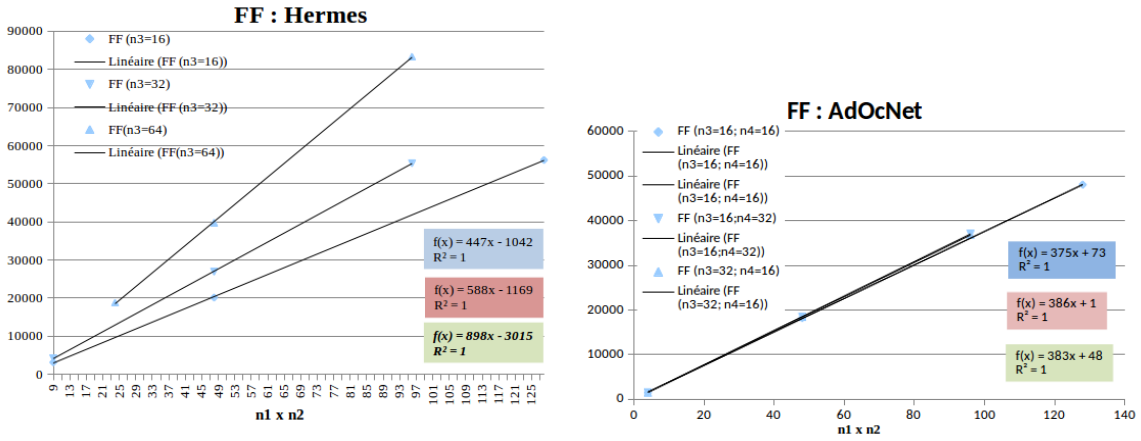
À partir d'un nombre de $n_1 \times n_2$ nœuds connectés au NoC, le concepteur peut maintenant estimer les ressources du FPGA consommées pour un jeu de paramètres du NoC modélisé.

Pour Hermes, la taille des flits a un impact significatif sur toutes les ressources.



(a) NoC Hermes (b) AdOCNet

FIGURE 5.7 : Modèles de régression pour la consommation des MLUT



(a) NoC Hermes (b) AdOCNet

FIGURE 5.8 : Modèles de régression pour la consommation des bascules (FF pour « flip flop »)

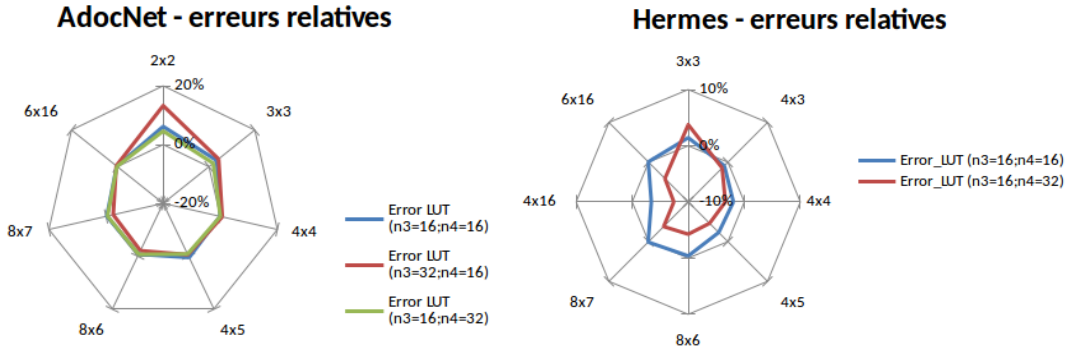
Il y a donc autant de modèles de régression linéaire pour chaque ressource que de taille de flit envisagé. Ces modèles sont significativement différents pour les LUT, les MLUT et les FF. AdOCNet a une structure bien plus linéaire, indifféremment de la taille des flits et de la profondeur des mémoires tampons.

Par conséquent, le nombre de synthèses logiques à réaliser est plus élevé pour Hermes que pour AdOCNet. Il est nécessaire d'extraire 3 points pour chaque taille de flits pour Hermes, contrairement à AdOCNet qui ne nécessite que de trois points (les trois modèles étant très proches).

5.1.3.4 Validation du modèle

Les erreurs relatives pour les LUT, MLUT et FF sont calculées entre les résultats après synthèse et les résultats estimés. Le taux d'erreur négatif indique que le nombre de ressources estimées est plus grand que le résultat de synthèse, et inversement pour un taux d'erreur positif.

Les erreurs relatives concernant les LUT pour Hermes et AdOCNet sont présentées en figure 5.9a et 5.9b. Le modèle surestime les ressources pour le premier cas et les sous-estime pour le second.



(a) AdOCNet (b) NoC Hermes

Figure 5.9 : Taux d'erreurs relatives

Pour Hermes, le taux d'erreur est de -6% à +2% pour $n_3 = 16$ et $n_4 = 16$. Pour AdOCNet, le taux d'erreur est de -8% à +4% pour $n_3 = 16$ et $n_4 = 32$.

Les erreurs minimales sont observées pour le NoC 4×16 (NoC Hermes) et les valeurs maximales sont pour le NoC 3×3 (NoC Hermes et AdOCNet). Les taux d'erreurs sont similaires (<10%) pour les MLUT et les FF pour Hermes et AdOCNet. Comme les erreurs relatives sont positives, le nombre de ressources estimées est en dessous du nombre de ressources utilisées du FPGA.

Le taux d'erreur diminue jusqu'à moins de 2.5% pour les plus grandes tailles de NoC. Ceci indique que les résultats de l'estimation analytique sont un peu plus grands que les résultats obtenus après synthèse pour des petites tailles de NoC. Si les tailles de NoC 2×2 et 3×3 ne sont pas prises en compte, les intervalles d'erreur sont $[-7.90\%; 6.90\%]$ pour Hermes et $[-2.52\%; 6.14\%]$ pour AdOCNet.

Les estimations pour de petites tailles de NoC (2×2 et 3×3) peuvent être remplacées directement par des valeurs obtenu après synthèse car le temps de synthèse n'est pas très élevé (quelques minutes). Mais l'utilisation d'un NoC dans ce cas n'est sans doute pas pertinent. Un bus partagé ou des connexions point à point semblent plus appropriées dans ce cas.

Par conséquent, les modèles mathématiques fiables peuvent être obtenus à partir de trois synthèses logiques pour un NoC de taille supérieure à 3×3 nœuds.

Notons certaines limites concernant la reproductibilité de la méthode. En effet, l'utilisation en ressources mesurée peut différer en fonction des versions du logiciel (*Vivado*) et des ordinateurs utilisés (deux machines de configurations différentes peuvent produire des résultats différents). Il est donc important que la synthèse finale du NoC et l'estimation des ressources soient faites avec la même configuration de logicielle (outils de synthèse logique) et matérielle (configuration de l'ordinateur).

5.2 Discussions sur les limites de la modélisation

Dans cette partie, trois points sont discutés :

— le taux d'erreur élevé pour des petites tailles de NoC,

— l'impact des options de synthèse sur le modèle généré,

— et l'impact des changements de la structure du NoC sur la linéarité du modèle.

5.2.1 Estimations pour les petites tailles de NoC

Les modèles sont utilisés pour accélérer le dimensionnement d'un NoC ($n_1 \times n_2$). Nous avons montré que les modèles pour les petits NoC fournissent des estimations biaisées du nombre de ressources FPGA utilisées (plus de 10% d'erreur relative).

Une solution est d'utiliser un modèle particulier pour ces points. Une autre est de ne pas utiliser de modèle mais de faire directement la synthèse durant la phase d'exploration de l'espace de conception.

La figure 5.10 présente le temps de synthèse en fonction des paramètres du NoC AdOCNet (variables $n_1 \times n_2$, n_3 et n_4). Ce temps de synthèse dépend du PC utilisé et principalement de $n_1 \times n_2$. Les petites tailles de NoC peuvent être synthétisées en moins de cinq minutes. Cela n'affecte pas la durée d'exploration de manière significative.

5.2.2 Impact des options de synthèse sur le modèle

Les modèles obtenus dépendent beaucoup des options de l'outil de synthèse utilisé. Les options choisies peuvent, par exemple, être liées aux spécificités d'un SoC comme l'utilisation de la mémoire.

Dans les expériences et analyses précédentes, les options par défaut sont choisies pour l'outil de synthèse logique XST (ISE) de Xilinx. Dans cette configuration, XST peut utiliser toutes les ressources du FPGA cible. Ces options peuvent être appropriées pour des systèmes hétérogènes utilisant des IP et des processeurs embarqués.

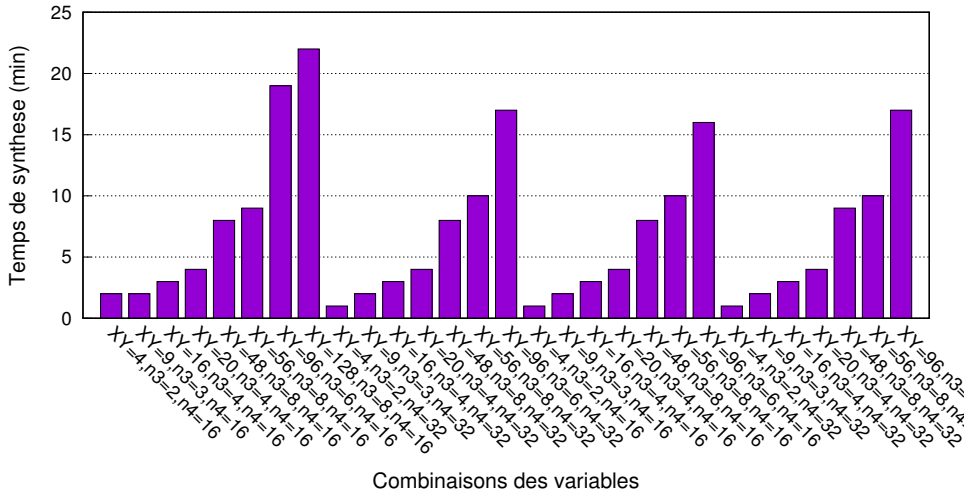


FIGURE 5.10 : Temps de synthèse de AdOCNet pour différentes combinaisons des variables $n_1 \times n_2$, n_3 et n_4.

Dans certains cas, certaines ressources peuvent être réservées pour des IP dédiés : LUT, FF pour les IP (utilisant aussi les blocs DSP) et MLUT pour les processeurs embarqués pour les mémoires d'instructions et de données.

Par conséquent, les modèles dépendent du type de système et des ressources FPGA. Une autre ressource majeure est l'utilisation de mémoire RAM appelées BRAM. L'impact des options de synthèse du l'utilisation des BRAM est discutée dans l'analyse suivante. Deux options de synthèse sont considérées en synthétisant AdOCNet.

Opt_BRAM_0 : XST ne peut pas utiliser les blocs de mémoire RAM (BRAM) lorsqu'elles sont déjà utilisées par les processeurs embarqués. Cette configuration doit être préférée pour les systèmes de type MPSoC.

Opt_BRAM_1 : XST peut utiliser autant de mémoire RAM que possible pour la structure de communication (le NoC).

Les modèles linéaires pour les LUT et les FF et les MLUT/BRAM pour les deux options de synthèse pour AdOCNet sont présentés ci-dessous :

Opt_BRAM_0 : $N_{LUT} = 390 \times x$, $N_{FF} = 720.6 \times x - 83.4$, $N_{MLUT} = 44 \times x$

Opt_BRAM_1 : $N_{LUT} = 330 \times x$, $N_{FF} = 687 \times x - 32$, $N_{MLUT} = 3 \times x$

Avec x = $n_1 \times n_2$ pour tous les modèles.

5.2.3 Impact de la structure du NoC sur les modèles

Ajouter ou remplacer les composants dans la structure d'un NoC influence la linéarité du modèle. Le modèle change mais reste linéaire pour les cas suivants :

— changement d'un algorithme de routage,

— remplacement du type de flot de contrôle (basé sur les crédits, poignée de main, etc.),

— ajout de canaux virtuels,

— spécification d'autres options de synthèse.

D'autres changements dans la structure peuvent supprimer la linéarité du modèle. En effet, l'ajout de mécanismes particuliers d'ordonnancement des communications comme dans [DFBM14], en particulier l'ordonnancement de type « weighted round robin » avec le NoC Hermes.

L'estimation des ressources du circuit final dépend du NoC mais aussi des différents blocs spécifiques à l'architecture flot de données utilisée. Les choix architecturaux sont présentés dans le chapitre suivant.

Chapitre 6 : Proposition d'une architecture distribuée programmable

It's hardware that makes a machine fast. It's software that makes a fast machine slow.
Craig Bruce

Sommaire

Dans ce chapitre est définie l'architecture du circuit généré par la synthèse. Pour répondre au besoin de « scalabilité » et pour obtenir un comportement de type « data-driven », proche du modèle de calcul, nous avons défini une architecture distribuée de type flot de données basée sur un réseau. La partie 6.1 présente les choix d'implémentation du modèle de calcul sous forme de circuit basé sur un réseau. La partie 6.2 présente comment est structuré un terminal de calcul et notamment comment sont distribuées les données pour réaliser des tâches différentes sur les mêmes IP. Les parties 6.3 et 6.4 traitent plus particulièrement de la gestion de la granularité des données et des branchements conditionnels, respectivement. Enfin, la partie 6.5 présente les mécanismes de distribution des *configurations* et des *données* à partir d'un terminal dédié à la communication avec l'ordinateur hôte et comment sont stockées les informations nécessaires à la reprogrammation de l'architecture proposée.

6.1 Choix architecturaux

6.1.1 Objectifs et contraintes

L'objectif de l'architecture devant supporter l'exécution de l'algorithme analysé, est de permettre l'exécution des *tâches identifiées* sur les *opérateurs sélectionnés* tout en offrant des mécanismes pour le parallélisme de tâches, de données et de flux.

De plus, dans le cadre de l'informatique reconfigurable, il est important d'éviter, si possible, les synthèses longues et répétitives à chaque modification de l'algorithme (comme le nombre d'itérations d'une boucle, par exemple).

Cependant, la réalisation d'une architecture basée sur une communication en réseau pour une cible FPGA impose, de manière générale, les contraintes suivantes :

Une insensibilité à la latence : la latence mesurée pour la traversée d'un NoC par un paquet peut varier en fonction de la congestion dans le réseau et des distances réseau (nombre de routeurs traversés) entre deux IP. L'exécution d'une tâche ne peux donc pas être faite à un cycle prédéfini mais sur la réception de la totalité de ses entrées.

Une « scalabilité » : un NoC est essentiellement utilisé pour cette propriété. Les fréquences de fonctionnement du circuit généré doivent être indépendantes de sa taille. Le contrôle du séquencement des opérations ne doit donc pas créer de goulot d'étranglement afin de respecter cette contrainte.

Une utilisation d'accélérateurs dédiés : étant données les fréquences de fonctionnement des FPGA, l'utilisation des processeurs dits « soft core » dans les FPGA n'est pas pertinent pour l'accélération de calcul. En effet, ces derniers ont des fréquences de fonctionnement inférieures d'un ordre de grandeur à celles des processeurs « hard core », c'est-à-dire aux ASIC

(« application specific integrated circuit ») et ne peuvent donc rivaliser en termes de performances (débits et latences).

Le respect du modèle flot de données : l'architecture doit respecter au mieux le modèle de calcul comme la granularité et le type des données, les acteurs spécialisés, et l'exécution lors de la disponibilité des opérandes.

6.1.2 Terminaux flots de données

Dans notre modèle flot de données, les jetons peuvent être stockés sur les arcs, en mode FIFO (« first-in first-out »). Dans un réseau utilisant des mémoires tampon en entrée des routeurs (comme celui que nous utilisons), les liens sont partagés, tout comme ces mémoires dont l'utilisation dépend de l'état du réseau (congestion). Si ces mémoires sont pleines, l'exécution des tâches en aval doit s'arrêter pour éviter d'écraser ou de perdre les données produites.

De plus, le temps de transit des données le long des arcs est considéré nul dans le modèle alors qu'il est variable dans le circuit final. Une synchronisation est donc nécessaire d'autant que les données arrivent les unes après les autres dans un ordre qui peut changer. En effet, étant donné l'algorithme de routage utilisé, un destinataire reçoit les données dans le même ordre que celui dans lequel elles ont été envoyées (l'ordre des flux est respecté). Mais si plusieurs sources sont considérées, l'ordre dans lequel les données sont reçues peut différer de celui dans lequel elles ont été émises (l'ordre d'arrivée des entrées pour un même flux n'est pas garanti).

Les architectures de type flots de données utilisent en général un tableau d'affichage (« scoreboard » ou architecture de Tomasulo) pour déclencher un calcul sur la disponibilité des données.

Nous choisissons aussi d'utiliser une table de disponibilité des données. Cependant, nous faisons le choix de distribuer ce mécanisme de contrôle au niveau de chaque terminal. Un terminal doit donc disposer des registres tampon des données reçues pour tolérer ces variations de latence dues au réseau et permettre la synchronisation (exécution sur disponibilité des données).

Dans notre modèle de calcul, cinq types de nœuds coexistent : les nœuds de calcul, les nœuds de type *select* et *switch* pour les branchements et enfin les nœuds de types « map » et « reduce » pour la manipulation des tableaux. L'architecture de type flot de données proposée dispose donc aussi des types de terminaux correspondants, un pour chaque type de nœud. Un terminal de communication est ajouté pour communiquer avec le CPU hôte.

6.1.3 Caractéristiques générales d'un terminal

Les architectures générées par les outils de synthèse HLS actuels sont centralisés, avec une partie contrôle (automate d'état) et une partie opérative avec les chemins de données. Les connexions sont alors de type points à points.

Si les opérateurs sont connectés par un NoC, utiliser un automate d'état peut engendrer des problèmes de synchronisation, de goulot d'étranglement et par conséquent de performances.

Pour y remédier, nous proposons une architecture de type flot de données capable d'exécuter le modèle produit lors de la phase d'analyse. Il stocke les données localement, au sein des registres de chaque terminal, jusqu'à leur consommation.

Un terminal est le circuit connecté à un port local d'un des routeurs du NoC (l'ensemble routeur et terminal forment un nœud du réseau). Le terminal développé pour notre architecture :

— formate les données sous forme de paquets pour communiquer via le NoC,
— mémorise les données d'entrée jusqu'à leur consommation,
— identifie la tâche qui doit être exécutée sur l'opérateur,
— suspend son activité si le réseau est surchargé (congestion),
— prend en compte le « pipeline » interne de l'opérateur.

6.1.4 Résumé des choix d'implémentation

L'architecture proposée présente 6 types de terminaux :

des terminaux de calcul : gèrent la synchronisation, la mémorisation et le partage des tâches sur une IP.

des terminaux de type « switch » : permettent la commutation des paquets vers la branche sélectionnée après une suite de tests booléens (correspondants aux *if* et *elif*)). C'est, en quelque sorte, un démultiplexeur de paquets réseau.

des terminaux de type « select » : permettent de sélectionner les données provenant d'une des branches en amont et de les router vers une destination unique, pour une tâche de sélection donnée. C'est, en quelque sorte, un multiplexeur de paquets réseau.

des terminaux de type « map » : permettent de découper un paquet en plusieurs paquets de tailles plus petites.

des terminaux de type « reduce » : permettent de recomposer un paquet à partir de plusieurs paquets.

un terminal de communication : permet les échanges de données entre le CPU hôte et le NoC embarqué sur FPGA.

L'ensemble de ces terminaux dispose de mémoires tampon pour synchroniser les données d'entrée de chaque tâche correspondante. Le contrôle de l'exécution des tâches sur un terminal est fait localement de manière distribuée et asynchrone.

Avant l'exécution des tâches, un terminal doit être configuré (programmation des chemins de données). Une configuration permet de sauvegarder les identifiants des données produites pour chaque tâche et leurs destinations dans le

réseau. Au sein d'un terminal, un mécanisme pour le multicast a été développé pour permettre l'envoi d'une donnée à plusieurs destinataires.

Dans le réseau, les données sont échangées avec leurs méta-données (voir partie 6.2.1.2) et peuvent être des scalaires ou des vecteurs.

Dans ces travaux de thèse, l'ensemble des terminaux et mécanismes présentés a été développé. Nous allons maintenant présenter le fonctionnement des terminaux de calcul.

6.2 Terminaux de calcul

6.2.1 Partage des tâches

La figure 6.1 présente la micro-architecture d'un terminal de calcul.

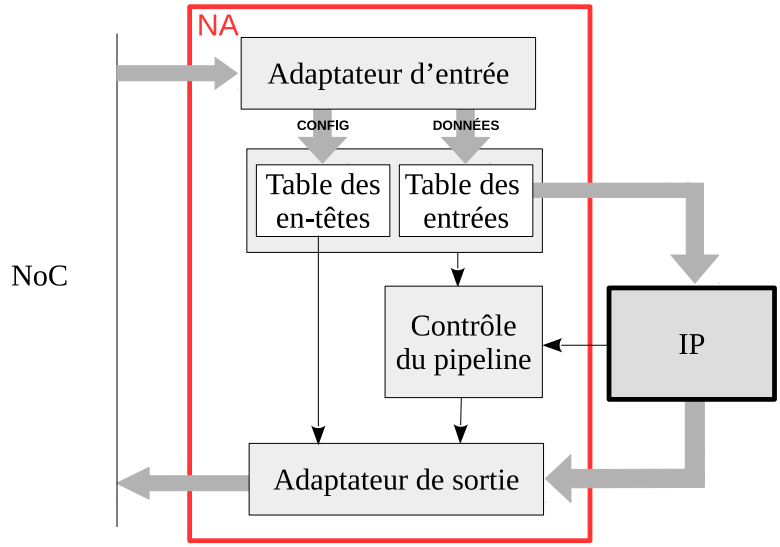


FIGURE 6.1 : Micro-architecture d'un terminal dédié au calcul

La génération et la réception des paquets sont réalisées au niveau des adaptateurs réseau. Les données d'entrées sont stockées dans une mémoire (table des entrées). Les en-têtes des paquets envoyés sont préalablement mémorisés dans une mémoire (table des en-têtes). Le gestionnaire du « pipeline » interne est essentiellement un registre à décalage permettant de déterminer la validité des données de sortie de l'opérateur et de respecter l'intervalle d'introduction des données (DII pour « data introduction interval »).

6.2.1.1 Identification des données

Une tâche est identifiée par l'*opération* réalisée et les *données* qu'elle va consommer. Un opérateur, lui, est identifié par son adresse sur le réseau.

Dans le modèle flot de données, les données sont représentées sous forme de *jetons*. Dans les architectures de type flot de données, un jeton rassemble des

données (ici des scalaires ou des vecteurs) et des meta-informations pour leur identification.

Dans notre architecture, les données sont échangées sous forme de paquets. Un paquet contient, bien-sûr, des données, mais aussi l'adresse de destination du paquet et le nombre de données contenues dans ce paquet. D'autres informations doivent être ajoutées pour permettre d'identifier une tâche cible et un paquet contenant des données pour la configuration ou pour le calcul.

6.2.1.2 Structure des en-têtes

Une tâche associe un opérateur aux données qu'il consomme. Dans un paquet réseau, l'en-tête est le groupe des premiers flits rassemblant des méta-informations (ou méta-données) sur les données transportées. Dans le NoC utilisé, l'en-tête est constitué de deux flits : l'un pour l'adresse de destination du paquet et l'identification des données, l'autre pour la charge du paquet (nombre de flits de données). L'adresse de destination est contenue dans la première moitié du premier flit. La deuxième moitié est structurée de telle manière à identifier les données et les tâches qui doivent les consommer. Dans la suite de ce chapitre, nous utiliserons le terme en-tête pour identifier ce premier flit « identificateur » des données.

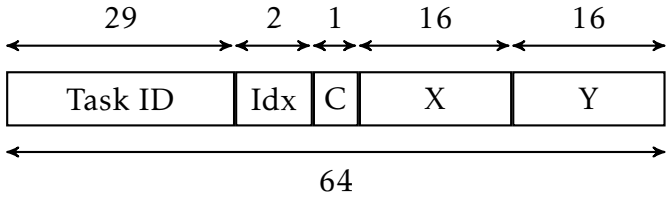


FIGURE 6.2 : Structure d'un en-tête standard (unité : nombre de bits).

La figure 6.2 présente un exemple de premier flit d'un en-tête de paquet à destination d'un terminal de calcul avec une taille de flit de 64 bits et une IP à 4 entrées. La taille du flit et le nombre d'entrées d'une IP sont déterminés à la synthèse en fonction des IP sélectionnés. Les champs X et Y contiennent respectivement les coordonnées en x et en y du nœud destinataire sur le réseau maillé 2D.

Le champ *C* identifie s'il s'agit d'un paquet de configuration.

Le champ *Idx* identifie l'index de la donnée en entrée de l'IP. Par exemple, une soustraction a deux entrées 1 et 2. Le champ « Idx » contient donc soit 0 soit 1 selon que la donnée doit être présentée sur la première ou deuxième entrée du soustracteur. La largeur de ce champ est fixée pour une IP donnée (codage en base 2 de l'index).

Enfin, dans notre architecture, chaque terminal peut exécuter des tâches différentes. Il y a donc toujours un champ pour identifier la tâche qui va consommer les données du paquet, nommée « TaskID ». La largeur de ce champ est variable en fonction de la taille du flit.

Trois types de paquets peuvent circuler dans le réseau : des paquets de configuration, de données et de contrôle.

6.2.2 Configuration des terminaux de calcul

Dans l'architecture proposée, la configuration des terminaux est réalisée à l'initialisation de l'application, préalablement à tout calcul. Comme la durée de la configuration d'une application n'est pas critique, nous avons choisi d'utiliser le troisième *flit* des paquets de configuration pour coder leur type. La figure 6.3 présente la structure d'un paquet de configuration.

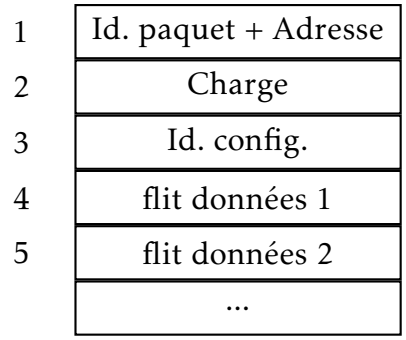


FIGURE 6.3 : Structure d'un paquet de configuration.

Les configurations possibles sont les suivantes :

— remplissage de la tables des en-têtes de chaque terminal et

— initialisation des constantes.

En effet, les tâches peuvent consommer des constantes ou des variables. Pour éviter une surcharge du réseau avec des envois redondants, notre choix a été de déterminer les valeurs des constantes lors de la phase de configuration des terminaux. Les constantes sont, bien-sûr, identifiées pour chaque tâche.

6.2.3 Granularité des données

La granularité d'une donnée peut varier selon le type des objets manipulés dans Python, puis identifiés comme jetons dans le modèle. Un paquet de type *calcul* correspond à un jeton et contient toujours une donnée.

Ces données peuvent être des scalaires (un flit) ou bien des vecteurs (plusieurs flits).

Dans le cas des IP consommant des vecteurs, une mémoire RAM tampon est instanciée pour chaque vecteur consommé et est utilisée pour l'ensemble des tâches. La taille des vecteurs détermine donc la quantité de mémoire utilisée. Notre outil de synthèse ne supporte pas encore l'utilisation de mémoires externes.

6.2.4 Mémoires tampon

La table des entrées permet de stocker les données pour des tâches différentes en attendant que l'exécution d'une tâche puisse être déclenchée. Une tâche peut être exécutée dès que l'ensemble de ses données d'entrée est disponible (présent dans la mémoire tampon).

Le tableau 6.1 montre un exemple de table d'entrée. Chaque ligne correspond à un identifiant de tâche (« TaskID ») et chaque colonne à l'index de la donnée (« DataIdx »).

TABLE 6.1 : Un exemple de table des entrées pour une IP à trois entrées.

TaskID	Index 1	Index 2	Index 3
0	0xFFF10182	0xA1510212	0x00000000
1	0x00000023	0x00055555	0x00000000
2	0x00D100E1	0x00000000	0x00000000
...			

Une mémoire supplémentaire est ajoutée pour chaque entrée de type *vecteur* et remplace la colonne associée à cette entrée.

Le statut des entrées pour chaque tâche est sauvegardé dans un registre dédié. La valeur de ce registre est testée à chaque réception de paquets et permet d'exécuter ou non la tâche. Ce registre est ensuite ré-initialisé.

6.2.5 Valeurs constantes et passage par défaut

Une fonction Python peut avoir des arguments avec des valeurs par défaut ou constantes. Si aucune variable n'est associée à un paramètre par défaut ou bien si c'est une constante qui lui est associée, la tâche est exécutée avec une valeur constante déterminée à la configuration.

Puisque chaque IP peut exécuter plusieurs tâches, une valeur constante est associée à une seule tâche. Nous avons choisi de sauvegarder les constantes au niveau de chaque terminal au moment de la configuration de l'application.

Cette manière de traiter les constantes peut aussi être utilisée pour les IP qui ont des registres de configuration. Ces derniers sont alors traités comme des constantes.

Lors des configurations des constantes, le statut de ces dernières, pour la tâche concernée, est mis à jour (l'entrée concernée est marquée *constante* pour la tâche). Ce statut est ensuite utilisé en tant que masque avec le registre de statut des entrées (disponibilité des variables) pour déclencher ou pas l'exécution de la tâche.

6.3 Traitement des vecteurs

Certaines opérations sur les données nécessitent un traitement particulier. D'une part, la composition d'un vecteur à partir de plusieurs données est traduite sur le réseau par la formation d'un unique paquet de grande taille à partir de paquets plus petits. C'est, par exemple, la composition d'une image à partir de ses sous régions.

D'autre part, la composition d'un vecteur à partir de données scalaires nécessite la génération d'un paquet à partir de plusieurs paquets (groupement des données). C'est, en quelque sorte, l'opération inverse.

La figure 6.4 présente les deux mécanismes à implémenter.

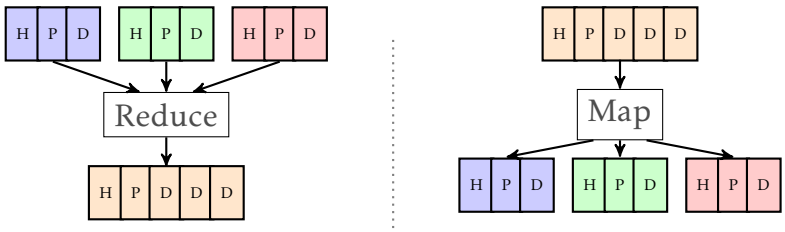


Figure 6.4 : Changement de granularité dans l'architecture proposée : les terminaux « reduce » et « map ».

Ces deux opérations sont reprises dans le modèle de programmation popularisé par Google [DG10] pour la manipulation de tailles importantes de données au sein d'architectures distribuées. De la même manière, les tâches et leurs terminaux correspondants sont nommés « map » et « reduce » pour respectivement décomposer et recomposer les paquets.

6.3.1 Terminaux de type « map »

Pour réaliser la décomposition de paquets, des terminaux spécialisés sont implémentés. Tout comme les terminaux de calcul, ils peuvent exécuter plusieurs tâches de décomposition en cas de manque de ressources.

Ces terminaux permettent de décomposer un vecteur en sélectionnant les tranches du paquet à « découper ». Une tranche est un ensemble de flits qui peut contenir au minimum un flit. Ces tranches sont programmées dans une mémoire et associées à des en-têtes.

La configuration des tranches est faite en stockant, pour une tâche de type « map », la liste de tailles de toutes les tranches. Puis chaque tranche est associée à un en-tête ou bien est ignorée.

En suivant cette stratégie, seuls les sous-ensembles contigus de flits peuvent composer un nouveau paquet à partir du paquet parent. Pour changer l'ordre des flits dans un paquet, il faut une paire de « map »/« reduce ».

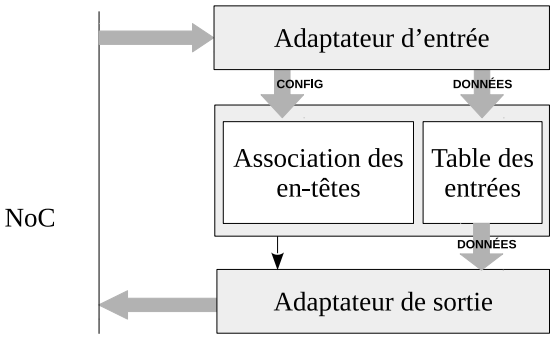


FIGURE 6.5 : Structure d'un terminal de type « map » (décomposition d'un paquet en paquets de granularités inférieures)

6.3.2 Terminaux de type « reduce »

Pour réaliser la re-composition de paquets, nous utilisons aussi des terminaux spécialisés pouvant exécuter plusieurs tâches de re-composition. Le fonctionnement de ce terminal est plus simple : les éléments d'un vecteur sont mémorisés jusqu'à disponibilité de tous les éléments puis un nouveau paquet regroupant ces données est construit.

La figure 6.6 présente l'architecture d'un terminal de composition de paquets. L'adaptateur d'entrée permet de remplir des mémoires tampon, pour chaque

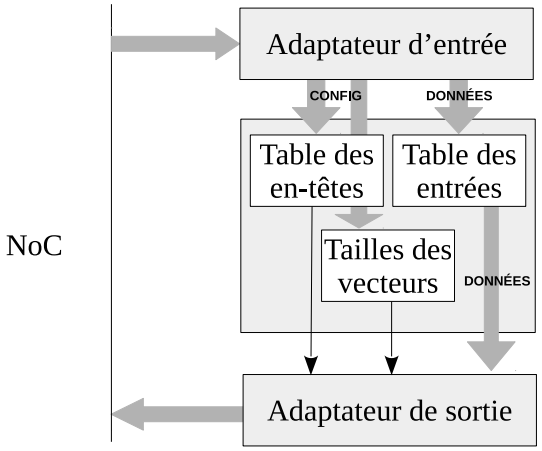


FIGURE 6.6 : Structure d'un terminal de type « reduce » (composition d'un paquet à partir de plusieurs autres)

tâche de composition, pouvant stocker les données scalaires ou vectorielles à encapsuler dans un paquet. Le nombre de données à encapsuler pour chaque tâche est configurable individuellement grâce à une mémoire de configuration (tailles des vecteurs). L'adaptateur de sortie construit les paquets à partir des données stockées avec la charge (nombre de données scalaires) précédemment configurée.

6.4 Terminaux de contrôle

Dans le modèle de calcul, deux types de tâches sont nécessaires aux branchements : le « switch » et le « select ». Dans notre architecture, les branchements conditionnels sont contrôlés par des terminaux spécialisés correspondants pour le routage conditionnel de paquets. Les tests booléens correspondants aux instructions *if/elif/else* sont effectués dans le terminal de type *switch*.

6.4.1 Tests booléens et acteurs « switch »

Un terminal de type « switch » peut recevoir deux types de paquets pendant le fonctionnement de l'application : un pour le contrôle (données pour le test booléen) et un pour les données à router.

La figure 6.7 présente la structure d'un terminal de type « switch ». Comme

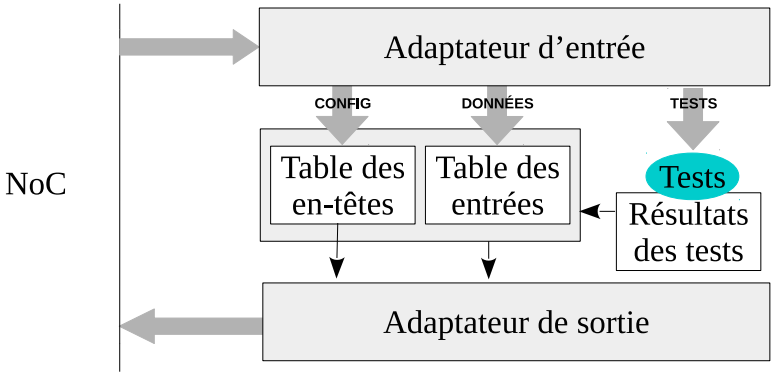


Figure 6.7 : Structure d'un terminal de type « switch »

pour le terminal de calcul, la table des entrées permet la mémorisation des données en attente d'être routées. Plusieurs données peuvent être en attente des résultats d'un test de branchement. Pour un branchement donné, la table des en-têtes associe une donnée à router à une adresse de destination.

À un branchement correspond un *groupe de tests*, un ensemble de *if*, *elif* et/ou *else*. Il y a autant de valeurs à tester que de *if* et *elif*. Les tests sont réalisés à la réception d'un paquet de contrôle et un registre d'état est mis à jour. Ce registre dispose d'un bit par test pour un *groupe de tests* donné.

Un test positif entraîne la prise d'une branche et la remise à zéro du registre d'état du groupe de tests. Lors de la configuration des terminaux de type « switch », un registre de référence par groupe de test est positionné. Il définit le nombre de tests au bout duquel la branche prise est celle du « else » si les tests sont tous négatifs.

6.4.2 Acteurs « select »

Un terminal de type « select » est plus simple et peut recevoir aussi deux types de paquets : un pour le contrôle (résultat du test booléen) venant du « switch »

correspondant et un pour les données provenant de la branche à sélectionner.

La figure 6.8 présente la structure d'un terminal de type « select ». Dans ce

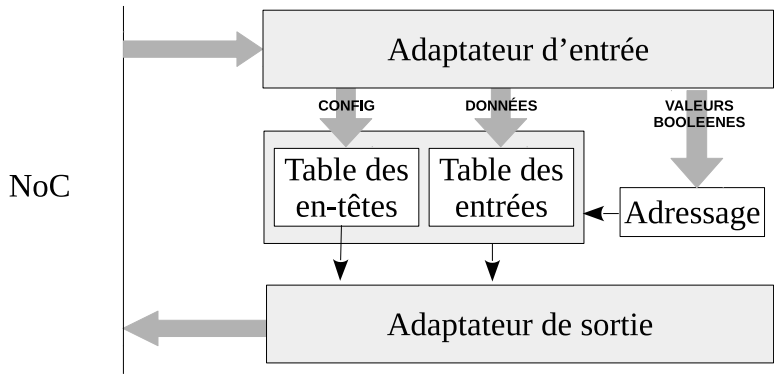


FIGURE 6.8 : Structure d'un terminal de type « select »

terminal de sélection, pour une tâche de sélection donnée, les données peuvent provenir de deux sources différentes. La table des entrées permet leur mémorisation en attendant le résultat du test provenant de la tâche de type « switch » correspondante (paquet de contrôle).

L'attente des paquets de contrôle est indispensable pour éviter les dépassements de flux. En effet, dans un contexte de fonctionnement « pipeliné », deux données successives peuvent prendre des branches différentes, chacune avec une latence propre. Il est donc possible que la donnée du deuxième flux arrive avant celle du premier. C'est la valeur booléenne contenue dans le paquet de contrôle qui définit quelle voie est sélectionnée.

Dans ce terminal, la table des en-têtes associe une adresse de destination à une donnée pour chaque tâche de sélection.

6.5 Programmation de l'architecture

Les différents terminaux présentés précédemment reposent sur une table des en-têtes pour l'envoi des paquets. Dans cette partie nous montrons comment il est possible de changer les chemins de données (destinations des paquets envoyés) en remplissant la mémoire associée à cette table.

6.5.1 Chemins de données

Dans l'architecture proposée, les dépendances de données sont programmées en associant à chaque donnée produite par un opérateur (terminal de calcul), une liste des tâches qui les consomment.

L'en-tête d'un paquet contient toute l'information de consommation pour la donnée qu'il contient. La liste des en-têtes pour toutes les données de l'application suffit à définir toutes les dépendances de données et donc l'application elle

même. Cette liste d'en-têtes est chargée pendant une phase de programmation au niveau de chaque terminal.

Dans cette architecture, le terme *programme* signifie la liste des paquets de configuration permettant de charger ces en-têtes, les constantes, les motifs pour les terminaux de type « map » et « reduce » au niveau de chaque nœud du réseau.

Étant donné le routage déterministe du NoC utilisé, ces en-têtes définissent les *chemins de données* pour une application dans le réseau. Les en-têtes sont chargés dans des mémoires prévues à cet effet : les tables des en-têtes. La figure 6.2 présente un exemple de cette table.

Table 6.2 : Un exemple de remplissage d'une table des en-têtes.

TaskID	multi-cast	En-tête
0	2	0x00020102
1	0	0x00040103
2	0	0x00080001
3	1	0x00020101
...		

Cette table présente deux types d'associations pour un identifiant de tâche : le nombre de paquets à envoyer par donnée produite et la liste de ces en-têtes (sur 32 bits). Par exemple, pour l'identifiant 0 (TaskID), il y a trois destinataires (multi-cast vaut 2) : l'un à l'adresse (1,2), un autre à l'adresse (1,3) à la ligne 1, et le dernier à l'adresse (0,1), à la ligne 2.

Les mémoires dans lesquelles sont stockés les en-têtes pour l'ensemble des tâches exécutées sont remplies durant la *phase de programmation* de l'architecture. À ce moment là, les mémoires de chaque terminal sont remplies les unes après les autres. Les en-têtes sont définis après le placement et l'ordonnancement final du graphe des tâches sur les IP.

6.5.2 Destinataires multiples (multi-cast)

Pour permettre les envois multiples, le nombre de destinataires par tâche est stocké dans la table des en-têtes.

La procédure suivie à la réception d'une donnée d'entrée est décrite par la figure 6.9. L'identifiant de tâche du dernier paquet reçu est utilisé comme adresse. Celui-ci permet de sélectionner un en-tête et un nombre de destinataires. Un registre de statut est mis à jour jusqu'à ce que toutes les entrées d'une tâche soient disponibles. À ce moment là, le calcul est lancé, le registre de statut est remis à zéro et les données produites sont temporairement sauvegardées pour être envoyées aux bons destinataires. Enfin, la donnée produite est encapsulée dans un paquet pour être envoyée.

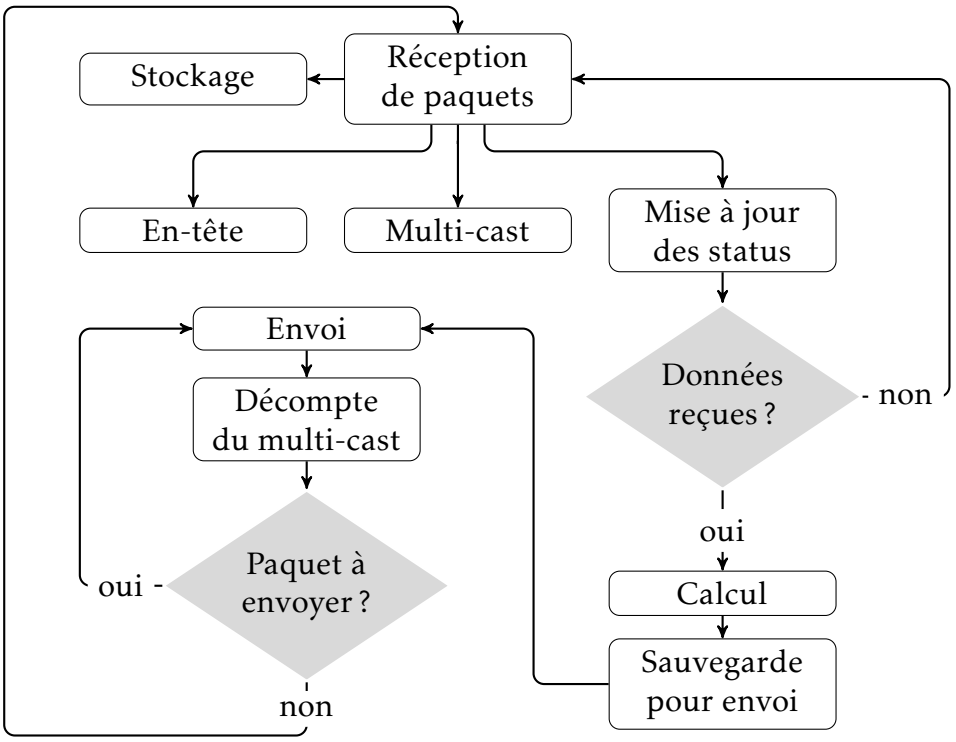


FIGURE 6.9 : Procédure d'envoi des données produites à plusieurs destinataires.

Un compteur est positionné pour compter le nombre de destinataires. À la fin du calcul, un paquet est envoyé pour chaque destinataire. L'adresse de l'en-tête utilisé pour chacun d'entre eux correspond à la somme d'une base (identifiant de tâche) et d'un offset (le compteur de destinataires).

Le contrôle de la production des paquets est donc distribué (géré localement), programmable et indépendant des latences de communication.

6.5.3 Pipeline et dépassements de flux

L'intérêt d'une telle architecture devient manifeste dès lors que l'on utilise ses capacités de mise en « pipeline ». Toutefois, étant données les latences imprévisibles, des dépassements de flux peuvent se produire. C'est pouquoi nous avons conçu un mécanisme de réordonnancement qui résout ce problème.

En effet, alors que les paquets de même provenance ont ordre d'arrivée garanti pour un même destinataire, ceux de provenances différentes ont un ordre d'arrivée qui dépend du réseau (distance, trafic, etc.). La figure 6.10 illustre ce problème. Alors que le nœud C s'attend à recevoir P_A0 et P_B0, il peut recevoir P_A0 et P_A1 comme présenté dans cette figure. Il est alors nécessaire de faire « attendre » P_A1 et recevoir d'abord P_B0.

Lorsqu'il n'y a pas de place en mémoire pour les données de deux flux différents mais partageant le même identifiant de tâche et le même index, si la consommation des données dépend d'autres données encore dans le réseau, il y

FIGURE 6.10 : Illustration du problème de dépassement de flux.

a blocage. En effet, les liens physiques du NoC sont utilisés pour des données d'index différents.

Pour cette raison, un mécanisme de renvoi des paquets au niveau des adaptateurs d'entrée a été développé. Le principe est simple : lorsqu'un terminal n'a plus de place pour un paquet (arrivé trop tôt), l'adaptateur d'entrée sauvegarde le paquet dans une mémoire FIFO dédiée. Le prochain paquet reçu peut ensuite être lu. À chaque réception d'un paquet provenant du routeur, l'ensemble des paquets présents en mémoire est à nouveau présentée au terminal. La solution proposée est schématisée (chemins de données) par la figure 6.11.

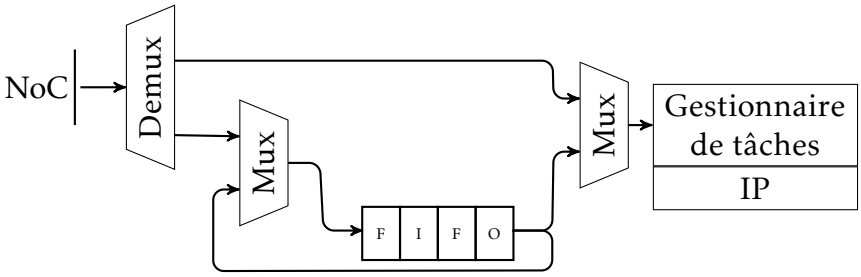


FIGURE 6.11 : Chemins de données de la solution proposée pour le réordonnancement de flux.

Il est important de noter que les paquets doivent être reçus dans l'ordre dans lequel ils ont été envoyés. C'est pouquoi nous choisissons un algorithme de routage déterministe dans le NoC utilisé. L'utilisation d'un NoC au comportement adaptatif ou tolérant aux fautes [KTMD14], des mécanismes garantissant une arrivée ordonnée doivent être implémentés.

Cette partie conclut ce chapitre et la présentation de la méthode de synthèse proposée. Dans le prochain chapitre, la méthode de synthèse est appliquée à deux applications concrètes et montre que notre approche peut être intéressante dans le cadre de l'informatique reconfigurable.

Chapitre 7 : Expérimentation et étude des performances

Expérience est mère de science.

Proverbe

Sommaire

Dans ce chapitre, les synthèses de deux algorithmes sont étudiées : l'un dans le domaine de la finance, l'autre pour le traitement d'images. Ces algorithmes sont de natures très différentes et permettent de mettre en évidence des caractéristiques variées de l'architecture proposée (variations de granularité des données, parallélisme, partage de tâches, extensibilité, programmabilité, etc.). Dans un premier temps c'est l'algorithme financier qui est présenté. À granularité fine, il peut être synthétisé en une architecture flot de données. Toutes les étapes du flot de synthèse sont détaillées et les performances sont étudiées pour plusieurs jeux de données d'entrée. Dans un deuxième temps, la méthode de synthèse est appliquée à un algorithme d'authentification d'œuvres d'art. À la différence du premier, les opérateurs et les granularités des données sont variés.

7.1 Les cartes FPGA

Dans ce chapitre, trois cartes FPGA sont considérées pour la synthèse d'architecture (HLS) et la synthèse logique. Contrairement aux outils HLS traditionnels, *SyntheSys* est configuré pour des cartes FPGA, c'est-à-dire :

Un circuit FPGA : les contraintes en ressources du FPGA (LUT, registres et RAM) sont utilisées pour dimensionner le NoC et sélectionner les IP de bibliothèques compatibles.

Des contraintes d'entrées/sorties : les ports du FPGA sont classés (horloges, ports de communication - PCIe/ethernet, LED, etc.) de manière à générer automatiquement les fichiers de contraintes pour l'outil de synthèse logique du FPGA. Les interfaces des IP de la bibliothèque sont aussi marquées afin de les associer aux ports des FPGA avec lesquels ils sont compatibles.

Les cartes utilisées dans ce chapitre sont :

VC707 : une carte de développement Xilinx à base de FPGA Virtex 7,

ML605 : une carte de développement Xilinx à base de FPGA Virtex 6,

MiniX : une carte développée par Adacsys à base de FPGA Spartan 6.

Lorsque les synthèses sont réalisées sans contraintes matérielles, les ressources FPGA sont considérées illimitées.

7.2 Application financière

Nous commençons par décrire l'algorithme puis présenter comment *SyntheSys* l'analyse. Ensuite, nous mesurons les performances en latence et en débit pour différentes applications. Pour ce faire, les mesures au cycle prêt sont faites après simulation du code VHDL généré avec les données produites après exécution du banc de test logiciel. Les résultats sont comparés d'abord à l'exécution d'un programme C, puis au circuit généré par un outil HLS du commerce. Enfin, nous mettons en évidence les avantages d'une architecture programmable en comparant l'utilisation de différentes applications avec *SyntheSys* et un outil HLS classique en prenant en compte les temps de synthèse nécessaires pour regénérer une application.

7.2.1 Présentation de l'algorithme

L'algorithme d'évaluation d'options financières présenté est nommée BOPM pour « binomial option pricing model » [MHB+14]. Très utilisé dans le monde financier, cet algorithme repose sur la construction d'un arbre pour calculer, à partir d'un jeu initial de données, la juste valeur d'une option à un instant donné.

Le programme Python associé à l'algorithme BOPM (voir algorithme 3) consiste à construire un arbre binomial à partir d'un vecteur d'entrée. Les données du vecteur peuvent être des valeurs pour la vente ou l'achat d'options financières.

```
1: function BOPM(InputVector)
2:     for Step in range(len(InputVector)) do
3:         for Step in range(len(InputVector)) do
4:             InputVector[i] ← BOPM_Unit(InputVector[i], InputVector[i+1])
5:     return InputVector[0]
```

Algorithme 3 : L'algorithme BOPM

Ce programme consiste en deux boucles *for* dont le nombre d'itérations est fixé dès la phase d'analyse en fonction de la taille du vecteur d'entrée. Une même instruction est appelée à chaque itération et correspond à la sous fonction *BOPM_unit*.

Celle-ci réalise sept multiplications, quatre additions et soustractions en nombres à virgules flottantes sur des vecteurs de 64 bits (voir algorithme 4). Elle est implémentée en VHDL et est référencée sous forme d'IP dans la bibliothèque de SyntheSys.

```
1: function BOPM_Unit(Item1, Item2)
2:     spots, values_call1, values_put1 ← Item1
3:     spots, values_call2, values_put2 ← Item2
4:     spots ← d*spots
5:     temp_call ← vmax(Rinv*(p*values_call2 + q*values_call1), vmax(spots-
   strike, 0.0))
6:     temp_put ← vmax(Rinv*(p*values_put2 + q*values_put1), vmax(strike-
   spots, 0.0))
7:     Result_values_call ← vmax(temp_call,0.0)
8:     Result_values_put ← vmax(temp_put, 0.0)
9:     return spots, Result_values_call, Result_values_put
```

Algorithme 4 : Contenu de la fonction BOPM_Unit

Le nombre d'instructions *BOPM_unit* exécutées est de $\frac{n(n-1)}{2}$ avec n la taille du vecteur d'entrée. Dans ce programme, il n'y a pas de branchements conditionnels.

7.2.2 Analyse du programme

L'analyse du programme produit un modèle flot de données. La figure 7.1 montre un graphe représentant les dépendances de données, généré après analyse du programme avec un vecteur d'entrée de taille 7. Ce graphe a une forme

d'arbre binomial et diffère un peu du modèle car chaque nœud du graphe représente une donnée. Les arcs, non marqués, représentent les dépendances. Les entrées et sorties sont respectivement en vert et bleu.

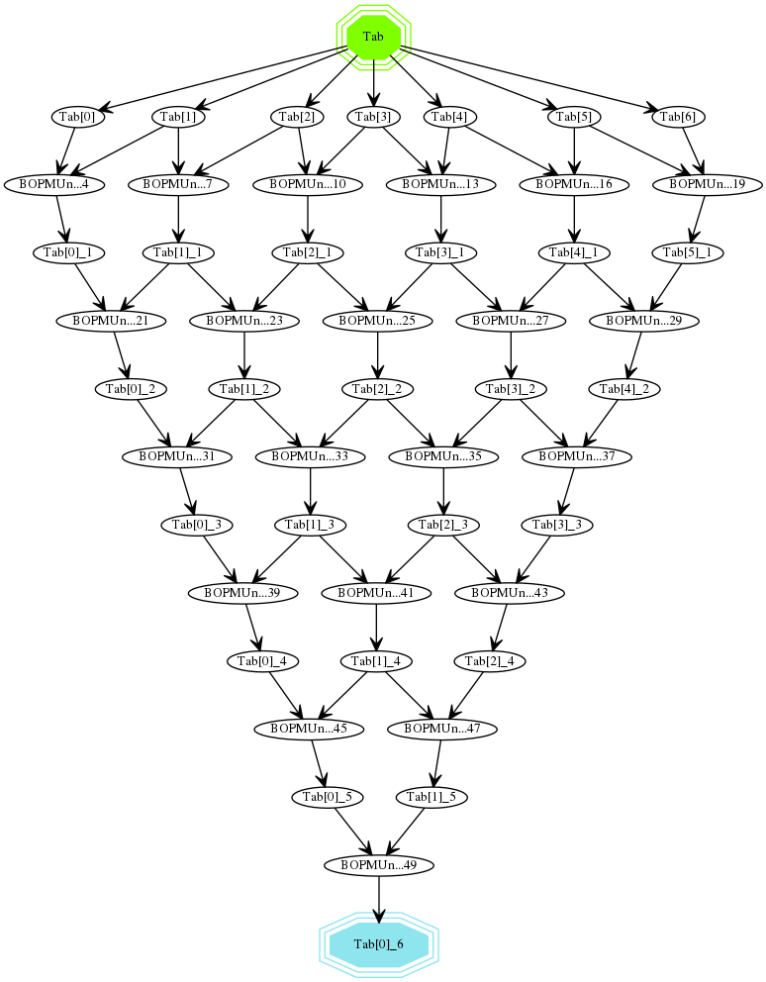


FIGURE 7.1 : Graphe de données pour l'algorithme BOPM (pour 7 valeurs initiales).

Comme le nombre d'itérations est statique (déterminé à l'analyse), les boucles sont entièrement déroulées. Aucune information supplémentaire n'est ajoutée au programme. Le temps requis pour l'analyse de différentes tailles d'arbres est présenté dans le tableau 7.1. L'analyse utilise l'interpréteur Python standard (Python version 3.5.2).

TABLE 7.1 : Durée de l'analyse du programme Python BOPM.

Nombre de tâches	10	15	21	28	36	45
Durée de l'analyse (Python 3.5.2) (ms)	20	29	38	49	62	75

Nous observons que les durées d'analyse augmentent avec le nombre de tâches analysées.

7.2.3 Graphe de tâches

Un graphe des tâches est produit par traduction du modèle flot de données. La figure 7.2 présente un graphe de tâches pour le BOPM à 7 entrées. Des tâches d'entrée et de sortie (« input » et « output ») sont ajoutées et les tâches de calcul correspondent aux fonctions matérielles associées aux fonctions logicielles de la bibliothèque.

Chaque nœud du graphe correspond à une tâche (fonction implémentée par le matériel). Les arcs représentent les communications et le poids des arcs correspond à la quantité de données échangées entre les nœuds.

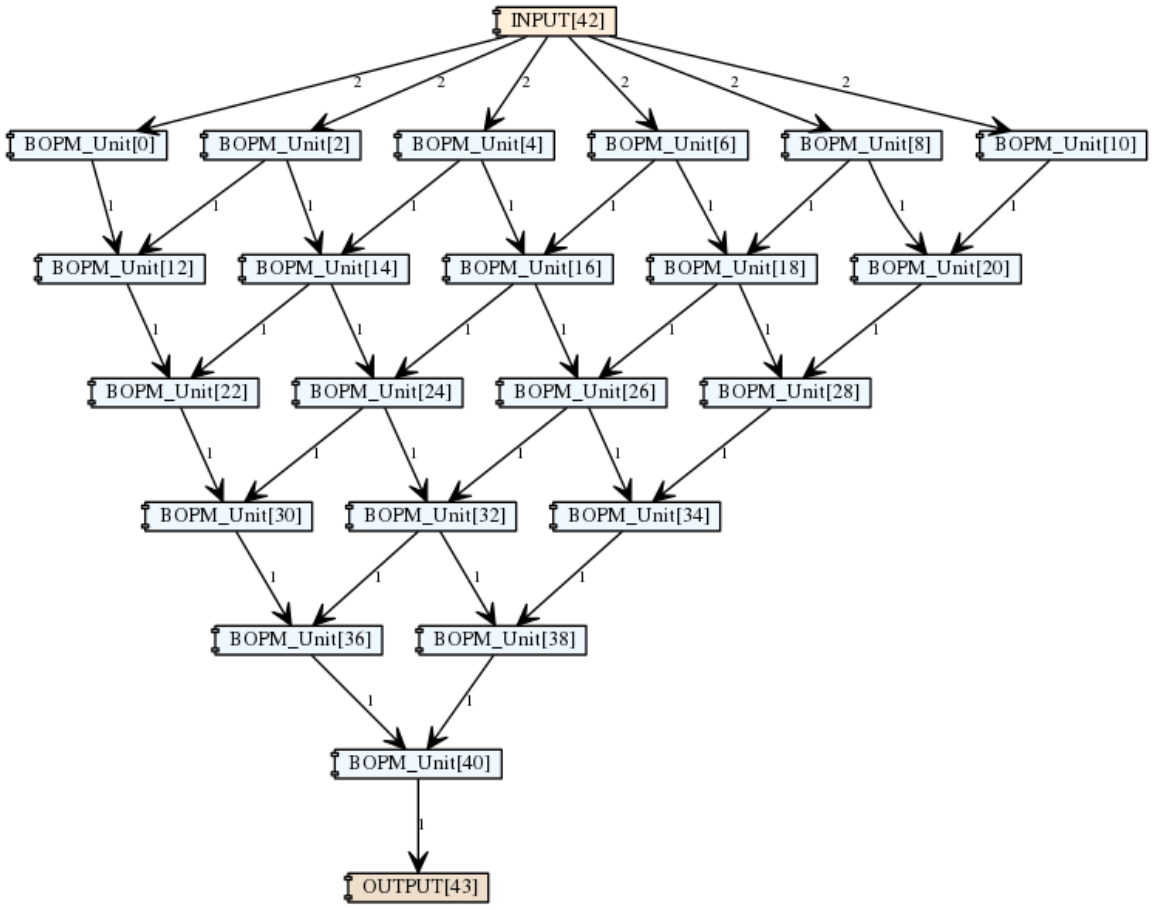


FIGURE 7.2 : Graphe des tâches (CTG pour « communication task graph ») montrant les dépendances de données de l'algorithme BOPM (7 entrées).

Lors de l'analyse, une fonction de bibliothèque est associée aux types de ses données d'entrée (signature d'une tâche). Une liste d'IP candidates est ensuite sélectionnée pour chaque tâche du modèle.

7.2.4 Résultats de simulation

7.2.4.1 Durées de programmation

Le circuit final est simulé puis vérifié sur matériel. Des résultats en termes de durées de configuration sont présentés dans le tableau 7.2. Ils sont donnés pour la cible Virtex 7, et obtenus avec le simulateur Modelsim (simulation RTL).

TABLE 7.2 : Différents types de latence du circuit BOPM pour la cible Virtex 7 (Simulation RTL).

Tailles du vecteur d'entrée	5	6	7	8	9	10
Nombre de tâches correspondantes	10	15	21	28	36	45
Configuration des chemins de données (µs)	1.10	1.59	2.10	2.88	3.36	4.53

Les durées de configuration des chemins de données représentent les temps de remplissage des *tables des en-têtes* de tous les terminaux.

L'évolution de plusieurs types de latences en fonction du nombre de tâches est présentée en figure 7.3. Ces résultats montrent les augmentations de toutes les latences avec le nombre de données d'entrées, et donc d'opérations à effectuer. Le temps de configuration augmente avec le nombre de dépendances et donc d'en-têtes à sauvegarder dans les mémoires des terminaux.

L'augmentation du délai entre deux résultats (inverse du débit) est due au temps de synchronisation au niveau de chaque opérateur. Plus il y a de tâches, plus les temps de synchronisation cumulés sont importants. L'augmentation des temps de communication et de synchronisation sont les principaux impacts de l'architecture (NoC et Flot de données) sur les performances par rapport à une architecture de communication point à point.

7.2.4.2 Latences

La figure 7.4 représente l'évolution des latences de calcul pour plusieurs tailles d'applications et sous différentes contraintes de ressources.

Nous pouvons observer une augmentation différente des latences en fonction des contraintes en ressources (différentes cartes FPGA). Pour un nombre de tâche inférieure à 50, les latences sont les mêmes indépendamment de la cibre FPGA. Cependant, au delà de 50 tâches, plus le FPGA ciblé est petit, plus la latence est grande. En effet, certaines IP sont partagées entre plusieurs tâches. Plus le partage est important, plus la latence augmente.

Nous pouvons observer les différents niveaux de saturation des IP pour chaque FPGA. Plus les ressources disponibles sont importantes, plus le nombre de tâches pour atteindre la saturation est important.

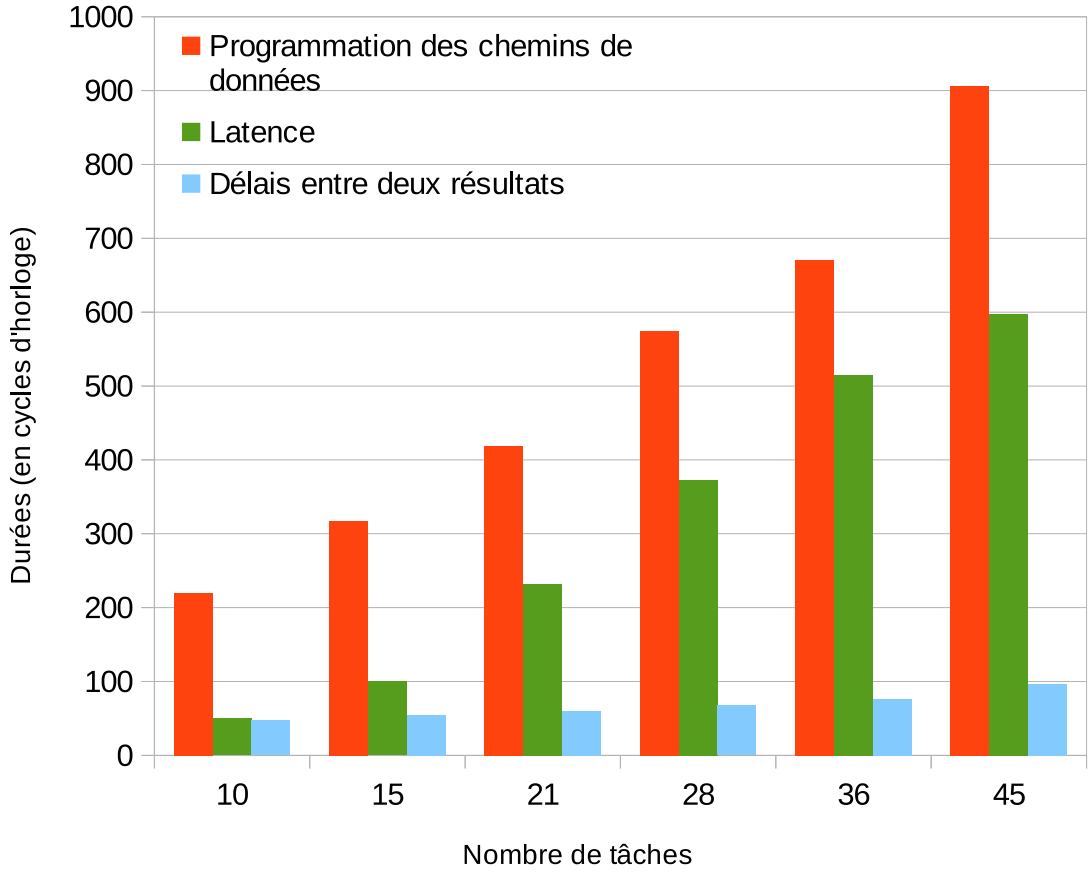


FIGURE 7.3 : Évolution de trois types de latence pour différentes tailles d'application (Virtex 7).

7.2.4.3 Débits

La figure 7.5 représente l'évolution des débits de différents circuits générés en fonction du nombre de tâches du BOPM.

Alors que l'on s'attend à avoir un débit constant pour des synthèses sans contraintes (l'intervale d'injection de données théorique est constant), les débits mesurés décroissent avec le nombre de tâches. Le débit maximal est obtenu avec un circuit généré sans contraintes.

En effet, comme observé précédemment, les temps de synchronisation cumulés dans le réseau augmentent avec le nombre de tâches et donc le débit diminue.

7.2.4.4 Comparaisons avec un programme C

Afin de comparer les performances entre un CPU et un FPGA utilisant notre architecture, les latences d'exécution d'un programme C et celles d'un FPGA

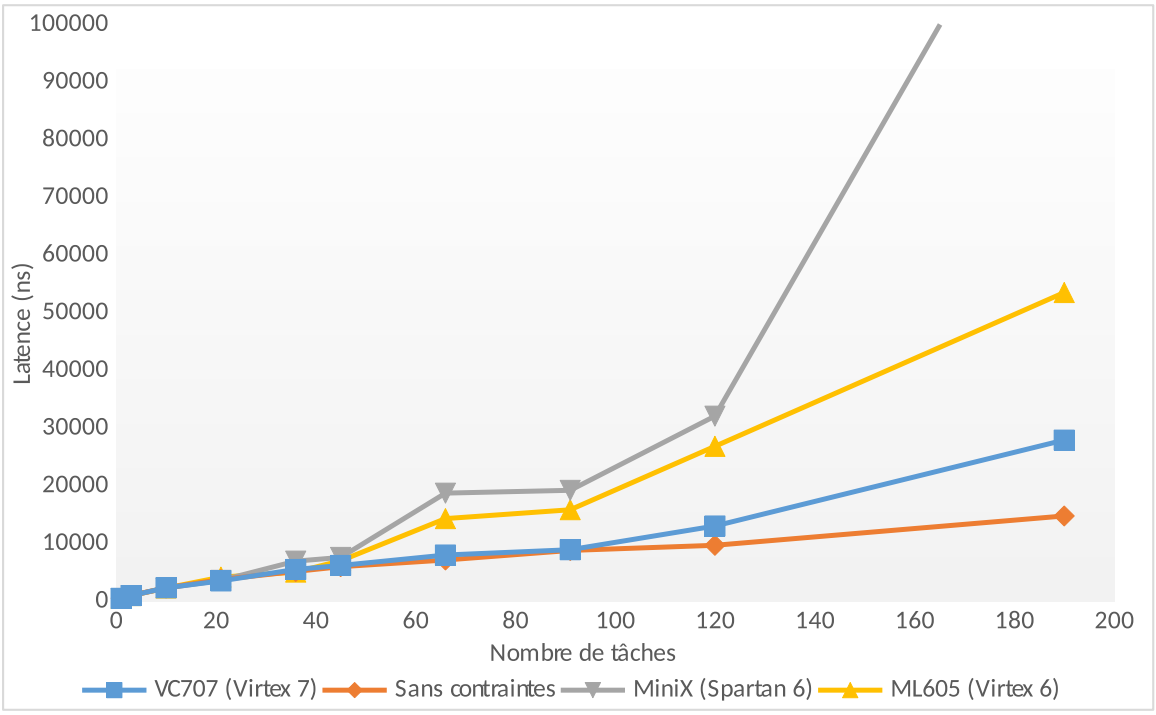


FIGURE 7.4 : Évolution de la latence sur plusieurs FPGA pour différentes tailles d'application.

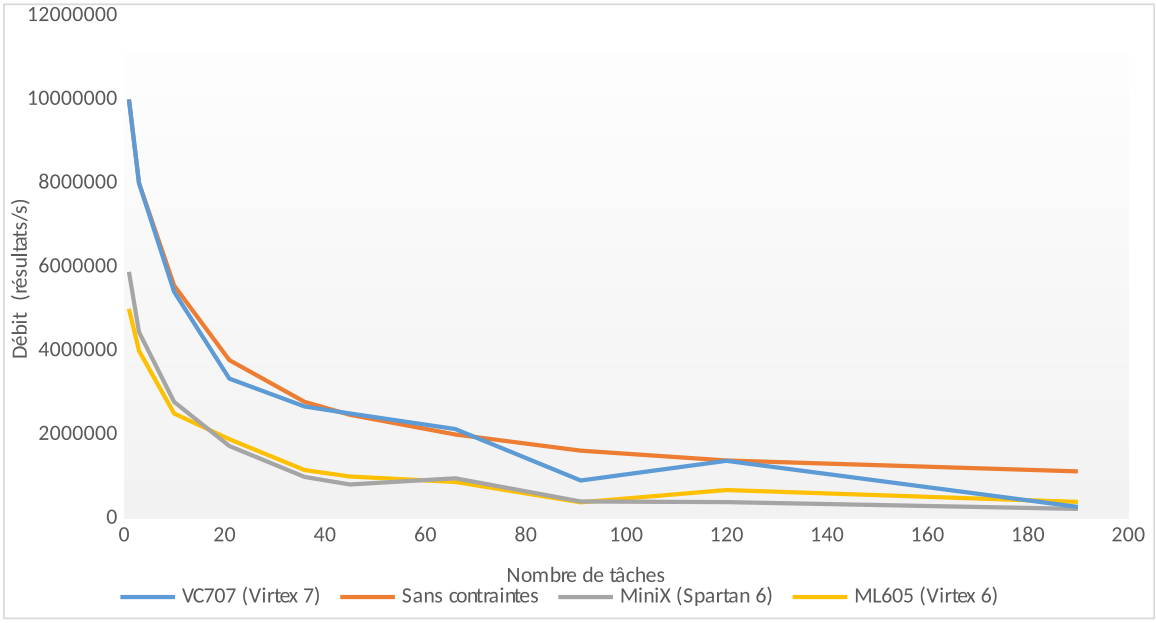


FIGURE 7.5 : Évolution du débit du BOPM sur plusieurs FPGA pour différentes tailles d'application.

Virtex 7 sont mesurées. Une horloge de fréquence 200MHz est utilisée pour cadencer le circuit du Virtex 7. Le programme C est exécuté sur un CPU Intel

Core i7 (3,4 GHz) à 8 cœurs avec un système Linux.

Deux programmes C ont été comparés. L'un avec des directives OpenMP et l'autre sans. Les directives OpenMP ont été ajoutées automatiquement en utilisant l'outil Pluto [BBK+08]. Il s'agit d'un logiciel de transformation source à source pour la parallélisation automatique de code.

La figure 7.6 montre que les latences des circuits générés par *SyntheSys* sont bien inférieures à celles d'un programme C, qu'il utilise ou pas des directives de parallélisation.

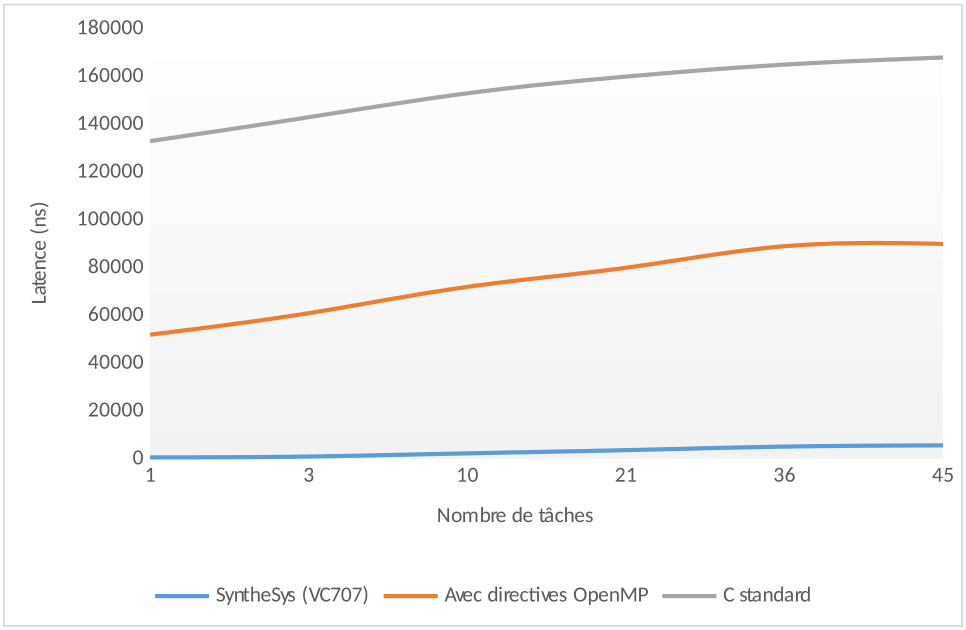


Figure 7.6 : Comparaison des latences entre les circuits générés avec *SyntheSys* et des exécutions sur CPU pour différentes tailles d'application.

La durée de la configuration des chemins de données et du remplissage du pipeline impactent beaucoup le temps total d'exécution pour un faible nombre d'options à évaluer. Mais dans un contexte industriel, ce sont des milliers d'options qui sont évaluées. Dans ce cas, l'utilisation des FPGA devient justifiée.

La figure 7.7 montre que le gain en débit du circuit FPGA (généré pour une cible Virtex 7) présenté par rapport au CPU diminuent avec le nombre de tâches. Le programme C est exécuté dans un environnement linux (Debian 8) avec un processeur Intel Xeon E5-2643 v3 (architecture Haswell) à 6 cœurs de calcul, fonctionnant à une fréquence de 3,40 GHz.

Les gains en débit espérés sont plutôt faibles et diminuent avec le nombre de tâches à éxécuter. Cette diminution est encore plus forte à partir du moment où les tâches sont partagées sur les opérateurs de calcul (à partir de 105 tâches). Ces mauvaises performances semblent indiquer que l'impact du réseau et des mécanismes de synchronisation ainsi que le mécanisme de partage des tâches gère mal l'augmentation du nombre de tâches et par conséquent du traffic réseau.

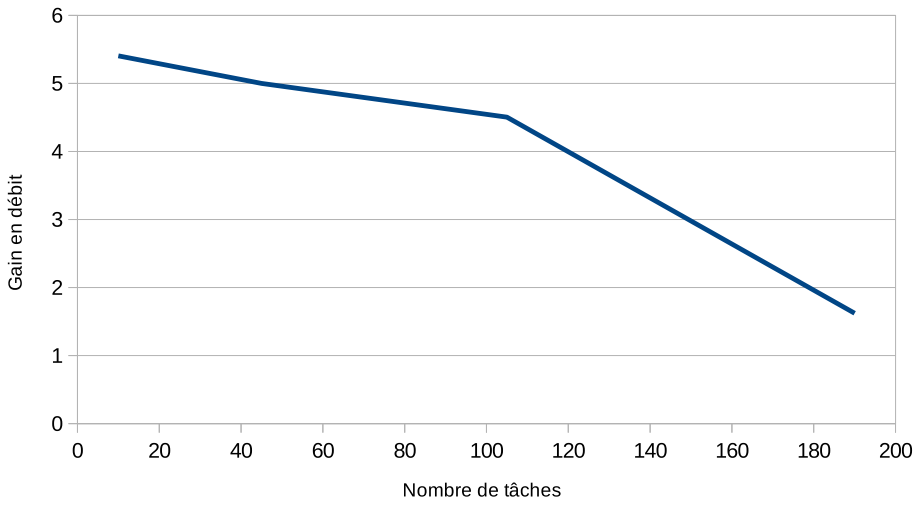


FIGURE 7.7 : Gain en débit entre les circuits générés par *SyntheSys* et programme C pour différentes tailles d'application BOPM.

D'autres mesures avec une fréquence du NoC plus élevée par rapport aux IP permettrait de vérifier si c'est la congestion du réseau qui provoque ces baisses de performances.

7.2.4.5 Comparaisons avec Vivado

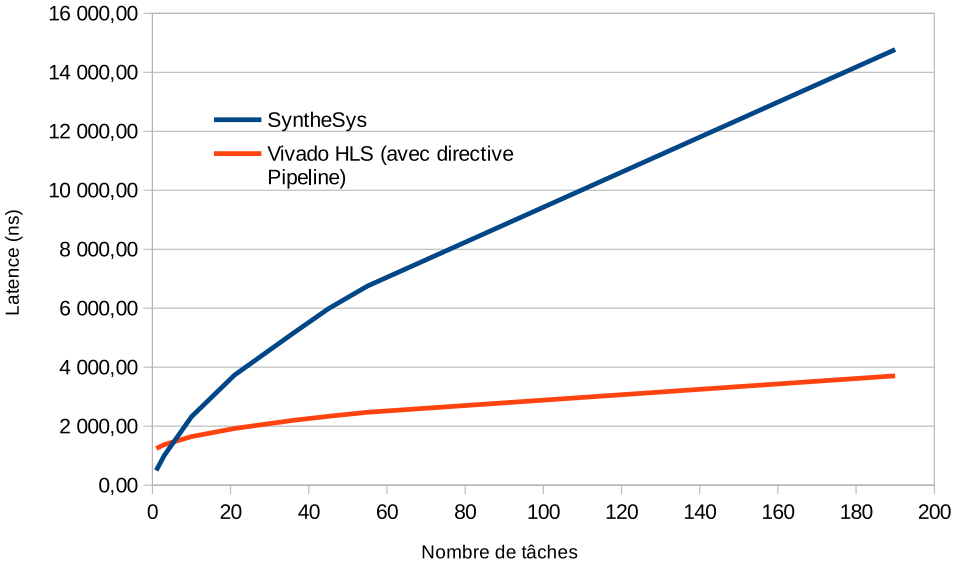


FIGURE 7.8 : Évolution de la latence en fonction du nombre de tâches de l'application BOPM générée avec *SyntheSys* et *Vivado*.

La comparaison des latences obtenues avec des circuits générés par *SyntheSys*

et *Vivado*, un outil de synthèse d'architecture du commerce, est présentée en figure 7.8. Les valeurs, pour *Vivado*, sont données par l'outil de synthèse. Pour *SyntheSys* les mesures sont faites en simulation fonctionnelle. Les résultats montrent de meilleures performances pour l'ensemble des circuits générés par *Vivado*.

Les différences de performances sont essentiellement dues à l'utilisation d'un NoC. En effet, les circuits générés par *SyntheSys* sont entièrement synchrones. La fréquence de fonctionnement du NoC est la même que celle des terminaux. Or, il faut au minimum trois cycles d'horloge pour générer un paquet. Les propriétés d'extensibilité du NoC (maintien en fréquence pour des circuits de grandes tailles) ne sont pas mises en évidence. Dans ce cas, l'utilisation d'un NoC pour cette application ne semble pas justifiée au regard des performances du circuit généré par *Vivado*.

7.2.5 Prototypage FPGA

La fréquence des IP et du NoC peut atteindre 200 MHz sur un FPGA Xilinx Virtex 7. Le comportement du circuit en simulation a été confirmé en utilisant une plateforme d'émulation fournie par l'entreprise Adacsys.

La consommation en ressources pour trois tailles de circuits est présentée en figure 7.9. L'impact du réseau et de l'architecture flot de données sur les consommations globales de ressources FPGA est faible. Le niveau de granularité étant élevé, l'architecture est appropriée pour ce type d'application. Pour trois circuits (de tailles 3, 15 et 24 IP), les ressources utilisées pour le réseau n'excèdent pas 6% et 14% de la consommation totale des LUT et des registres, respectivement. De même les ressources dédiées à la gestion des tâches ne dépassent pas 8% des LUT et 10% des registres.

Dans ce type d'architecture, la mémoire utilisée sur le FPGA pour l'ordonnancement matériel des tâches croît linéairement avec le nombre de dépendances de données ($N_{DataDep}$). La table des données d'entrée stocke un flit par dépendance de données. Le FPGA cible doit être sélectionné en fonction du nombre de dépendances. Puisque la taille d'un en-tête (W_{Flit}) est nécessaire pour identifier la tâche, la donnée transportée et la destination, la mémoire nécessaire M_{used} peut être calculée en utilisant l'expression suivante :

$$M_{used} = 2,5 \times W_{Flit} \times N_{DataDep}$$

Le nombre 2.5 provient du nombre de flit stockés par dépendance de données.

Les empreintes mémoires pour différentes applications BOPM sont présentées dans le tableau 7.3. Le nombre de dépendances de données d'un algorithme BOPM est de $(n-1) \times (n-2) + 1$ pour n tâches ; la tâche de sorties est, elle aussi, considérée. Paralléliser un tel algorithme a donc un coût mémoire d'autant plus élevé que le nombre de dépendances de données est important.

Afin d'illustrer d'autres caractéristiques de l'architecture proposée, une ap-

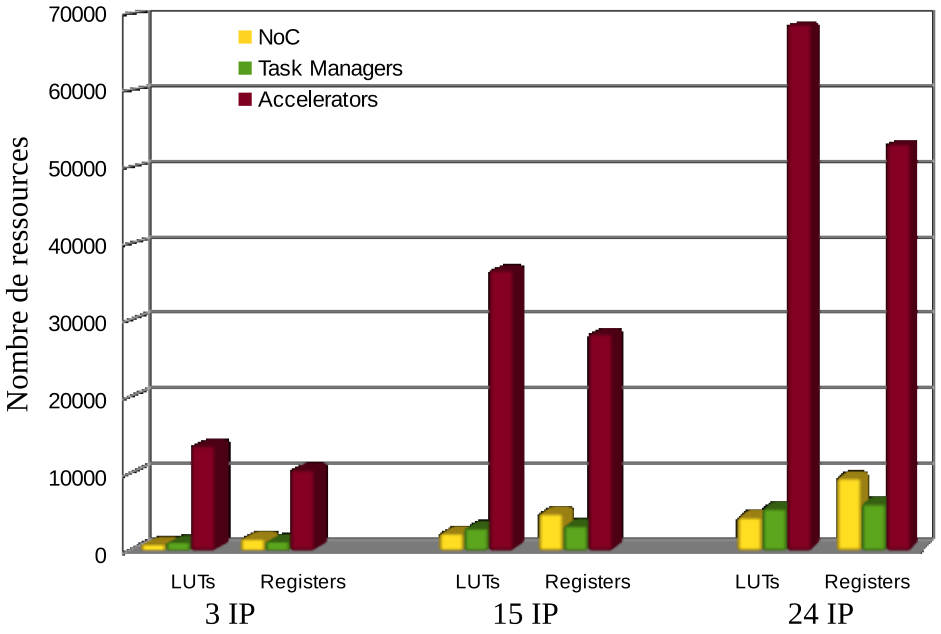


FIGURE 7.9 : Utilisation des ressources par trois circuits BOPM sur un Xilinx Virtex 7.

TABLE 7.3 : Utilisation mémoire pour la gestion des tâches de différentes applications BOPM (flit de 64 bits).

Taille du vecteur d'entrée (bits)	5	6	7	8	9	10
Nombre de tâches	10	15	21	28	36	45
Nombre de dépendances de données	13	21	31	43	57	73
Memoire utilisée (bits)	2080	3360	4960	6880	9120	11680

plication de traitement d'image est présentée dans la partie suivante.

7.2.6 Intérêt de la programmation

Comme beaucoup d'algorithmes, le BOPM est succeptible d'évoluer pour des améliorations futures ou pour la modification des constantes (par exemple la taille du vecteur d'entrée). Les constantes et les chemins de données sont programmables dans le circuit généré par *SyntheSys*, et c'est bien tout son intérêt. Or, un circuit dédié (généré ou non par HLS) nécessite une synthèse logique à chaque modification structurelle liée aux constantes, aux chemins de données ou au contrôle.

Avec *SyntheSys*, dans la mesure où le programme à accélérer ne fait appel à aucune nouvelle fonction, une synthèse logique n'est pas nécessaire. Un changement d'application nécessite deux étapes : une synthèse de haut niveau partielle

du programme (jusqu'au placement sur le réseau) puis une reprogrammation de l'architecture existante (déjà configurée pour un FPGA donné).

L'histogramme présenté en figure 7.10 montre les temps nécessaires pour programmer l'architecture générée pour plusieurs applications sur une même carte FPGA (VC707).

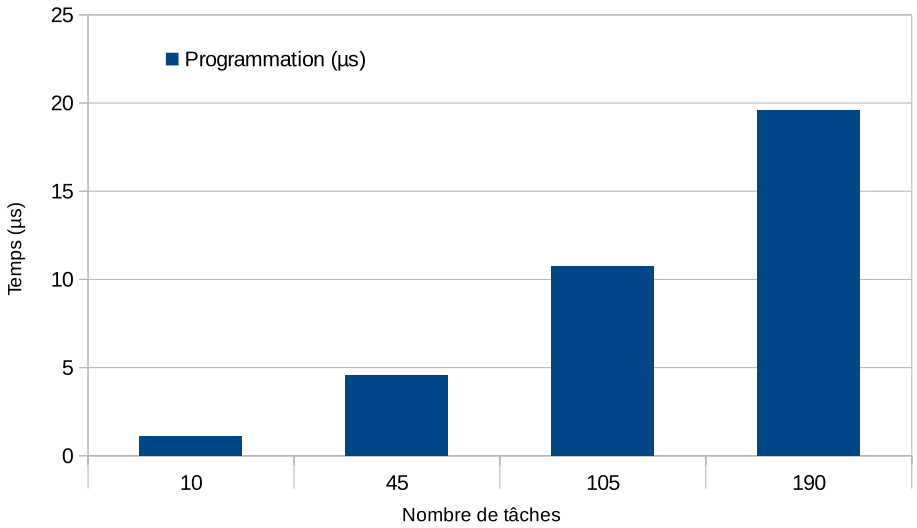


FIGURE 7.10 : Temps de programmation de l'architecture générée par *SyntheSys* pour la carte VC707.

Le temps de programmation présenté correspond à la latence entre l'envoi (depuis l'IP de communication avec le CPU hôte) du premier flit et la réception (par les différents terminaux) du dernier flit de configuration des tables des en-têtes. Il ne prends pas en compte le temps de communication entre le PC et le FPGA. Nous constatons que le temps de reprogrammation dépend du nombre de tâches à programmer et qu'il est de l'ordre de la microseconde.

Cependant, pour générer le programme, il faut refaire une synthèse HLS. L'histogramme présenté en figure 7.11 montre les temps nécessaires pour faire la synthèse HLS avec SyntheSys et *Vivado* HLS, pour plusieurs applications sur une même carte FPGA (VC707).

Ce diagramme montre que pour un faible nombre de tâches (ou d'itérations de la fonction BOPM_Unit) *SyntheSys* est plus rapide. Cependant, à partir de 190 tâches (partage des tâches nécessaire sur la VC707), *SyntheSys* devient plus lent. Ce comportement est dû essentiellement à l'algorithme d'ordonnancement (écrit en Python) qui doit partager les opérateurs à 190 tâches. De plus, *SyntheSys* est exécuté sur une machine moins performante (Intel Core i5, 4 cœurs à 1.70 GHz) pour des raisons pratiques (accès aux serveurs et licences *Vivado*).

Le tableau 7.4 présente les temps de synthèse logique (avec *Vivado*) des IP générés avec *Vivado HLS*. La machine utilisée opère sous linux avec un processeur Intel Xeon E7520 à 32 cœurs de calcul fonctionnant à 1.87 GHz.

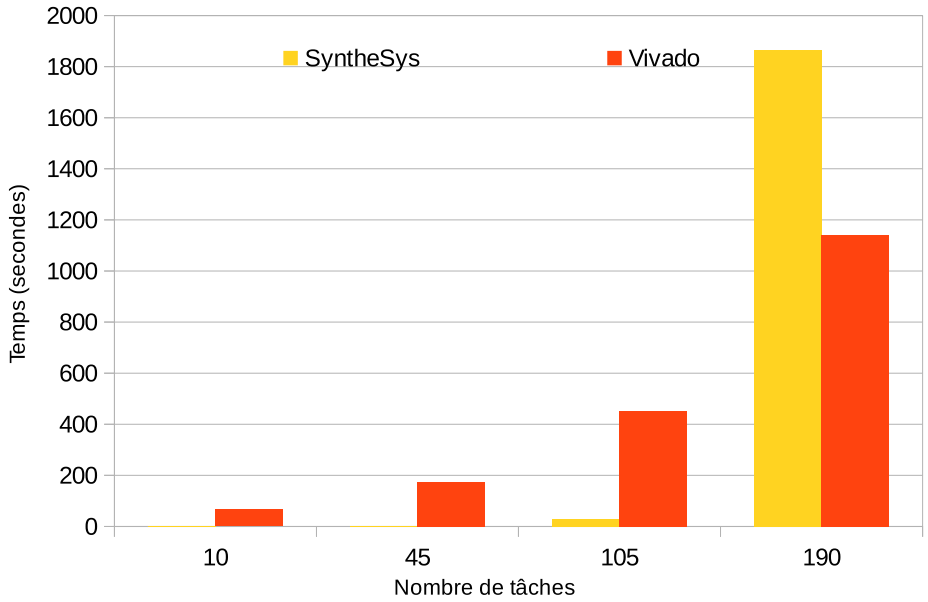


FIGURE 7.11 : Temps de synthèse HLS d'applications BOPM pour la carte VC707.

TABLE 7.4 : Temps de synthèse logique avec *Vivado* pour la carte VC707.

Nombre d'appel à BOPM_Unit	10	45	105	190
Temps de synthèse logique	3h35	4h09	4h46	5h11

Le temps de programmation du circuit généré par *SyntheSys* est très court (quelques microsecondes) comparé à une synthèse logique (plusieurs heures). Autrement dit, *SyntheSys* est particulièrement adapté pour des applications malléables, paramétrables ou succeptibles d'être régulièrement modifiées.

7.2.6.1 Conclusions sur le BOPM

Les résultats montrent qu'une architecture peut être générée pour différentes plateformes FPGA et que les performances de ces circuits dépendent des ressources disponibles lorsqu'il n'est pas possible de mettre tout le circuit sur le FPGA. Des « pipelines » plus longs induisent des latences plus grandes. En régime stable (« pipelines » remplis), plus le nombre de tâches est important, plus les débits diminuent à cause du traffic réseau.

Les performances obtenues grâce à *SyntheSys* sont tout de même meilleures que l'exécution d'un programme C avec ou sans directives parallèles OpenMP. La comparaison avec *Vivado HLS* ne prends pas en compte la communication (synchronisation des données en entrée) mais montre que les performances de *SyntheSys* se situent entre celles de *Vivado HLS* avec et sans directives d'optimisation.

La différence principale de *SyntheSys* par rapport aux outils HLS classiques,

est que le circuit généré est programmable. Un changement dans l'algorithme qui peut exploiter les opérateurs existants nécessite seulement une synthèse HLS (quelques secondes à quelques minutes) et une programmation de l'architecture (quelques microsecondes). Plusieurs heures sont donc économisées par cette méthode.

L'impact de la taille des IP doit aussi être considéré. En effet, des IP optimisées et à grosse granularité sont utilisées dans cet exemple. Elles ont été générées avec l'outil FloPoCo [DD11]. Dans le cas ou le niveau de granularité est plus faible (utilisation des opérateurs standard, par exemple), le « pipeline » global serait ralongé.Or les latences variables du réseau introduisent des temps de synchronisation faisant diminuer les performances globales de l'application. Il est donc toujours préférable, pour ce genre d'application, de choisir la granularité maximale afin de limiter le traffic dans le réseau mais seulement dans la mesure où le parallélisme global n'est pas affecté.

7.3 Application de traitement d'images

7.3.1 Présentation de l'algorithme

L'algorithme AuthMS permet de comparer deux images multispectrales et peut être utilisé pour l'authentification d'œuvres d'art. Une image classique est composée de trois images monochromes (une pour chaque couleur - rouge, vert et bleu). Dans le cas des images multispectrales, des centaines de longueurs d'ondes peuvent être considérées, notamment celles correspondant à des fréquences invisibles à l'œil humain.

L'objectif de cet algorithme est de déterminer si l'image d'une œuvre donnée est celle d'une copie ou de l'originale. L'image de l'œuvre originale est disponible en tant que référence pour la comparaison.

La figure 7.12 illustre l'algorithme. La première étape est le calcul des moyennes des régions pour les deux images multispectrales à comparer. Ensuite, une projection couleur est réalisée afin de mesurer la distance entres les régions dans l'espace des couleurs (rouge, vert et bleu). Si la différence est supérieure à un niveau de précision préalablement établi, les deux images sont différentes. Sinon, un calcul de distance dans l'espace multispectral est réalisé. Un nouveau seuil de précision permet d'établir si les deux images sont différentes ou pas.

7.3.2 Analyse du programme

Le programme implémenté réalise l'algorithme C disponible en annexe. L'analyse du programme Python produit les graphes flots de données (DFG) (voir figure 7.13). Tout comme avec le BOPM, les IP sont préalablement développées, vérifiées et référencées en bibliothèque. Nous avons utilisé l'outil FloPoCo pour générer les opérateurs arithmétiques composant nos IP.

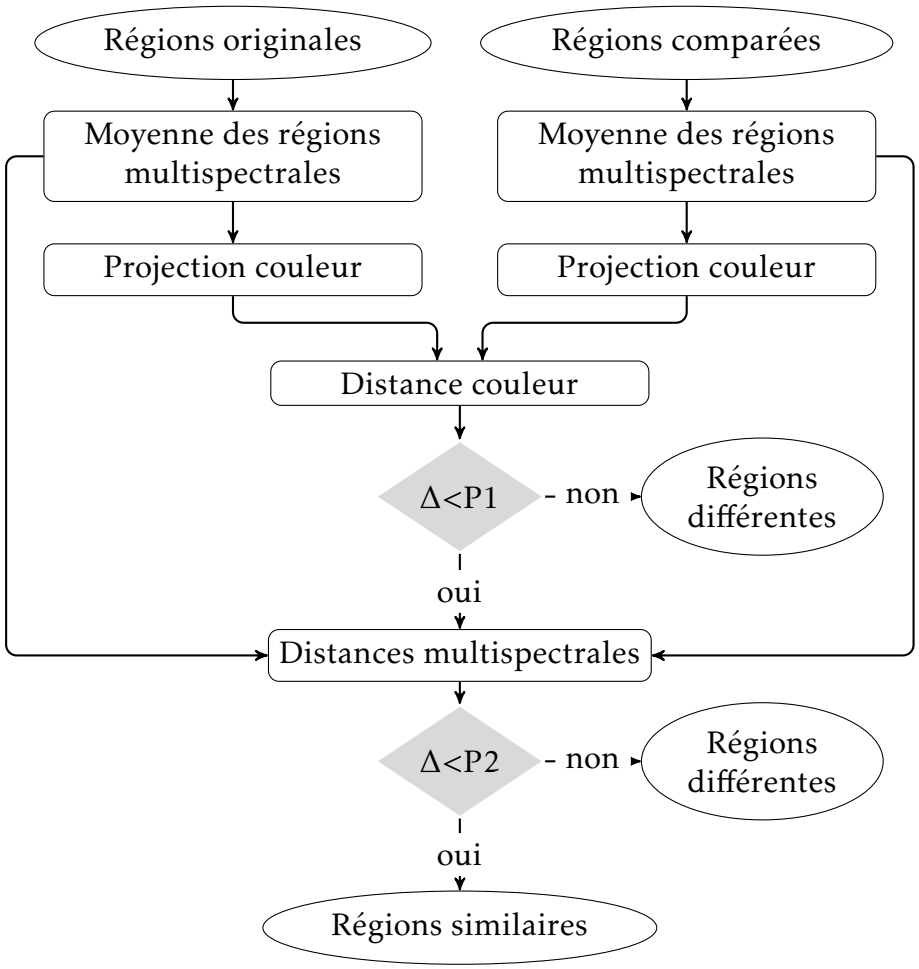


FIGURE 7.12 : Algorithme d'authentification d'œuvres d'art

Contrairement au BOPM, les IP de cet algorithme manipulent des vecteurs. Les vecteurs sont considérés par notre outil (*SyntheSys*) comme des données de grosse granularité. Dans l'application AuthMS différentes tailles de vecteurs sont manipulées ce qui correspond à des échanges de paquets de tailles différentes dans le réseau.

7.3.3 Graphe de tâches

Le graphe des tâches de communication (TCG) produit est présenté en figure 7.14. Les tâches « reduce » ont été introduites afin de construire un vecteur à partir des scalaires résultant des calculs de moyennes. En effet, les opérateurs suivants (AxisXYZ) manipulent des vecteurs de moyennes. Au niveau du réseau, cela correspond à générer un paquet unique rassemblant les moyennes récupérées dans les différents paquets provenant des opérateurs « RegionMean ». Ces données peuvent arriver dans n'importe quel ordre mais doivent être correctement ré-ordonnées pour ensuite construire le nouveau vecteur.

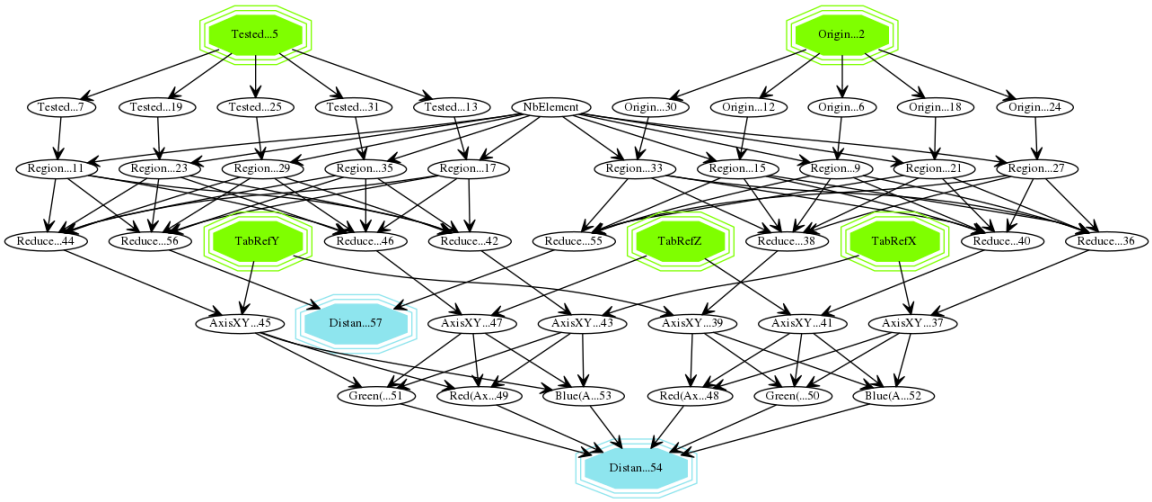


FIGURE 7.13 : Graphe flot de données pour l'algorithme d'authentification d'images multispectrales (pour une région de 5 longueurs d'onde).

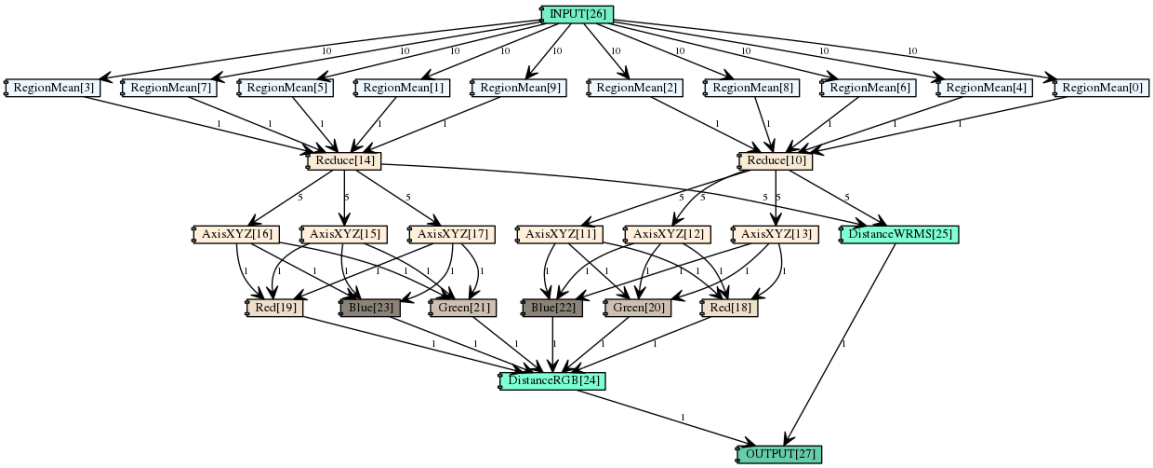


FIGURE 7.14 : Graphe des tâches de communication pour l'algorithme d'authentification d'images multispectrales (pour une région multispectrale à 5 longueurs d'onde).

Le parallélisme potentiel de l'application dépend du nombre de régions à traiter. Les opérateurs utilisés font du calcul en virgule flottante, moins performants que les opérateurs en virgule fixe, pour des raisons de simplicité. En effet, utiliser des opérateurs en virgule fixe demande une analyse de la précision qui n'est pas encore fonctionnelle dans l'outil *SyntheSys*.

7.3.4 Résultats de simulation

Le tableau 7.5 récapitule les caractéristiques des opérateurs utilisés dans l'application AuthMS générée. La latence correspond à la durée entre la consom-

mation de la première donnée d'entrée et la production de la dernière donnée de sortie. Le DII (pour « data introduction interval ») correspond au temps nécessaire entre deux consommations successives de données.

TABLE 7.5 : Caractéristiques des opérateurs utilisés dans AuthMS (cas où le nombre de raies - ou longueurs d'ondes - est de 5). Les valeurs sont données en nombre de cycles d'horloge.

Nom de l'IP	**Latence**	**DII**
RegionMean	158	15
AxisXYZ	82	15
Red	37	1
Green	37	1
Blue	37	1
DistanceRGB	108	1
DistanceWRMS	221	15

Les résultats en termes de latence sont présentés en figure 7.15 en fonction du nombre de longueurs d'onde, pour différents nombres de régions à traiter. Deux courbes sont présentées : une pour le circuit généré par *SyntheSys* et l'autre pour la durée d'exécution d'un programme C réalisant l'algorithme. Le programme C est exécuté dans un environnement linux (Debian 8) avec un processeur Intel Xeon E5-2643 v3 (architecture Haswell) à 6 cœurs de calcul, fonctionnant à une fréquence de 3,40 GHz.

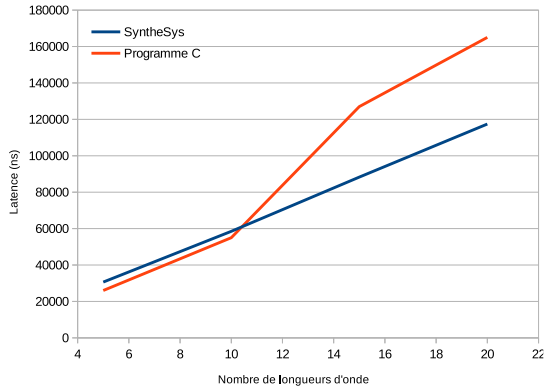
De manière générale, nous observons une augmentation de la latence en fonction du nombre de longueurs d'onde à traiter. Cependant cette augmentation est faible relativement à celle du programme C.

L'accélération par rapport au programme C est d'autant plus grande que le nombre de longueurs d'onde est important. C'est le parallélisme qui limite l'augmentation de la latence dans le circuit produit par *SyntheSys*. Cependant, pour des faibles nombres de régions à traiter et un petit nombre de longueurs d'ondes, le programme C est plus rapide. L'algorithme n'est donc accéléré qu'à partir de 10 longueurs d'ondes pour 25 régions à traiter.

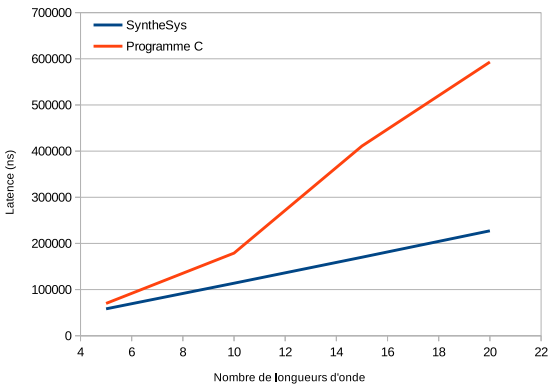
7.4 Conclusion

Nous voulions montrer dans quelle mesure l'architecture flot de données proposée peut accélerer un programme et comment sa programmabilité facilite la réutilisation d'un circuit déjà configuré sur FPGA.

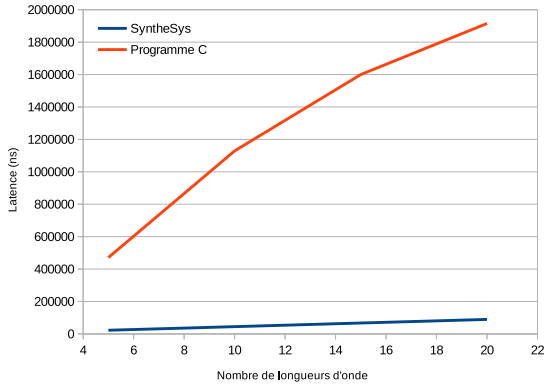
Les deux algorithmes proposés disposent d'un parallélisme potentiel élevé. Nous avons montré que l'accélération dépendait de l'architecture cible, du paral-



(a) Pour 25 régions



(b) Pour 50 régions



(c) Pour 100 régions

FIGURE 7.15 : Évolution de la latence totale de calcul en fonction du nombre de longueurs d'onde.

lélisme potentiel et de l'exploitation du « pipeline » de l'architecture (pour un grand nombre de données traitées).

Les circuits générés pour l'application financière sont plus performants que les programmes C optimisés ou non pour des processeurs à architectures parallèles. Cependant, la comparaison avec un outil de synthèse du commerce montre que l'accélération est faible comparée à des architectures dédiées. Toutefois, cette comparaison est à relativiser par la non prise en compte de la communication sur puce. En effet, la mesure avec Vivado suppose que l'ensemble des données est immédiatement disponible en entrée du circuit. Or, nous pouvons supposer que le temps d'acheminement des données entre le contrôleur de communication et l'accélérateur augmente avec le nombre de données d'entrée. La mesure des latences pour le circuit généré par *SyntheSys* prend en compte ce temps.

En revanche, les performances des circuits générés par *SyntheSys* pour l'algorithme d'authentification multispectrale ne sont pas toujours meilleures qu'avec un programme C. Leurs performances outrepasse celle des CPU au delà d'un certain nombre de régions à traiter (taille de l'image assez grande) pour un

nombre de longueurs d'onde donné. En effet, l'utilisation d'opérateurs flottants sur 64 bits rendent les latences des IP très grandes. Or, un des intérêts des FPGA est de pouvoir implémenter des opérateurs dédiés, notamment pour faire des calculs en virgule fixe. Une accélération encore meilleure est donc possible.

Certaines applications disposent de nombreux paramètres pouvant être adaptés pour différentes raisons. C'est le cas pour l'algorithme d'authentification multispectrale (tailles de l'image, tailles des régions, nombre de longueur d'onde, etc.). Nous avons montré que, dans de tels cas, les circuits programmables générés par *SyntheSys* peuvent faire économiser plusieurs heures de synthèse logique.

Le chapitre suivant conclut cette thèse et montre quelques perspectives de recherche pour une plateforme d'accélération par FPGA accessible aux programmeurs.

Chapitre 8 : Conclusion générale et perspectives

La racine du travail est amère, mais son fruit est doux
Proverbe danois

Sommaire

Ce chapitre conclut cette thèse avec, dans un premier temps, une synthèse des travaux réalisés et des résultats obtenus. Puis dans un deuxième temps, une liste de propositions d'améliorations et de poursuites de recherche est proposée.

8.1 Conclusion

La synthèse optimisée de systèmes est un problème complexe. Afin de palier à un plafonnement des fréquences de fonctionnement des systèmes, l'augmentation des performances passe maintenant par l'exploitation du parallélisme.

Dans les processeurs, l'exploitation du parallélisme a été rendue possible (historiquement) par l'utilisation de mécanismes de type flot de données, d'abord

de manière transparente aux programmeurs (algorithme de Tomasulo), puis de manière explicite (multithreading, OpenMP, etc.), mais avec moins de succès à cause de l'expertise exigée. La difficulté est multiple : extraire le parallélisme d'une spécification comportementale initiale, définir une architecture extensible et programmable et utiliser efficacement cette architecture.

Les réseaux sur puce sont nés du constat que les solutions de communication classiques sur puce n'était pas extensibles. Cependant, développer une application en réseau suit des contraintes particulières et requiert une méthodologie propre.

Les FPGA permettent l'utilisation d'opérateurs spécifiques et des infrastructures de communication sur mesure. Leur utilisation conjointe avec des processeurs pour faire de l'accélération de calcul s'appelle l'informatique reconfigurable. Ce domaine émergent ne bénéficie pas encore d'architecture fiable, d'interface normalisée ou encore d'outil d'analyse et de synthèse de haut niveau accessible aux non-experts.

Certaines contraintes rendent difficile l'adoption des FPGA pour leur utilisation en informatique reconfigurable. Parmi elles, nous pouvons citer une extraction facilitée du parallélisme potentiel, l'interfaçage entre le FPGA et le système d'exploitation, la compatibilité avec les différents matériels existants et enfin des contraintes de temps de développement et de synthèse logique.

Dans cette thèse, nous proposons un flot de synthèse qui a été mis en œuvre à travers l'outil *SyntheSys*. Cette méthode intègre la synthèse de NoC au flot de synthèse d'architecture classique. L'utilisation d'un réseau pose des difficultés notamment concernant la variabilité des latences qui rend difficile l'optimisation et l'estimation des performances du circuit généré.

Demander au programmeur toujours plus d'informations et d'expertise n'est pas, de notre point de vue, la solution au problème de l'accélération. Aujourd'hui, chaque outil de synthèse d'architecture requiert la réécriture et l'enrichissement du code du programme pour être correctement compilé et parallélisé. C'est pourquoi nous avons choisi de proposer l'analyse d'un language de très haut niveau qui ne requiert pas de connaissances architecturales pour l'accélérer sur FPGA.

Une bonne synthèse commence par une bonne analyse. Dans cette thèse, nous avons proposé une méthode d'analyse symbolique qui s'appuie sur un interpréteur afin de faciliter la parallélisation et de bénéficier de fonctionnalités abstraites du langage. Il s'agit d'une approche originale inspirée des techniques de vérification logicielle (vérification formelle).

Pour son implémentation sur FPGA, nous avons choisi d'utiliser un modèle et une architecture de types flot de données, insensibles à la latence. Le modèle suppose des mémoires non bornées au niveau des liens de communication, ce qui demande un dimensionnement correcte pour éviter les inter-blocages.

Afin de définir la taille maximale du NoC pouvant être configurée sur le FPGA, nous estimons les ressources consommées par l'ensemble du circuit à l'aide d'un modèle mathématique. Ce modèle est préalablement construit à l'aide

d'une méthode généralisable à tous les NoC du type de celui utilisé.

Notre mode de stockage temporaire des données pour la synchronisation au niveau des nœuds nécessite d'une part, que les paquets envoyés, dans un ordre donné, par terminal arrivent dans ce même ordre au destinaire. D'autre part, l'ordre des paquets arrivant à un terminal peut varier en fonction des conditions de traffic dans le réseau et des « distances réseau » entre source et destination. Pour éviter un dépassement de flux (ordre du « pipeline »), un mécanisme de réordonnancement des paquets reçu doit permettre de retarder l'arrivée d'un paquet d'un flux suivant si les données d'un flux précédent n'ont pas encore été toutes consommées.

Nous avons montré comment une telle architecture peut fonctionner pour deux types d'applications. L'utilisation d'une architecture comme celle proposée peut effectivement accélérer ces programmes en utilisant un FPGA. L'accélération obtenue dépend du parallélisme potentiel de l'algorithme (dépendances de données) et du parallélisme disponible de l'architecture (limitée par les ressources disponibles du FPGA).

Le réseau a cependant un impact important sur les performances des systèmes générés. Les latences variables nécessitent des synchronisations qui diminuent d'autant plus le débit que le nombre de tâches est important.

Cependant, si les performances observées dans nos expérimentations restent plus faibles que celles des circuits générés par des outils de synthèse d'architecture classique, le système généré peut encore être amélioré pour prendre en compte les différences de fréquences entre opérateurs. En effet, l'outil *SyntheSys* ne gère pas encore les circuits à plusieurs domaines d'horloge. Or, les mesures n'ont été effectuées qu'avec des circuits entièrement synchrones, qui ne profite pas des fréquences de fonctionnement maximales de certains terminaux.

Mais le principal avantage des circuits générés par *SyntheSys* est leur programmabilité. Pour une cible FPGA donnée, plusieures heures peuvent ainsi être économisées en reprogrammant le circuit plutôt que regénérer l'architecture complète.

Nos travaux peuvent faire l'objet de nombreuses améliorations (optimisation de la synthèse et des circuits générés) et permettent de dégager d'autres perspectives de recherche. La partie suivante développe plusieurs de ces améliorations et perspectives.

8.2 Perspectives

8.2.1 Gestion du multicast

Une limitation du NoC utilisé est le besoin de générer autant de paquets que de destinataires des résultats d'un calcul. Un mécanisme d'envoi en multicast permettrait de diminuer le temps de génération des paquets. En effet, pendant la génération des paquets, l'opérateur est mis en attente. Il serait donc intéressant

de mesurer l'impact d'un tel mécanisme sur les performances. Les applications dont les données produites par un terminal sont envoyées à un grand nombre d'autres terminaux (avec des nœuds à degrés sortant élevés) devraient le mieux profiter de cette fonctionnalité.

8.2.2 Partage d'applications

L'exploitation des opérateurs dans l'architecture proposée est indépendante de la notion d'application. En effet, seule l'information des destinataires pour chaque tâche est configurée au niveau d'un terminal. Il est donc tout à fait possible d'exécuter plusieurs applications sur un même ensemble d'opérateurs matériels.

L'ajout et la suppression d'application dans l'architecture peuvent être faits dynamiquement sans arrêter les autres applications. Pour que cela soit possible, il est nécessaire de sauvegarder l'état d'utilisation des opérateurs, notamment l'utilisation des tables des en-têtes, soit dans un serveur gérant l'utilisation du FPGA, soit dans le FPGA lui même.

8.2.3 Migration de tâches

Dans notre architecture, une tâche X est identifiée par l'adresse pointée par la table des en-têtes d'un terminal T_X. Or cette adresse est sauvegardée au niveau de l'en-tête du paquet émis par le terminal T_Y exécutant la tâche Y précédente, c'est-à-dire celle qui produit le résultat consommé par la tâche X.

Migrer la tâche X du terminal T_X à un autre, disons T_Z, est possible. Seules deux étapes sont alors nécessaires :

1. Copier les en-tête des paquets générés après exécution de la tâche X du terminal d'origine T_X vers le terminal cible T_Z.
2. Modifier l'en-tête stocké dans le terminal T_Y pour qu'il cible la tâche X du terminal T_Z.
3. L'ensemble des terminaux précédents doit signaler la prise en compte de la migration à l'ordonnanceur qui peut alors considérer l'adresse associée à la tâche X du terminal T_X libérée.

La liste des en-têtes associée à la tâche X du terminal d'origine T_X ne doit pas être effacée tant qu'il y a des paquets dans le réseau qui ont été générés avec l'ancien ordonnancement. Pour faire cela, une solution est que les terminaux des tâches précédentes envoient un paquet indiquant la prise en compte de la migration au terminal T_X qui exécutait la tâche migrée. De cette manière, comme les paquets arrivent dans l'ordre dans lequel ils ont été envoyés, la réception du paquet de prise en compte de la migration signifie qu'il ne reste plus, dans le réseau, de paquets émis par la source qui résultent de l'ancien ordonnancement.

8.2.4 Partage d'un FPGA pour plusieurs utilisateurs

Des travaux ont déjà été réalisés au sujet du partage d'un ensemble de FPGA pour un ensemble d'utilisateurs [DKW99]. Mais la granularité du partage se situe au niveau du FPGA (un FPGA par utilisateur).

Nous pouvons pousser le partage au niveau des opérateurs dans le FPGA. En effet, si plusieurs applications peuvent être exécutées sur un même FPGA, plusieurs utilisateurs (à chacun ses applications) peuvent donc se partager un ensemble de ressources. Le partage des opérateurs peut être modifié selon le niveau d'exclusivité (ou de priorité) de l'utilisateur. L'utilisateur le plus prioritaire pourrait bénéficier de l'excusivité de l'utilisation de certains terminaux pendant que les autres se partageraient le reste des terminaux disponibles. Cette fonctionnalité nécessite cependant le mécanisme de migration de tâches.

8.2.5 Applications multi-FPGA

Les NoC répartis sur plusieurs puces (NoMC pour « Network on Multi-Chip ») ont fait l'objet d'études dans [SLS10, SV06, CCL+10]. L'extension sur plusieurs puces est rendue possible par l'utilisation d'un NoC hiérarchique. Trois niveaux hiérarchiques sont utilisés : le niveau système, le niveau « cluster » et le niveau local. L'établissement de liens dynamiques entre les puces est réalisé au niveau système.

La gestion des tâches proposée dans cette thèse peut supporter des dépendances de tâches entre plusieurs FPGA si l'information d'adressage multi-FPGA est présentée dans les en-têtes stockés au niveau des terminaux.

8.2.6 L'auto-adaptation

La modification du comportement d'une application en réponse à l'évolution d'une contrainte est aussi appelée auto-adaptation. Une application est dite malléable lorsqu'un ensemble de tâches peut être attribué à un ensemble d'éléments de calcul à une date donnée. Elles sont capables d'adapter leur degré de parallélisme en fonction du nombre de cœurs disponibles.

Les premières méthodes de gestion des tâches étaient centralisées. Depuis 2011, les premiers systèmes décentralisés sont apparus afin de résoudre les problèmes de formation de points chauds (zones de congestion) dans le trafic. La méthode de gestion distribuée nommée DistRM [KBL+11], à l'usage des systèmes many core, utilise le concept des clusters.

Une poursuite naturelle des recherches réside dans la conception de mécanismes pour migrer les tâches en réponse à un stimuli comme par exemple, la mise hors service d'un terminal (erreur matérielle) ou bien encore un changement de mode de fonctionnement (économie d'énergie).

8.2.7 Le FPGA en « cloud »

La plateforme générée peut être utilisée pour ce que l'on appelle le « cloud FPGA » ou FPGA en nuage, c'est-à-dire l'utilisation des FPGA à distance. L'outil *SyntheSys* dispose pour cela de fonctionnalités pour effectuer la synthèse FPGA sur des serveurs distants, ce qui évite aux utilisateurs de devoir acquérir une licence pour faire de la synthèse logique sur les FPGA qu'ils utilisent. Le fichier de configuration du FPGA peut ensuite est téléchargé sur une des cartes disponibles sur les serveurs prévus à cet effet, évitant à l'utilisateur de devoir acheter et gérer le matériel, et notamment la partie communication avec le FPGA. Un synopsis de ces fonctionnalités est présenté sur la figure 8.1.

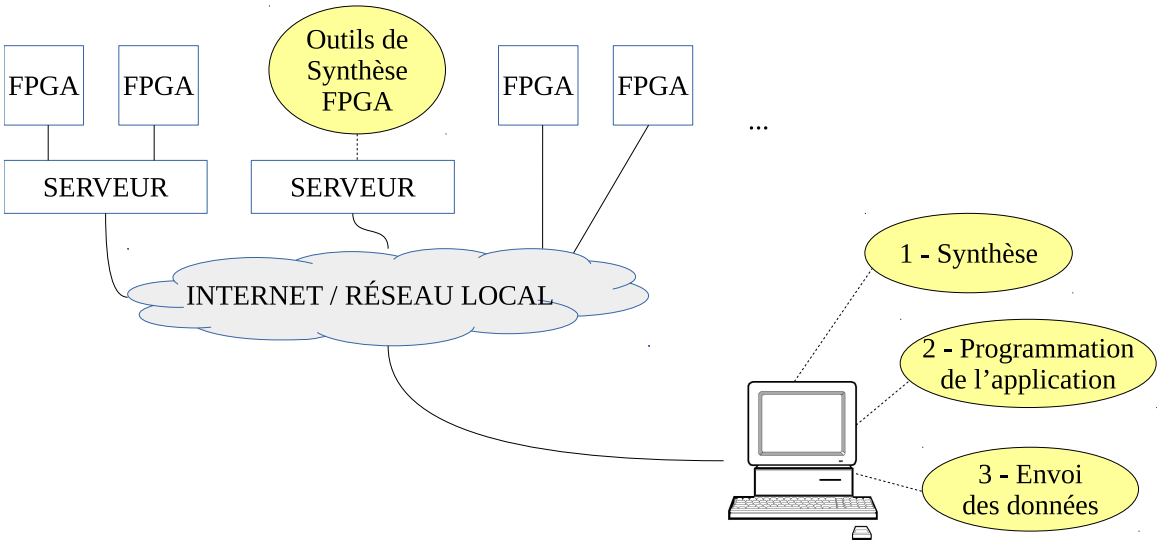


Figure 8.1 : Utilisation de *SyntheSys* dans le cadre des FPGA en nuages (« Cloud FPGA »)

Ces fonctionnalités étaient nécessaires à Adacsys afin de d'offrir ses services d'accélération de manière dématérialisée.

SyntheSys supporte déjà la synthèse logique et la programmation des FPGA en réseau. Il manque l'utilisation de manière transparente d'un ensemble de FPGA pour un ensemble d'utilisateurs (gestion des accès et du partage du matériel).

8.2.8 Association avec d'autres outils HLS

L'architecture proposée fait communiquer des accélérateurs de granularité variable. Nous avons conclu qu'une granularité trop faible avait un impact négatif sur les performances à cause de la prédominance des latences de communication sur les latences de calcul.

La plupart des outils classiques de HLS produisent des accélérateurs (granularité élevée). Faire une synthèse HLS partielle des sous-graphes de données de

l'application à faible granularité permettrait d'éviter une chute de performance. Les deux outils sont donc complémentaires.

Une utilisation conjointe des deux outils nécessite de partitionner le graphe flot de données produit pour isoler les sous-graphes qui vont former un accélérateur. Celui-ci peut-être généré par un outil de synthèse HLS classique après génération du code d'entrée adapté à cet outil (C annoté en général) reproduisant le comportement décrit par le sous-graphe extrait.

8.2.9 Systèmes embarqués

Le système généré utilise des IP de communication pour communiquer avec le CPU hôte. Les entrées et sorties peuvent provenir de convertisseurs analogique/numérique, de générateurs de nombres aléatoires ou bien encore d'une caméra. *SyntheSys* pourrait, danc ce cas, être utilisé pour des systèmes embarqués. Cependant un travail doit encore être fait sur les garanties de service proposés par le réseau afin de l'utiliser dans des applications temps réels à contraintes dures.

8.2.10 Gestion de la mémoire

Une des limitations du système proposé est la faible quantité de mémoire disponible sur le FPGA. En effet, la taille et le nombre de données échangées entre IP est limitée par la taille des mémoires tampons utilisées pour la synchronisation des données. Par exemple, une image de cinq méga pixels ne peut être stockée sur un FPGA avec la technologie d'aujourd'hui. Une solution est d'utiliser des mémoires externes plus grandes. Bien-sûr, le choix de ces mémoires et comment les utiliser ne doit pas être demandé à l'utilisateur non expert. L'automatisation de ces tâches est encore à étudier.

Liste des publications

Conférences internationales

[1] **Matthieu Payet**, Virginie Fresse, Frédéric Rousseau, Pascal Rémy. Dynamic Data Flow Analysis for NoC Based Application Synthesis. In *Proceedings of the 2015 International Symposium on Rapid System Prototyping (RSP)*, Amsterdam, Netherlands, 2015, pp. 61-67. ***[Publié]***

[2] Virginie Fresse, Catherine Combes, **Matthieu Payet**, Frédéric Rousseau. Methodological Framework for NoC Resources Dimensioning on FPGAs. The 2nd International Workshop on Design and Performance of Networks on Chip. In *Procedia Computer Science*, vol. 56, pp. 391-396, 2015. ***[Publié]***

Journaux internationaux

[3] Virginie Fresse, Catherine Combes, **Matthieu Payet**, Frédéric Rousseau. NoC Dimensioning from Mathematical Models. In *International Journal of Computing and Digital Systems*, Volume 5 , Issue 2, February 2016. ***[Publié]***

Conférences nationales

[4] **Matthieu Payet**, Virginie Fresse, Frédéric Rousseau et Pascal Rémy. Extraction du parallélisme à l'exécution pour la synthèse d'applications basées sur un NoC. Pour la conférence en parallélisme architecture et système - COMPAS, Lille, France, 2015. ***[Publié]***

References

[AANS+14] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal, and Mikel Luján. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[Agi10] ADS Agilent. Agilent technologies. *Yarnton, England*, 2010.

[AJR+05] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid. System level synthesis of dataflow programs : HEVC decoder case study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, 2013-05.

[AK02] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures : a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[BB04] D. Bertozzi and L. Benini. Xpipes : a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2) :18 – 31, 2004.

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, pages 132–146. Springer, 2008.

[BDDDD13] Nicolas Brunie, Florent De Dinechin, and Benoît De Dinechin. Conception d'une matrice reconfigurable pour coprocesseur fortement couplé. In *Symposium en Architectures nouvelles de machines*, France, January 2013.

[BDM02] L. Benini and G. De Micheli. Networks on chips : A new SoC paradigm. *Computer*, 35(1) :70–78, 2002.

[BDT13] Shuvra S Bhattacharyya, Ed F Deprettere, and Bart D Theelen. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems*, pages 905–944. Springer, 2013.

[BHLM94] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4 :155–182, 1994.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2) :118–129, 1991.

[BJT99] Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In *International Static Analysis Symposium*, pages 51–68. Springer, 1999.

[CA07] Stephen Craven and Peter Athanas. Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded systems*, 2007(1) :13–13, 2007.

[CAD+12] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE, 2012.

[CCL+10] Xiaowen Chen, Shuming Chen, Zhonghai Lu, Axel Jantsch, Bangjian Xu, and Heng Luo. Multi-fpga implementation of a network-on-chip based many-core architecture with fast barrier synchronization mechanism. In *NORCHIP, 2010*, pages 1–4. IEEE, 2010.

[CGMT09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *Design & Test of Computers, IEEE*, 26(4) :8–17, 2009.

[CM08] Ewerson Carvalho and Fernando Moraes. Congestion-aware task mapping in heterogeneous mpsocs. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4. IEEE, 2008.

[COM08] Chen-Ling Chou, Umit Y Ogras, and Radu Marculescu. Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10) :1866–1879, 2008.

[CPMB14] Francesco Conti, Chuck Pilkington, Andrea Marongiu, and Luca Benini. He-p2012 : architectural heterogeneity exploration on a scalable many-core platform. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 114–120. IEEE, 2014.

[CR85] Lori A. Clarke and Debra J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1) :15–35, 1985.

[CS14] James Coole and Greg Stitt. Fast, flexible high-level synthesis from opencl using reconfiguration contexts. *Micro, IEEE*, 34(1) :42–53, 2014.

[Cul86] David E Culler. Dataflow architectures. Technical report, DTIC Document, 1986.

[Dav99] Alan L Davis. A data flow evaluation system based on the concept of recursive locality. In *afips*, page 1079. IEEE, 1899.

[DD11] Florent De Dinechin. The arithmetic operators you will never see in a microprocessor. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 189–190. IEEE, 2011.

[dDAB+09] B.D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P.G. de Massas, F. Jacquet, S. Jones, N.M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013-09.

[DFBM14] Atef Dorai, Virginie Fresse, El-Bay Bourennane, and Abdellatif Mtibaa. A novel architecture for inter-fpga traffic collision management. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pages 487–495. IEEE, 2014.

[DG10] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing, January 19 2010. US Patent 7,650,331.

[DH14] Anna Derezinska and Konrad Halas. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*, pages 155–164. Springer, 2014.

[DHS12] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[DKW99] Oliver Diessel, David Kearney, and Grant Wigley. A web-based multiuser operating system for reconfigurable computing. In *International Parallel Processing Symposium*, pages 579–587. Springer, 1999.

[DPSV06] Douglas Densmore, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. A platform-based taxonomy for esl design. *IEEE Design & Test of Computers*, (5) :359–374, 2006.

[EDD+07] Samuel Evain, Rachid Dafali, Jean-Philippe Diguet, Yvan Eustache, and Emmanuel Juin. μspider cad tool : Case study of noc ip generation for fpga. In *Dasip07*, page x, 2007.

[FWX14] Xinjian Fan, Xuelin Wang, and Yongfei Xiao. A shape-based stereo matching algorithm for binocular vision. In *Security, Pattern*

Analysis, and Cybernetics (SPAC), 2014 International Conference on, pages 70–74. IEEE, 2014.

[GDGN03] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE, 2003.

[GHP+09] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10) :1517–1530, 2009.

[GKW85] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1) :34–52, 1985.

[GSM+14] Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. Real-time 3d reconstruction for fpgas : A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl sdk. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 326–329. IEEE, 2014.

[HLY+06] Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Hardware-software codesign of multimedia embedded systems : the PeaCE. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings*, pages 207–214, 2006.

[HM05] Jingcao Hu and Radu Marculescu. Energy-and performance-aware mapping for regular noc architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4) :551–562, 2005.

[HOJH06] Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, and Soonhoi Ha. Conversion of reference c code to dataflow model h.264 encoder case study. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

[HVG+07] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3) :50, 2007.

[I+88] Robert A Iannucci et al. *Two fundamental issues in multiprocessing*. Springer, 1988.

[IBS20] Ra Inta, David J. Bowman, and Susan M. Scott. The "chimera" : An off-the-shelf CPU/GPGPU/FPGA hybrid computing platform. *International Journal of Reconfigurable Computing*, 2012, 2012-03-20.

[JMBDM04] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Xpipescompiler : a tool for instantiating application specific networks on chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 884–889. IEEE, 2004.

[JMP+11] Jörn W Janneck, Ian D Miller, David B Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2) :241–249, 2011.

[Jo04] Jingcao Hu and R. others. Energy and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4) :551–562, 2005-04.

[K+14] Nachiket Kapre et al. Heterogeneous dataflow architectures for fpga-based sparse lu factorization. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. *In Information Processing*, 74 :471–475, 1974.

[KBA+15] Hana Krichene, Mouna Baklouti, Mohamed Abid, Philippe Marquet, and Jean-Luc Dekeyser. G-mpsoc : Generic massively parallel architecture on fpga. *WSEAS Transactions on circuits and systems*, 14, 2015.

[KBL+11] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. Distrm : distributed resource management for on-chip many-core systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 119–128. ACM, 2011.

[KDW10] Srinidhi Kestur, John D. Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 288–293. IEEE, 2010.

[KJS+02] Shashi Kumar, Axel Jantsch, J.-P. Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.

[KS05] Youngsoo Kim and Suleyman Sair. Designing real-time h. 264 decoders with dataflow architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 291–296. ACM, 2005.

[KTMD14] Cédric Killian, Camel Tanougast, Fabrice Monteiro, and Abbas Dandache. Smart reliable network-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(2) :242–255, 2014.

[LK03] Tang Lei and Shashi Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 180–187. IEEE, 2003.

[LM87] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, 1987.

[LRB11] S. M. Logesh, D. S. Ram, and M. C. Bhuvaneswari. Survey of high-level synthesis techniques for area, delay and power optimization. *International Journal of Computer Applications*, 32, 2011.

[LS97] C Lavarenne and Y Sorel. Modele unifié pour la conception conjointe logiciel-matériel a unified model for software-hardware co-design. *Traitement du signal*, 14(6) :569–578, 1997.

[LSV97] Edward A. Lee and Alberto Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 234–241. IEEE Computer Society, 1997.

[M+09] Radu Marculescu et al. Outstanding research problems in noc design : system, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1) :3–21, 2009.

[MCM+04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes : an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1) :69–93, 2004.

[MHB+14] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Erik Hochapfel, and Sandrine Vaton. Energy-efficient fpga implementation for binomial option pricing using opencl. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6. IEEE, 2014.

[Moo11] Chuck Moore. Data processing in exascale-class computer systems. In *The Salishan Conference on High Speed Computing*, 2011.

[MPCVG04] I. Miro-Panades, F. Clermidy, P. Vivet, and A. Greiner. Physical implementation of the DSPIN network-on-chip in the FAUST architecture. In *Second ACM/IEEE International Symposium on Networks-on-Chip, 2008. NoCS 2008*, pages 139–148, 2008-04.

[OHM05] Umit Y. Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in NoC design : a holistic perspective. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74. ACM, 2005.

[OMVEC13] Garrick Orchard, Jacob G Martin, R Jacob Vogelstein, and Ralph Etienne-Cummings. Fast neuromimetic object recognition using

fpga outperforms gpu implementations. *Neural Networks and Learning Systems, IEEE Transactions on*, 24(8) :1239–1252, 2013.

[PBS+03] Chandrakant D. Patel, Cullen E. Bash, Ratnesh Sharma, Monem Beitelmal, and Rich Friedrich. Smart cooling of data centers. In *ASME 2003 International Electronic Packaging Technical Conference and Exhibition*, pages 129–137. American Society of Mechanical Engineers, 2003.

[PCD15] Jean Perier, Wissem Chouchene, and Jean-Luc Dekeyser. Circuit merging versus dynamic partial reconfiguration - the homade implementation. *i-Manager's Journal on Embedded Systems*, 4(1) :14, 2015.

[PDM+15] Maxime Pelcat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, and Shuvra S. Bhattacharyya. Models of Architecture. Research Report PREESM/2015-12TR01, 2015, IETR/INSA Rennes ; Scuola Superiore Sant'Anna, Pisa ; Institut Pascal, Clermont Ferrand ; University of Maryland, College Park ; Tampere University of Technology, Tampere, December 2015.

[Pea20] Karl Pearson. Notes on the history of correlation. *Biometrika*, 13(1) :25–45, 1920.

[PFYDL15] Ke Pang, Virginie Fresse, Suying Yao, and Otavio Alcantara De Lima. Task mapping and mesh topology exploration for an fpga-based network on chip. *Microprocessors and Microsystems*, 39(3) :189–199, 2015.

[PGJ+05] Partha Pratim Pande, Cristian Grecu, Michael Jones, Andre Ivanov, and Resve Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8) :1025–1040, 2005.

[PGTM99] Matthew A Postiff, David A Greene, Gary S Tyson, and Trevor N Mudge. The limits of instruction level parallelism in spec95 applications. *SIGARCH Computer Architecture News*, 27(1) :31–34, 1999.

[PK04] Ruxandra Pop and Shashi Kumar. A survey of techniques for mapping and scheduling applications to network on chip systems. *School of Engineering, Jonkoping University, Research Report*, 4 :4, 2004.

[PTD13] Kenneth Pocek, Russell Tessier, and André DeHon. Birth and adolescence of reconfigurable computing : A survey of the first 20 years of field-programmable custom computing machines. In *Highlights of the First Twenty Years of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 3–19, 2013.

[QB15] Shaodong Qin and Mladen Berekovic. A comparison of high-level design tools for soc-fpga on disparity map calculation example. *arXiv preprint arXiv :1509.00036*, 2015.

[Red16] Red hat drives FPGAs, ARM servers | EE times, 2016.

[RRS11] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to recommender systems handbook*. Springer, 2011.

[SAK12] David Sheffield, Michael Anderson, and Kurt Keutzer. Automatic generation of application-specific accelerators for fpgas from python loop nests. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 567–570. IEEE, 2012.

[SC13] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1) :60–76, 2013.

[SEW+07] V. Soteriou, N. Eisley, H. Wang, B. Li, and L. S. Peh. Polaris : A system-level roadmapping toolchain for on-chip interconnection networks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(8) :855–868, 2007.

[SHNS86] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satoshi Sekiguchi. Evaluation of a prototype data flow processor of the sigma-1 for scientific computations. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 226–234. IEEE Computer Society Press, 1986.

[sim] Mathworks simulink : Simulation and model-based design. `http://fr.mathworks.com/products/simulink/`. Accessed : 2016-09-01.

[SLS10] Marta Stepniewska, Adam Luczak, and Jakub Siast. Network-on-multi-chip (nomc) for multi-fpga multimedia systems. In *Digital System Design : Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 475–481. IEEE, 2010.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422–433. IEEE, 2003.

[SV06] Balasubramanian Sethuraman and Ranga Vemuri. optimap : a tool for automated generation of noc architectures using multi-port routers for fpgas. In *Proceedings of the conference on Design, automation*

and test in Europe : Proceedings, pages 947–952. European Design and Automation Association, 2006.

[SWN02] W. Sun, M.J. Wirthlin, and S. Neuendorffer. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2) :254–265, 2007-02.

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing : architectures and design methods. *IEE Proceedings - Computers and Digital Techniques*, 152(2) :193, 2005.

[TFR11] Junyan Tan, Virginie Fresse, and Frederic Rousseau. From mono-fpga to multi-fpga emulation platform for noc performance evaluations. In *International Conference on Parallel Computing-ParaFPGA Symposium*, pages 7–pages, 2011.

[Tom67] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1) :25–33, 1967.

[Vee86] Arthur H Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18(4) :365–396, 1986.

[Viv] Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html. Accessed : 2016-09-01.

[VPCM09] Emmanuel Vaumorin, Maxime Palus, Fabien Clermidy, and Jérôme Martin. Spirit ip-xact controlled esl design tool applied to a network-on-chip platform. *Design and Reuse Industry Articles*, 2009.

[WBM06] E.W. Wachter, A. Biazi, and F.G. Moraes. HeMPS-s : A homogeneous NoC-based MPSoCs framework prototyped in FPGAs. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1 –8, 2011-06.

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, (5) :15–31, 2007.

[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. Baring it all to software : Raw machines. *Computer*, 30(9) :86–93, 1997.

[WYJL09] Xiaohang Wang, Mei Yang, Yingtao Jiang, and Peng Liu. Power-aware mapping for network-on-chip architectures under bandwidth and latency constraints. In *Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on*, pages 1–6. IEEE, 2009.

[ZSJ+09] Yan Zhang, Yasser H Shalabi, Rishabh Jain, Krishna K Nagar, and Jason D Bakos. Fpga vs. gpu for sparse matrix vector multiply. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 255–262. IEEE, 2009.

Annexes

Annexe A : Algorithme d'ordonnancement HOIMS

```
Require: TaskGraph, FPGAResources
Ensure: APCG, BestSchedule
 1: function HOIMS(FPGAModel, ModList, APCG) TaskMapCandidates ← CandidateModules(TaskGraph, Constraints)
 2: |  for TaskMapping, GlobalDII in ModuleSelections(TaskMapCandidates, Constraints) do
 3: |  |  if not CorrectMS(TaskMapping, TaskGraph) then
 4: |  |  |  continue
 5: |  |  CurSched ← EmptyScheduling(TaskMapping, GlobalDII)
 6: |  |  UnscheduleTasks(CurSched)
 7: |  |  TaskList ← SortTasksByPriority(TaskGraph)
 8: |  |  while TaskList is not empty do
 9: |  |  |  Task ← TaskList.pop(0)
10: |  |  |  (TSMin, TSMax) ← Task.Makespan(TaskMapping)
11: |  |  |  (TS,SelectedMod,TaskToReplace) ← DynamicAllocate(GlobalDII, Task, TSMin, TSMax, CurSched, TaskMapping, Constraints, TaskGraph)
12: |  |  |  if TaskToReplace then
13: |  |  |  |  TaskList.append(TaskToReplace)
14: |  |  |  |  TaskToReplace.Unschedule()
15: |  |  |  Sched ← Schedule(Task,TS,SelectedMod,CurSched)
16: |  |  if Latency(CurSched) < Latency(BestSched) then
17: |  |  |  BestSched ← CurSched
18: |  return APCG(BestSched, TaskGraph), BestSched
```

Algorithme 5 : NoC oriented HOIMS algorithm

Annexe B : Algorithme d'allocation dynamique

```
Require: GlobalDII, Task, TSMin , TSMax, CurrentSched, TaskMapping, FPGA-
    Model, TaskGraph
Ensure: BestComponent : Tuple of (TimeStep, Module, IsConflicted) values
 1: function DynamicAllocate(GlobalDII, Task, TSMin , TSMax, CurSched,
    TaskMapping, FPGAModel, TaskGraph)
 2:     CompModList ← CompatibleModules(Task, CurSched, TaskMapping)
 3:     if not WorthShare(Task) or CompModList is Empty then
 4:         RscExceeded, RscEstimated=AllocExceedRsc(FPGAModel, Task, Task-
    Mapping[Task], CurSched, TaskGraph)
 5:         if RscExceeded is False then
 6:             NewModule=TaskMapping[Task].Copy()
 7:             AddNewModule(GlobalDII, CurSched, NewModule)
 8:             return (TSMin, NewModule, False)
 9:     MaxSimCompC ← Null
10:     MaxSimCompNC ← Null
11:     for TS in range(TSMin, TSMax+1) do
12:         MaxSimComp ← SelectMaxSimComp(GlobalDII, TS, Task, CompMo-
    dList, CurSched)
13:         if not MaxSimComp.Conflicted then          ▷ composant sans conflit
14:             if MaxSimCompNC.Similarity < MaxSimComp.Similarity then
15:                 MaxSimCompNC = MaxSimComp
16:         else                                       ▷ composant avec conflit
17:             if MaxSimCompC.Similarity < MaxSimComp.Similarity then
18:                 MaxSimCompC = MaxSimComp
19:     if MaxSimCompNC is not Null then
20:         return MaxSimCompNC          ▷ Meilleur composant non conflictuel
21:     else if MaxSimCompC is not Null then              ▷ Partage conflictuel
22:         NewSim = Similarity(MaxSimCompC, CurrentSched)
23:         if MaxSimCompC.Similarity < NewSim then
24:             return MaxSimCompC                    ▷ désordonnance la tâche
25:     RscExceeded, RscEstimated=AllocExceedRsc(FPGAModel, Task, Task-
    Mapping[Task], CurrentSched, TaskGraph)
26:     if RscExceeded is True then       ▷ Exceed the FPGA available resources
27:         return MaxSimCompC                                ▷ force sharing
28:     else                                          ▷ Resources available
29:         NewModule=TaskMapping[Task].Copy()
30:         AddNewModule(GlobalDII, CurrentSched, NewModule)
31:         return NewComp(TSMin, NewModule)
```

Algorithme 6 : Fonction d'allocation dynamique

Annexe C : Algorithme d'authentification d'images multispectrales

```
Require: OriginalRegs, TestedRegions, TabRefX, TabRefY, TabRefZ : float vectors
Ensure: Authentified : boolean
function AuthentifyMS(OriginalRegs, TestedRegions, TabRefX, TabRefY, TabRefZ)
    OriRegionMeans ← list()
    TestedRegionMeans ← list()
    for OriginalR, TestedR in zip(OriginalRegs, TestedRegs) do
        OriRegionMeans.append(RegionMean(OriginalR))
        TestedRegionMeans.append(RegionMean(TestedR))
    ▷ Calcul des valeurs CIE XYZ
    X1 ← AxisXYZ(OriRegionMeans,TabRefX)
    Y1 ← AxisXYZ(OriRegionMeans,TabRefY)
    Z1 ← AxisXYZ(OriRegionMeans,TabRefZ)
    X2 ← AxisXYZ(TestedRegionMeans,TabRefX)
    Y2 ← AxisXYZ(TestedRegionMeans,TabRefY)
    Z2 ← AxisXYZ(TestedRegionMeans,TabRefZ)
    ▷ Calcul des valeurs RGB
    R1 ← Red(X1,Y1,Z1)
    R2 ← Red(X2,Y2,Z2)
    G1 ← Green(X1,Y1,Z1)
    G2 ← Green(X2,Y2,Z2)
    B1 ← Blue(X1,Y1,Z1)
    B2 ← Blue(X2,Y2,Z2)
    ▷ Distance couleur
    dis_RGB ← DistanceRGB(R1,R2,G1,G2,B1,B2)
    if dis_RGB > P1 then
        return False
    ▷ Distance multispectale
    dis_WRMS ← DistanceWRMS(OriRegionMeans,TestedRegionMeans)
    if dis_WRMS > P2 then
        return False
    return True
```

Algorithme 7 : Fonction d'authentification d'images multispectrales