THÈSE DE DOCTORAT

Présentée devant

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

Pour l'obtention

DU GRADE DE DOCTEUR

Par

MICKAËL DARDAILLON

COMPILATION D'APPLICATIONS FLOT DE DONNÉES PARAMÉTRIQUES POUR MPSOC DÉDIÉS À LA RADIO LOGICIELLE

Encadré par Tanguy RISSET et Kevin MARQUET
Équipe Inria SOCRATE, Laboratoire CITI, INSA Lyon
École Doctorale Informatique et Mathématiques
Spécialité Informatique

En partenariat avec
Jérôme MARTIN, CEA LETI
Henri-Pierre CHARLES, CEA LIST

Soutenue le 19 Novembre 2014

Jury

Albert COHEN	Directeur de recherche, *Inria*	Président du jury
Renaud PACALET	Directeur d'études, *Télécom ParisTech*	Rapporteur
Frédéric ROUSSEAU	Professeur des universités, *Polytech Grenoble*	Rapporteur
Alain GIRAULT	Directeur de recherche, *Inria*	Examinateur
Christophe MOY	Professeur à Supélec, *Supélec Rennes*	Examinateur
Tanguy RISSET	Professeur des universités, *INSA Lyon*	Directeur de thèse
Kevin MARQUET	Maître de conférences, *INSA Lyon*	Co-directeur de thèse
Jérôme MARTIN	Ingénieur de recherche, *CEA LETI*	Encadrant de thèse
Henri-Pierre CHARLES	Directeur de recherche, *CEA LIST*	Encadrant de thèse

À mon père

Remerciements

Ce manuscrit présente le travail de mes trois années de thèse, qui a rassemblé de nombreux acteurs à qui je souhaite rendre hommage. Je remercie en premier lieu Renaud Pacalet et Frédéric Rousseau d'avoir accepté d'en être les rapporteurs, ainsi que Albert Cohen, Alain Girault et Christophe Moy d'avoir accepté de l'examiner.

J'adresse mes plus sincères remerciements à Tanguy Risset pour m'avoir offert l'opportunité d'effectuer cette thèse et d'en avoir pris la direction. Il a su par ses connaissances, ses conseils et une grande patience me guider dans la réalisation de ces travaux. Je remercie également Kevin Marquet, en tant que co-directeur de thèse, d'avoir partagé son énergie et son expérience qui m'ont permis d'aborder les points difficiles de ce travail.

Je tiens à remercier tout particulièrement Jérôme Martin pour son encadrement lors de mon séjour au CEA. J'ai eu le plaisir de bénéficier de son expertise technique et de sa vision du domaine, le tout avec une grande disponibilité. Je remercie aussi Henri-Pierre Charles pour ses conseils et ses remarques dans le suivi de cette thèse. Je remercie également Frédéric Heitzmann pour son aide, ainsi que Laurent Bertholle pour son apport dans le développement du compilateur.

Ma reconnaissance va également à Jean-Marie Gorce et Fabien Clermidy, directeurs du laboratoire CITI et LISAN durant cette thèse, pour m'avoir accueilli dans leurs laboratoires respectifs et permis de découvrir le monde de la recherche. Le déroulement d'une thèse bi-localisée implique de nombreux déplacements, et je remercie Gaëlle Tworkowski pour son assistance dans la prise en charge de toutes ces missions.

Ce travail de thèse s'est déroulé dans une ambiance enrichissante, et je tiens à remercier tous mes collègues tant à Lyon qu'à Grenoble avec qui j'ai partagé ces trois ans. Je remercie en particulier Nicolas Stouls avec qui j'ai pu m'impliquer à la fois dans l'enseignement et dans l'animation de la vie du laboratoire.

Enfin, je voudrai remercier toute ma famille et mes amis pour m'avoir accompagné durant ces trois années de thèse, et plus particulièrement mon père qui m'a offert la possibilité de suivre ma propre voie, et à qui je dédie cette thèse.

Résumé

Résumé

Le développement de la radio logicielle fait suite à l'évolution rapide du domaine des télécommunications. Les besoins en performance et en dynamicité ont donné naissance à des MPSoC dédiés à la radio logicielle. La spécialisation de ces MPSoC rend cependant leur programmation et leur vérification complexes. Des travaux proposent d'atténuer cette complexité par l'utilisation de paradigmes tels que le modèle de calcul flot de données. Parallèlement, le besoin de modèles flexibles et vérifiables a mené au développement de nouveaux modèles flot de données paramétriques.

Dans cette thèse, j'étudie la compilation d'applications utilisant un modèle de calcul flot de données paramétrique et ciblant des plateformes de radio logicielle. Après un état de l'art du matériel et logiciel du domaine, je propose un raffinement de l'ordonnancement flot de données, et présente son application à la vérification des tailles mémoires. Ensuite, j'introduis un nouveau format de haut niveau pour définir le graphe et les acteurs flot de données, ainsi que le flot de compilation associé. J'applique ces concepts à la génération de code optimisé pour la plateforme de radio logicielle Magali. La compilation de parties du protocole LTE permet d'évaluer les performances du flot de compilation proposé.

Abstract

The emergence of software-defined radio follows the rapidly evolving telecommunication domain. The requirements in both performance and dynamicity has engendered software-defined-radio-dedicated MPSoCs. Specialization of these MPSoCs make them difficult to program and verify. Dataflow models of computation have been suggested as a way to mitigate this complexity. Moreover, the need for flexible yet verifiable models has led to the development of new parametric dataflow models.

In this thesis, I study the compilation of parametric dataflow applications targeting software-defined-radio platforms. After a hardware and software state of the art in this field, I propose a new refinement of dataflow scheduling, and outline its application to buffer size's verification. Then, I introduce a new high-level format to define dataflow actors and graph, with the associated compilation flow. I apply these concepts to optimised code generation for the Magali software-defined-radio platform. Compilation of parts of the LTE protocol are used to evaluate the performances of the proposed compilation flow.

Table des matières

Table des figures

Liste des tableaux

Glossaire

3GPP *3rd Generation Partnership Project*, organisme visant à définir les normes pour les réseaux mobiles de troisième et quatrième génération. (p. 4, 58, 79, 92)

ADC Convertisseur analogique numérique, convertisseur qui transforme un signal analogique continu en un signal numérique discret. (p. 9, 10)

ARM *Advanced RISC Machines*, société spécialisée dans le développement d'architectures de processeurs sous la forme de propriétés intellectuelles. (p. 11–14, 52, 60, 71, 74, 83, 87, 94)

ASIC *Application Specific Integrated Circuit*, circuit intégré dédié à une application spécifique. (p. 9)

ASIP *Application Specific Instruction set Processor*, processeur dont le jeu d'instruction est dédié à une application. (p. 13)

BDF *Boolean DataFlow*, modèle de calcul flot de données. (p. 25)

BPDF *Boolean Parametric DataFlow*, modèle de calcul flot de données. (p. 27, 32, 98)

CAL *CAL Actor Language*, langage flot de données. (p. 19, 30, 45)

Clang *Front end* LLVM pour les langages C, C++ et Objective-C. (p. 64, 93)

CORBA *Common Object Request Broker Architecture*, architecture logicielle standard pour faciliter la communication de systèmes distribués. (p. 17)

CPU *Central Processing Unit*, processeur généraliste. (p. 10, 11, 15, 85)

CSDF *Cyclo-Static DataFlow*, modèle de calcul flot de données. (p. 18, 24, 45)

DDF *Dynamic DataFlow*, modèle de calcul flot de données. (p. 24, 25)

DMA *Direct Memory Access*, contrôleur prenant en charge les transferts mémoire. (p. 14, 60, 80, 94)

DSP *Digital Signal Processor*, processeur spécialisé pour le traitement du signal numérique. (p. 9–13, 60, 71)

FFT *Fast Fourier Transform*, algorithme de calcul de la transformée de fourier discrète largement répandu en télécommunications. (p. 12, 49, 55, 60, 65, 67, 69, 70)

FIFO *First In First Out*, mémoire fonctionnant comme une file d'attente, la première donnée arrivée étant la première lue. (p. 22, 23, 29, 31, 36–38, 60, 61, 64, 66, 67, 69, 72)

FPGA *Field Programmable Gate Array*, circuit électronique dont l'architecture est reprogrammable. (p. 10, 11, 14–17)

GALS *Globally Asynchronous, Locally Synchronous*, modèle d'architecture connectant des unités locales synchrones avec une architecture asynchrone. (p. 59)

GNU Radio Environnement ouvert de développement de radio logicielle. (p. 10, 11, 15, 17, 18, 96, 97)

GPU *Graphics Processing Unit*, circuit dédié aux traitements graphiques. (p. 11, 18)

H.264 Norme de compression vidéo. (p. 39)

JTRS *Joint Tactical Radio System*, projet de radio logicielle du département de défense américain. (p. 17)

KPN *Kahn Process Network*, modèle de calcul distribué. (p. 19, 62)

LINK *LINK layer*, couche en charge de l'encodage, du décodage et de l'organisation logique des trames réseaux. (p. 11)

LLVM Infrastructure de compilation ouverte. (p. 45, 51–54, 56, 64, 71, 74, 75, 93, 94, 97)

LTE *Long Term Evolution*, norme de réseau mobile de quatrième génération défini par l'organisme 3GPP. (p. 4, 5, 12–14, 18–20, 38, 58, 59, 62, 75, 76, 78–83, 88, 89, 92, 93, 96)

MAC *Medium Access Control*, couche de contrôle d'accès au réseau. (p. 11)

MCDF *Mode Controlled DataFlow*, modèle de calcul flot de données. (p. 24)

MIMO *Multiple Input Multiple Output*, utilisation de plusieurs antennes en émission et réception. (p. 13–15, 17, 43, 52, 58–60, 65, 79)

model checking Technique de vérification automatique de systèmes représentés sous la forme de graphes. (p. 29, 36, 38, 39, 67, 84, 89)

MPSoC *Multiprocessor System-on-Chip*, système sur puce multiprocesseur. (p. 16, 46)

NoC *Network on Chip*, réseau intégré au circuit électronique pour la communication entre ces unités. (p. 13, 14, 59)

OFDM *Orthogonal Frequency-Division Multiplexing*. (p. 12–15, 17, 58, 60)

ORCC *Open RVC-CAL Compiler*, compilateur du langage flot de données CAL ouvert. (p. 45, 98)

OSI *Open Systems Interconnection*, modèle de communications entre ordinateurs proposé par l'ISO. (p. 9)

PaDaF *Parametric Dataflow Format*, format de langage flot de données paramétrique. (p. 5, 42, 43, 45, 46, 48, 49, 52, 53, 55, 56, 58, 64, 66, 71, 78, 80, 83, 88)

PROMELA PROcess MEta LAngage, Langage de modélisation pour la vérification utilisé dans SPIN. (p. 29, 36, 37, 39, 67, 69, 84, 95)

PSDF *Parametrized Synchronous DataFlow*, modèle de calcul flot de données. (p. 25, 31)

QEMU Machine virtuelle émulant d'autres processeurs. (p. 87)

RISC *Reduced Instruction Set Computer*, processeur au jeu d'instruction réduit. (p. 12, 13)

SADF *Scenario Aware DataFlow*, modèle de calcul flot de données. (p. 24)

SAS *Single Apparence Schedule*, ordonnancement où chacun des acteurs n'apparait qu'une seule fois. (p. 28–30, 66)

SCA *Software Communication Architecture* (SCA), système applicatif pour la radio logicielle lancé par le département de la défense américain. (p. 17)

SDF *Synchronous DataFlow*, modèle de calcul flot de données. (p. 18–20, 23–25, 28, 30, 38, 44)

SIMD *Single Instruction Multiple Data*, processeur qui exécute la même instruction sur des données multiples. (p. 11, 12, 19)

SPDF *Schedulable Parametric DataFlow*, modèle de calcul flot de données. (p. 23, 25–27, 30, 31, 38, 42, 43, 45, 64, 67, 93, 98)

SPIN Outil permettant la vérification de système selon la technique du model checking. (p. 29, 36, 39, 67, 84, 95)

SSA *Static Single Assignement*, Représentation à assignation unique utilisé en compilation. (p. 52, 54)

SystemC Ensemble de classes C++ permettant la modélisation d'une architecture mixte matérielle et logicielle, en vu de sa simulation. (p. 46, 50, 52, 87, 94)

TSDF *Timed Synchronous DataFlow*, modèle de calcul flot de données. (p. 19)

UMTS *Universal Mobile Telecommunications System*, technologie pour la téléphonie mobile 3G. (p. 11)

USRP *Universal Software Radio Peripheral*, plateforme matérielle de radio logicielle. (p. 10, 17)

VHDL *VHSIC Hardware Description Language*, langage de description matérielle. (p. 11, 14, 15, 49, 52)

Viterbi Algorithme de correction d'erreurs utilisé en télécommunications. (p. 12–14)

VLIW *Very Large Instruction Word*, processeur à parallélisme explicite dans le jeu d'instruction. (p. 11, 14, 19, 60, 71)

WCDMA *Wideband Code Divison Multiple Access*, technologie pour la téléphonie mobile 3G. (p. 11, 12, 15)

WiMAX *Worldwide Interoperability for Microwave Access*, standard de communication sans fil utilisé pour l'accès à internet haut débit. (p. 12, 15)

1 Introduction

During a conference talk question and answer period, someone raised his hand and asked "How should we program a Cell processor ?" The speaker looked a bit taken aback and replied "Any way you want !"

— Cell processor architect

Je présente dans ce manuscrit un bilan du travail réalisé durant ma thèse sur la *compilation d'applications flot de données paramétriques pour MPSoC dédiés à la radio logicielle*. Cette thèse est le fruit d'une collaboration entre l'équipe Inria Socrate du laboratoire CITI (Centre d'Innovation en Télécommunications et Intégration de services) de l'INSA de Lyon et le laboratoire LISAN (Laboratoire d'Intégration Silicium des Architectures Numériques) de l'institut CEA LETI de Grenoble, financée par la Région Rhône Alpes (ADR 11 01302401). Cette collaboration a pour objectif de rassembler les compétences matérielles et logicielles de ces deux laboratoires, en particulier dans le domaine du traitement du signal et des télécommunications, pour améliorer la programmation des futures plateformes de radio logicielle.

Cette thèse traite de l'application du problème de la programmation des plateformes parallèles au domaine de la radio logicielle. Je situerai d'abord le problème général au travers de l'évolution des architectures matérielles, puis je présenterai les modèles de calcul flot de données proposés comme une solution possible à ce problème. J'exposerai ensuite le contexte particulier de cette thèse, la radio logicielle, et les contributions de ce travail. Je finirai par détailler le plan de ce manuscrit en relation avec le contexte exposé.

1.1 Programmation des plateformes parallèles

La programmation des plateformes parallèles, ou comment découper un programme de grande taille en plusieurs programmes plus petits pouvant être exécutés en concurrence, est un problème qui anime la communauté scientifique depuis plus d'une quarantaine d'années. Ce problème est resté un temps limité au domaine du calcul haute performance. L'évolution des plateformes matérielles, en particulier l'introduction du multicœur sur les plateformes Intel en 2005, a rendu ce problème omniprésent, depuis les serveurs jusqu'aux systèmes embarqués. Pour comprendre l'évolution des plateformes matérielles, je propose de regarder les différentes limitations qu'elles rencontrent, ou murs, tels que décris par Patterson *et al.* [Asanovic 06].

Power Wall La première limitation est connue sous le nom de mur de l'énergie. Elle vient de la limite à l'énergie consommable par une plateforme matérielle, soit par sa batterie, soit par sa dissipation d'énergie. L'énergie consommée est directement proportionnelle à la fréquence du processeur ; elle limite donc la fréquence de fonctionnement. L'évolution technologique des transistors permet d'intégrer à chaque génération plus de transistors sur la même puce. La consommation de ces transistors ne réduit pas autant que leur augmentation en nombre ; tous les transistors d'une plateforme matérielle ne peuvent donc plus être alimentés simultanément. Cette limitation peut être contournée par plusieurs méthodes :

— La fréquence de processeur variable pour adapter la consommation en fonction de la charge de travail ;
— L'architecture multicœurs pour varier le nombre de cœurs de calcul en fonction de la charge de travail ;
— L'activation de circuits dédiés uniquement pour des opérations spécifiques, par exemple pour des primitives cryptographiques. Cette dernière méthode est aussi connue sous le nom de *dark silicon* [Esmaeilzadeh 11].

Memory Wall La seconde limitation est connue sous le nom de mur de la mémoire. Elle est dûe au coût beaucoup plus important d'une opération d'accès mémoire par rapport au coût d'une opération de calcul. Cette limitation est exacerbée par les architectures multicœurs, dont les différents cœurs sont en concurrence pour accéder à la mémoire. Pour pallier à cette limitation, les architectures matérielles ont connu plusieurs évolutions :

— La multiplication des niveaux de cache pour masquer la latence de la hiérarchie mémoire ;
— Les architectures à mémoire non homogène ou distribuée, dans laquelle les processeurs ne partagent plus une mémoire physique unique, avec un temps d'accès uniforme ;
— Les architectures sans gestion matérielle de la cohérence mémoire. La cohérence mémoire est conservée dans les approches précédentes pour masquer la complexité de la hiérarchie mémoire au logiciel. Cette cohérence est cependant de plus en plus complexe à maintenir avec l'augmentation du nombre de processeurs. Ces nouvelles architectures proposent de reporter cette complexité depuis le matériel vers le logiciel.

ILP Wall La troisième limitation est connue sous le nom de mur du parallélisme d'instruction. Extraire du parallélisme d'instruction offre un gain de performance limité, que ce soit par analyse lors de la compilation ou par l'ajout de matériel dédié dans le processeur. Ce mur limite le gain de performances par l'utilisation d'un processeur plus complexe. Pour exploiter

d'autres sources de parallélisme, les architectures matérielles ont évoluées vers le multicœurs. Elles tirent ainsi parti du parallélisme de données ou de tâches, car plusieurs cœurs simples sont plus efficaces en temps et en énergie qu'un cœur complexe. Ce parallélisme est aussi limité, cette limite est modélisée la loi d'Amdahl [Amdahl 67]. Pour remédier à ce problème, certaines architectures utilisent des cœurs hétérogènes. Les processeurs simples exécutent le code parallèle pour un coût énergétique limité, alors que les processeurs plus complexes sont en charge d'accélérer l'exécution de code séquentiel.

Face à toutes ces limitations, la conception de nouvelles architectures est un défi qui n'a pas de réponse unique. Les plateformes matérielles actuelles utilisent des architectures multicœurs, hétérogènes, dédiées à un domaine d'application spécifique, ou encore à mémoire distribuée non cohérente. Le second défi est alors pour le développeur d'arriver à tirer parti de toutes ces spécificités. La programmation de plateformes matérielles se fait à partir d'un modèle de celle-ci pour arriver à s'abstraire de la plateforme et de sa complexité. Cette abstraction, si elle est cohérente avec la plateforme réelle, permet de programmer cette plateforme de manière efficace. Elle rend de plus le programme portable vers d'autres plateformes correspondants à cette même abstraction.

L'abstraction utilisée durant des années pour programmer ces plateformes matérielles était un processeur unique et une mémoire unifiée. Le problème posé par l'évolution des plateformes matérielles est que cette abstraction n'est plus cohérente avec les plateformes actuelles. De plus, il n'existe pas à l'heure actuelle d'abstraction satisfaisante pour l'ensemble des plateformes matérielles. Face à cette problématique, de nouvelles approches proposent de représenter les applications à un plus haut niveau d'abstraction. Le compilateur prend la place du développeur ; il est en charge de spécialiser les applications pour différents types de plateformes, pour tirer parti de leurs spécificités. Dans cette thèse, je me concentre sur une de ces abstractions : la famille des modèles de calcul *flot de données*.

1.2 Modèles de calcul flot de données

Les modèles de calculs flot de données visent à représenter l'application à un haut niveau d'abstraction, indépendamment de la plateforme. Une application est représentée comme une suite de tâches, appelés acteurs, qui effectuent des traitements sur des données en entrée, pour produire de nouvelles données en sortie. Les acteurs communiquent entre eux de manière explicite au travers des données. Ces flot de données exposent les dépendances entre les acteurs. De cette manière, les modèles flots de données présentent le parallélisme de tâches : chaque acteur indépendant peut être exécuté de manière concurrente, et le *pipeline* de traitement : les traitements consécutifs peuvent être synchronisés par les données. Le traitement d'un acteur est représenté sous la forme d'un code séquentiel, dont le compilateur extrait le parallélisme d'instruction. Ces modèles sont présentés en détails dans le Chapitre 3.

La question principale qui nous intéresse dans cette thèse est le lien entre le modèle de calcul flot de données et la plateforme matérielle qui exécute une application suivant ce modèle de calcul. Le niveau d'abstraction rend possible de nouvelles analyses pour la vérification de l'application, en particulier l'absence d'interblocage dû aux communications. Le compilateur est alors responsable du placement et de l'ordonnancement correct, c'est à dire sans interblocage, et efficace des acteurs sur les unités de calcul. Il est aussi en charge de la spécialisation du code pour une plateforme spécifique, et de la génération de code qui utilise les mécanismes de cette plateforme. Je propose dans ce travail de contribuer à la résolution de ce problème dans un contexte matériel particulier, la radio logicielle.

1.3 Radio logicielle

Le décodage de la couche physique d'un protocole de télécommunication est, pour les systèmes classiques, pris en charge par une implémentation matérielle du protocole. La radio logicielle, telle que proposé par Mitola [Mitola III 92], remplace ce matériel par une implémentation logicielle sur une plateforme programmable. Ce choix d'implémentation vise à apporter les bénéfices du logiciel aux protocoles radios : réduction du temps de développement pour un nouveau protocole ; protocole dynamique et reconfigurable ; possibilités d'évolutions du protocole ; réutilisation du matériel pour plusieurs protocoles.

La radio logicielle est utilisée d'une part pour l'expérimentation de nouveaux protocoles grâce à son implémentation rapide, et d'autre part pour l'implémentations de protocoles existants, par exemple les téléphones mobiles de quatrième génération. Ces protocoles posent des contraintes fortes quant au débit de données à traiter, au temps de traitement, au dynamisme de l'application, ainsi qu'à la consommation énergétique dans le domaine de l'embarqué. Ces contraintes ont donné lieu à des plateformes dédiées au domaine de la radio logicielle. Leurs architectures complexes peuvent intégrer des unités de calcul hétérogènes, de la mémoire distribuée, et des unités de calcul dédiées. Cette complexité rend ces plateformes difficile à programmer sans une abstraction adaptée. Les modèles de programmation basés sur le modèle de calcul flot de données sont présentés comme une solution possible pour la radio logicielle, adaptée à plusieurs égards. L'abstraction proposée par les modèles flot de données, sous la forme de traitements sur des données, est proche du traitement du signal utilisé en radio logicielle. Le décodage d'un protocole radio implique une grande partie de manipulation de données, manipulation explicite dans les modèles flot de données.

La question qui se pose de nouveau est comment faire le lien entre un protocole radio encodé sous la forme d'une application flot de données et une plateforme matérielle de radio logicielle ? De nombreux travaux ont déjà étudié ce problème [Mrabti 09, Ben Abdallah 10, Moreira 12, Castrillon 13, Arnold 14]. Une des principales limites de ces travaux est l'utilisation de modèles de calcul flot de données statiques, qui empêchent la représentation de protocoles de télécommunications dynamiques. En effet, les développements récents en télécommunications vont en direction d'une *radio cognitive*, c.-à-d. une radio intelligente capable de s'adapter à son environnement dynamiquement. Les modèles de calcul flot de données dynamiques explicitent les parties dynamiques du protocole ; ce dynamisme peut alors être analysé et pris en charge de manière efficace par le compilateur puis la plateforme matérielle. Une autre limitation pour l'expérimentation sur la radio logicielle est le coût de développement d'un compilateur flot de données et de la génération de code pour une plateforme matérielle. Cette thèse contribue aux travaux existants par plusieurs aspects :

- L'utilisation d'un modèle de calcul flot de données paramétrique pour représenter le dynamisme de l'application ;
- Un raffinement de l'ordonnancement dans le modèle flot de données : le *micro-ordonnancement* ;
- Une nouvelle représentation de haut niveau des applications flot de données, basée sur le langage C++ ;
- L'intégration de cette représentation dans un compilateur existant ;
- Le développement d'un compilateur flot de données paramétrique pour la plateforme de radio logicielle Magali du CEA LETI ;
- L'implémentation de parties du protocole de télécommunications 3GPP-LTE pour valider mon approche.

1.4 Plan du manuscript

Ce manuscrit de thèse se découpe en 5 chapitres principaux.

Le Chapitre 2 rassemble l'état de l'art de la radio logicielle. Je présenterai d'abord les différentes plateformes matérielles pour la radio logicielle et je proposerai une séparation en plusieurs catégories. J'introduirai ensuite les environnements de programmation pour la radio logicielle. Cet état de l'art se concentrera sur le modèle de programmation flot de données.

Le Chapitre 3 est une étude de l'ordonnancement de graphes flot de données. Je détaillerai ici les modèles de calcul flot de données, avec un intérêt particulier pour le modèle de calcul flot de données paramétrique SPDF utilisé dans la suite de la thèse. J'introduirai ensuite des notions d'ordonnancement sur des graphes flot de données paramétriques, puis je proposerai le micro-ordonnancement comme raffinement flot de données pour l'analyse et l'ordonnancement. En particulier, je présenterai une nouvelle vérification des tailles mémoires basée sur le micro-ordonnancement.

Le Chapitre 4 présentera la première partie du flot de compilation développé durant cette thèse, depuis la représentation de haut niveau jusqu'à la représentation intermédiaire. Je proposerai dans ce chapitre une nouvelle représentation de haut niveau basée sur le langage C++, PaDaF (*Parametric Dataflow Format*). Je présenterai ensuite la compilation de cette représentation de haut niveau. En particulier, je détaillerai son intégration dans le compilateur LLVM existant. Cette intégration implique l'exécution d'une partie du code durant le processus de compilation, expliquée dans ce chapitre.

Le Chapitre 5 portera sur la seconde partie du flot de compilation, depuis la représentation intermédiaire vers la plateforme Magali. Je détaillerai en premier lieu la plateforme Magali utilisée comme cible du compilateur afin de mieux comprendre les enjeux du chapitre. Je présenterai ensuite la vérification et l'optimisation de haut niveau, basées sur la représentation intermédiaire. Cette étape comprend la vérification, le placement et l'ordonnancement du graphe flot de données. La seconde partie du chapitre détaillera la génération de code bas niveau pour la plateforme Magali. Elle présentera la mise en œuvre de la vérification mémoire pour cette plateforme. Elle passera en revue la génération de code pour les différentes unités de calcul de Magali, les communications et le modèle de contrôle distribué.

Le Chapitre 6 présentera les expérimentations sur la plateforme Magali. Je détaillerai en premier lieu le protocole de télécommunications utilisé comme référence pour l'évaluation de cette plateforme, le LTE, et les applications de tests extraites de ce protocole. J'évoquerai ensuite les gains de temps de développement par l'utilisation du format PaDaF, ainsi que les temps de compilation et de vérification des applications. En prélude des performances des applications, j'analyserai les performances attendues en fonction des différents modèles de contrôle possibles sur la plateforme Magali. À la lumière de ces analyses, je présenterai enfin les performances des applications compilées pour la plateforme Magali. Ces résultats démontreront la viabilité de l'approche proposée dans cette thèse.

Le Chapitre 7 conclura cette thèse. Il synthétisera le travail réalisé durant ces 3 années de manière chronologique. La synthèse sera suivie d'une discussion sur les principales contributions présentées. Enfin j'ouvrirai des perspectives sur les travaux futurs, en particulier l'étude des manipulations de données, avec plusieurs formats envisageables pour les représenter, ainsi que les analyses et les optimisations qui pourraient en tirer parti.

2 État de l'art

The wonderful thing about standards is that there are so many of them to choose from.

— Grace Hopper, *The Unix haters handbook*

Sommaire du chapitre

2.1 Introduction

Dans ce premier chapitre, je vais décrire l'état de l'art dans le domaine de la radio logicielle. Cet état de l'art se découpe en deux parties : matériel et logiciel.

Dans la Section 2.2, je présenterai les différentes plateformes matérielles utilisées en radio logicielle. Pour cela, j'introduirai l'architecture générale d'une radio logicielle et je situerai la partie étudiée dans cette thèse. Je proposerai ensuite une classification des différentes approches pour la radio logicielle, avec une présentation de plusieurs plateformes matérielles correspondant à chacune de ces approches.

Dans la Section 2.3, je présenterai les différents modèles de programmation pour les plateformes de radio logicielle. Cet état de l'art exposera tout d'abord les principaux environnements de programmation du domaine, puis une étude plus détaillée des modèles de calcul, depuis des approches conservatrices vers de nouveaux modèles basés sur le paradigme flot de données.

Chacune de ces sections sera conclue par une discussion permettant d'illustrer les défis posés par ces approches, et permettra de situer le cadre de mon travail.

2.2 Plateformes matérielles

2.2.1 Rappel sur l'architecture d'un émetteur/récepteur radio

Les différents composants d'un émetteur/récepteur radio sont illustrés sur la Figure 2.1. Je les décris depuis l'antenne, en charge de l'acquisition du signal, jusqu'au traitement en bande de base, sur lequel mon travail se concentre.

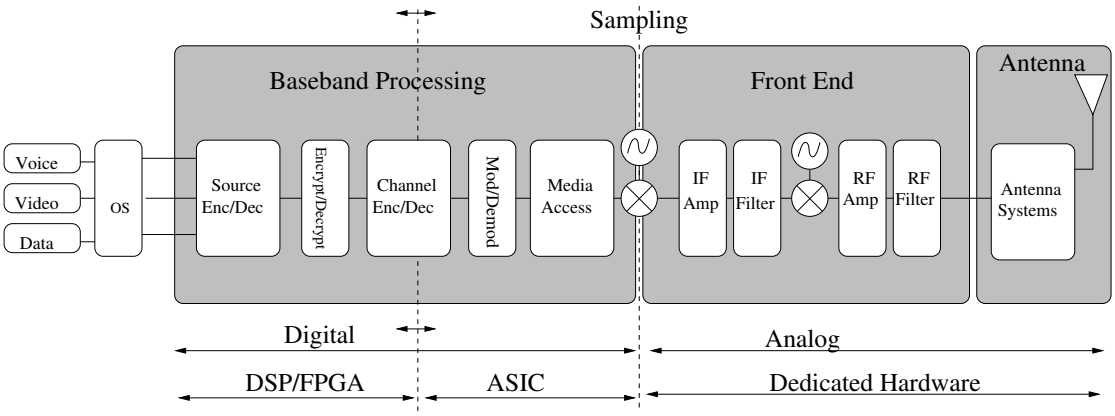


FIGURE 2.1 – Schéma bloc d'une radio logicielle, illustrant la séparation entre la partie logicielle et la partie matérielle, ainsi que la partie programmable, configurable et fixée dans le matériel.

Antenna L'*antenne* sert d'interface entre le signal sur la couche physique, l'onde électromagnétique, et le signal traité par le *front end*, un signal électrique analogique. Cette antenne peut être sélective en fréquence, permettant de filtrer une bande de fréquence spécifique, ou au contraire large bande. Une antenne sélective fixe le filtrage dans le matériel physique ; dans le cas contraire, ce filtrage est retardé à une phase ultérieure.

Front End Le *front end* est en charge de la mise en forme du signal analogique avant son passage en numérique lors de l'échantillonnage (*Sampling* sur la Figure 2.1). Afin de convertir le signal reçu dans une fréquence exploitable, le principe du récepteur *superhétérodyne* est utilisé. Le signal radio reçu à une fréquence F_r est multiplié par un oscillateur local à la fréquence F_{ol}. Le résultat de ce produit après filtrage est un signal à la fréquence intermédiaire $F_i = F_r - F_{ol}$. La fréquence de l'oscillateur local F_{ol} étant réglable, nous pouvons ramener un signal radio ayant une fréquence F_r variable à une fréquence intermédiaire F_i fixe pour le filtrer, l'amplifier et l'échantillonner.

De manière générale, plus un traitement est retardé dans la chaîne, plus il est configurable. Une approche radicale envisageable est d'échantillonner l'ensemble de la bande de fréquence, ce qui reporte le filtrage à la partie numérique programmable. Cette approche pose 2 défis : d'une part technologique et d'autre part en puissance de calcul.

Sur le plan technologique, pour un signal d'intérêt se trouvant à 4,9 GHz, le théorème de Nyquist-Shannon exige un échantillonnage à au moins deux fois cette fréquence, soit une fréquence d'échantillonnage à 10 GHz. De plus, l'ensemble des signaux de la bande de fréquence échantillonnée peuvent avoir des amplitudes très variables, ce qui implique un échantillonnage sur un grand nombre de bits. En termes de puissance de calcul, le flux de données devient alors colossal. En supposant que l'on possède un ADC (Convertisseur Analogique Numérique) ayant une fréquence d'échantillonnage de 10 GHz ainsi qu'une dynamique de 16 bits, le signal à traiter numériquement possède un débit de 160 Gbits/s ! À titre de comparaison, la capacité de traitement actuel d'un *front end* radio embarqué dans un téléphone est inférieur au Gbit/s.

Des domaines spécifiques tels que la radioastronomie utilisent d'ores et déjà cette approche. Son coût reste cependant prohibitif pour la plupart des applications. De plus, malgré l'évolution technologique qui pourrait baisser ce coût, les besoins actuels pour décoder des signaux plus complexes ainsi que réduire la consommation énergétique vont limiter cette approche à des cas particuliers dans les années à venir. Nous pouvons donc envisager un débit maximum de réception de l'ordre du Gbit/s en entrée du traitement en bande de base.

Baseband processing Le *Traitement en bande de base* est en charge du traitement numérique de la *couche physique* selon le modèle OSI (*Open Systems Interconnection*). Ce traitement est découpé en une partie configurable et une partie programmable (ASIC/DSP sur la Figure 2.1). Plus cette partie programmable est grande, plus la radio est dite *logicielle*.

Afin d'encourager une définition commune du terme radio logicielle, le *Wireless Inovation Forum* (anciennement *SDR Forum*) propose de distinguer 5 niveaux [Glossner 07] : le niveau 0 correspond à une implémentation matérielle ; le niveau 1, *software-controlled radio (SCR)*, à une radio dont les paramètres sont contrôlés logiciellement ; le niveau 2, *software-defined radio (SDR)*, à une radio dont le traitement en bande de base est implémenté logiciellement ; le niveau 3, *ideal software radio (ISR)*, suppose un échantillonnage à l'antenne (suppression du *front end* analogique) pour rendre le traitement RF programmable ; le niveau 4, *ultimate software radio (USR)*, étend ces capacités avec une transition rapide (de l'ordre de la milliseconde) entre plusieurs protocoles. Dans le cadre de cette thèse, la définition de la radio logicielle correspond au niveau 2. La suite de cet état de l'art va se concentrer sur le traitement en bande de base, correspondant à la partie programmable.

Afin de ranger par catégorie les différentes plateformes de radio logicielle, il est nécessaire d'établir des critères objectifs. Définir des critères uniquement sur la base des technologies

utilisées (processeur dédié, type de mémoire, interconnexion, etc.) peut être délicat, la plupart des plateformes étant hétérogènes. De plus, la technologie utilisée n'est pas forcément un critère pertinent pour l'utilisateur d'une plateforme de radio logicielle. Les caractéristiques importantes d'une plateforme pour l'utilisateur sont : la programmabilité et la puissance de calcul, qui vont conditionner les protocoles supportés, ainsi que la consommation d'énergie qui peut être un facteur limitant pour l'adoption d'une plateforme.

Cependant, pour le programmeur, l'architecture est d'importance majeure car elle a une influence déterminante sur le modèle de programmation et les outils utilisables sur une plateforme donnée. En partant de ces considérations, je propose 5 catégories :

— l'approche *CPU généraliste* qui se base sur l'architecture d'un ordinateur ;
— l'approche *coprocesseur* qui ajoute un coprocesseur pour accélérer le calcul ;
— l'approche *multiprocesseurs* qui utilise une architecture dédiée à base de DSP (*Digital Signal Processor*) ;
— l'approche par *blocs configurables* qui remplace les DSP par des unités configurables dédiées à un traitement spécifique ;
— l'approche par *blocs programmables* qui construit sa propre architecture grâce à l'interconnexion programmable de blocs de base.

Ces approches seront décrites dans les sections suivantes, avec pour chacune d'entre elles des exemples d'implémentation. Les solutions logicielles utilisées seront décrites dans la Section 2.3.

2.2.2 Approche CPU généraliste

L'approche CPU (*Central Processing Unit*) généraliste décrite sur la Figure 2.2 utilise un processeur généraliste comme plateforme de calcul. C'est une solution flexible et facile à programmer, mais avec une forte consommation d'énergie pour un objectif de performance.

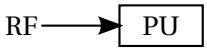


FIGURE 2.2 – Approche CPU généraliste.

La plateforme USRP (*Universal Software Radio Peripheral*) [USRP 14] est représentative de l'approche CPU généraliste. Elle possède un FPGA (*Field Programmable Gate Array*) pour le stockage des données en sortie de l'ADC. L'essentiel du traitement du signal est effectué sur le CPU, connecté au FPGA par un lien USB (USRP1) ou ethernet (USRP2). Cette plateforme est largement répandue et supportée par des logiciels tiers. Elle est développée pour fonctionner avec GNU Radio, mais est aussi compatible avec Labview ou Matlab.

Récemment, Microsoft a développé SORA [Tan 11]. Cette plateforme est connectée à l'ordinateur par un bus PCIe, qui permet de réduire la latence et augmenter le débit de données. Elle décode la norme Wi-Fi 802.11 b/g datant de 2003 en temps réel.

Le développement continu des technologies en micro-électronique pousserait à croire que les futurs ordinateurs seront capables de décoder tous les protocoles en temps réel. Comme le débit de données à traiter augmente plus vite que la puissance de calcul [Ulversoy 10], ce type d'architecture est donc capable de supporter uniquement des protocoles antérieurs.

2.2.3 Approche coprocesseur

Afin d'accélérer les calculs, des optimisations de l'approche CPU généraliste ont été récemment explorées. Comme présenté sur la Figure 2.3, elles reposent sur l'addition d'un coprocesseur pour réaliser les calculs lourds. Cela permet de réduire la consommation d'énergie tout en gardant une grande flexibilité et programmabilité.

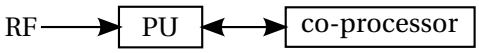


FIGURE 2.3 – Approche coprocesseur.

Le travail présenté par Horrein, Hennebert et Pétrot [Horrein 11] utilise un GPU (*Graphics Processing Unit*) comme coprocesseur dans un flot GNU Radio. Il permet un gain de l'ordre de 3 à 4 en temps de calcul.

La plateforme KUAR (Kansas University Agile Radio) [Minden 07] utilise un PC associé à un FPGA. Le choix du modèle de programmation est laissé au développeur, depuis l'implémentation en VHDL (*VHSIC Hardware Description Language*) jusqu'à une implémentation sur processeur proche de GNU Radio.

D'autres approches utilisent des DSP génériques comme processeur centralisé, ce qui permet une meilleure efficacité tout en restant générique. Texas Instruments propose un DSP 3 cœurs avec des accélérateurs spécialisés [Agarwala 07]. Cette puce fournit une plateforme pour les stations de base WCDMA, avec un support allant jusqu'à 64 utilisateurs et plusieurs protocoles.

La plateforme ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) [Bougard 08] développée par l'Imec est une architecture reconfigurable à gros grain. Elle est construite autour d'un CPU central et de l'accélérateur ADRES. L'ADRES est une grille de 16 unités fonctionnelles, vu par le processeur comme un coprocesseur VLIW (*Very Large Instruction Word*). Chacune d'entre elles est un processeur SIMD (*Single Instruction Multiple Data*), qui tire parti du parallélisme de données. La puce est programmée en C et utilise le compilateur DRESC [Mei 02]. Le compilateur DRESC permet de dérouler les boucles de traitement afin de les exécuter sur l'accélérateur ADRES. Cette plateforme vise les télécommunications avec des expérimentations sur du 802.11n à 108 Mbps et du LTE à 18 Mbps, avec une consommation moyenne de 333 mW [Bougard 08].

Les architectures précédentes offrent une efficacité énergétique limitée de par l'utilisation d'une architecture généraliste. Les catégories suivantes viennent combler cette lacune en utilisant des architectures spécialisées avec des processeurs dédiés au traitement du signal.

2.2.4 Approche multiprocesseurs

L'approche multiprocesseurs, représentée sur la Figure 2.4, possède une grande programmabilité. Cependant, son architecture dédiée n'autorise l'exécution que d'une certaine classe d'applications (traitement du signal dans ce cas).

La puce NXP EVP16 [Berkel 05] est composée de plusieurs unités de calcul. Un processeur ARM prend en charge le contrôle ainsi que les couches MAC et LINK. Un DSP conventionnel, un processeur vectoriel et plusieurs accélérateurs matériels sont utilisés pour le traitement du signal. Le processeur vectoriel est construit comme une chaîne de traitement vectorisée et adressé comme un VLIW. Il décode l'UMTS à un débit de 640 kbps à 35 MHz, avec un débit maximum de 300 MHz.

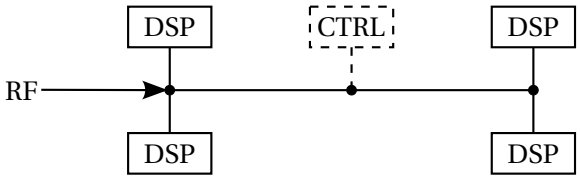


FIGURE 2.4 – Approche multiprocesseurs avec contrôle centralisé ou distribué.

Infineon a construit la puce MuSIC [Ramacher 07] comme une solution multi-DSP pour la radio logicielle. Le contrôle est réalisé par un processeur ARM. Le traitement du signal est assuré par 4 DSP SIMD et des processeurs dédiés pour le filtrage et le dédocage canal. La consommation en WCDMA est de 382 mW dans des conditions défavorables et 280 mW dans des conditions usuelles. Cette puce est fournie comme une solution commerciale sous le nom X-GOLD SDR 20 par Infineon [Ramacher 11]. Elle est programmée à l'aide d'un mélange de code C et de code assembleur pour les calculs critiques.

L'architecture Sandblaster [Schulte 04] utilise un DSP développé spécifiquement pour la radio logicielle. Le parallélisme de tâche est supporté par un mécanisme matériel. Sur le SB3011 [Glossner 07], 4 cœurs Sandblaster sont intégrés et contrôlés par un processeur ARM. Cette puce est programmée en C à l'aide d'un compilateur dédié. Sa consommation maximum est de 171 mW pour du WCDMA à 384 kbps.

L'université du Michigan à Ann Arbor développe la plateforme SODA [Woh 06], et son évolution ARDBEG [Woh 08]. SODA est développée comme un système complet pour la radio logicielle. Elle consiste en un ARM pour le contrôle et 4 DSP pour le traitement du signal. ARDBEG raffine cette plateforme en ajoutant un turbo décodeur matériel et des optimisations pour les DSP. La programmation est faite à l'aide de code C. La consommation sur la plateforme ARDBEG pour du WCDMA et du 802.11a est inférieure à 500 mW.

L'université de Dresden en Allemagne développe la puce Tomahawk [Arnold 14], visant le LTE et le WiMAX. Elle utilise deux processeurs *Tensilica* RISC (*Reduced Instruction Set Computer*) pour le contrôle, six DSP vectoriels et deux DSP scalaires pour le traitement du signal, ainsi que des accélérateurs matériels pour le filtrage et le décodage. L'ordonnancement est fait par un système matériel dédié et du code C est utilisé pour la programmation. Selon les estimations des auteurs, la consommation de la plateforme est d'environ 1,5 W.

Picochip [Pulley 03] propose une approche du traitement du signal basée sur un grand nombre de cœurs simples. Ces cœurs sont placés sur une matrice déterministe. Un flot de compilation basé sur le langage C est fourni par l'entreprise. Aucun résultat n'est présenté ; l'entreprise annonce l'OFDM et les stations de base 4G comme applications de référence.

L'université de Californie à Davis développe la puce ASAP (Asynchronous Array of Simple Processors) [Truong 09]. Ce projet vise à réaliser le traitement du signal sur des processeurs simples. Tous les processeurs peuvent communiquer avec leurs plus proches voisins sur une grille. La version 2 ajoute des accélérateurs pour la FFT, Viterbi et l'estimation de mouvement vidéo, tout en augmentant le nombre de cœurs à 167. Une implémentation complète du 802.11a/g avec un débit de 54 Mbps a une consommation de 198 mW.

Ces approches offrent un bon compromis entre la programmabilité et les performances. Leur consommation énergétique est cependant trop importante pour certaines plateformes dont l'énergie est la ressource critique.

2.2.5 Approche par blocs configurables

Afin de réduire la consommation d'énergie, certaines plateformes échangent leurs DSP contre des blocs configurables, comme illustré sur la Figure 2.5.

La différence entre un DSP spécialisé et un bloc configurable est très fine. Un DSP est capable de réaliser n'importe quel calcul, même si cela peut s'avérer inefficace. Un bloc configurable est vraiment spécialisé, et ne pourra effectuer que le traitement pour lequel il est construit. Par exemple, certains blocs configurables ne sont pas capables d'exécuter de code conditionnel. Des microcontrôleurs peuvent leur être associé pour réaliser ces opérations, que l'on appellera le code de contrôle. Cela implique une grande différence en terme de programmation, ces plateformes n'étant pas compatibles avec un flot de compilation classique.

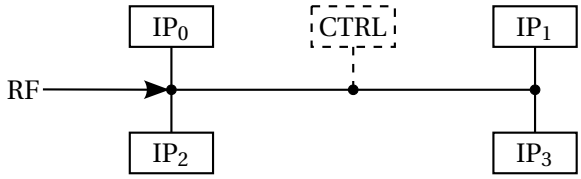


FIGURE 2.5 – Approche par blocs configurables avec contrôle centralisé ou distribué.

Fujitsu développe la plateforme LSI [Nishijima 06]. Cette plateforme utilise largement des accélérateurs matériels, associés à des processeurs reconfigurables. Tous ces composants sont reliés par un réseau d'interconnexions, et contrôlés par un processeur ARM. La puce est capable d'exécuter le 802.11a/b avec un débit maximum de 43 Mbps.

La plateforme de radio logicielle BEAR [Derudder 09] est l'évolution de l'ADRES développé par l'Imec. Elle est constituée d'un processeur ARM pour le contrôle et de trois ASIP (*Application Specific Instruction set Processor*) pour la synchronisation sur différents *front end* radios. Deux architectures reconfigurables à gros grain ADRES sont utilisées pour le traitement en bande de base avec un accélérateur Viterbi. La plateforme est programmée en C ou en Matlab en utilisant le flot de développement de l'Imec. Sur le plan de la consommation d'énergie, BEAR exécute un protocole 2x2 MIMO OFDM à 108 Mbps pour 231 mW. Imec commercialise la plateforme BEAR sous forme de licence.

La plateforme de radio logicielle Magali [Clermidy 09a] est développée par le CEA LETI comme plateforme de démonstration. Elle est construite autour d'un NoC (*Network on Chip*), chaque périphérique bénéficie d'un accès au réseau, et un processeur ARM agit comme contrôleur central. Le calcul est réalisé par les DSP spécialisés Mephisto [Clermidy 10] et des blocs reconfigurables pour l'OFDM, le décodage et le désentrelacement. Des mémoires distribuées intelligentes [Martin 09] sont réparties sur le NoC. La puce exécute le protocole LTE en 4x2 MIMO en réception dans le pire scénario avec une consommation de 236 mW [Jalier 10]. Pour plus de détails sur la plateforme Magali utilisée dans nos expérimentations, veuillez vous référer à la Section 5.2.

La plateforme Genepy du CEA LETI utilise une granularité plus large pour son approche distribuée. Elle reprend le NoC et les DSP spécialisés de la plateforme Magali. Le contrôle est réalisé par des processeurs RISC distribués. L'architecture est composée de grappes de calcul homogènes. Chaque grappe sur le réseau est composée de deux cœurs Mephisto, une mémoire distribuée intelligente et un contrôleur RISC. La plateforme est homogène, sans

accélérateur matériel. La consommation affichée sur une réception LTE 4x2 MIMO est de 192 mW.

La plateforme ExpressMIMO [Schmidt-Knorreck 12] est développée par EURECOM. Toutes les unités configurables possèdent une interface commune, un DMA et un microcontrôleur. Chaque unité a un bloc configurable spécifique pour le traitement du signal. La plateforme vise une implémentation MIMO OFDM, procédé de codage utilisé dans plusieurs protocoles tels que le Wi-Fi et le LTE. Elle utilise l'environnement de programmation ouvert OpenAirInterface [OAI 14].

L'université de Twente au Pays-Bas développe la puce Annabelle [Zhang 09]. Elle est construite autour d'un NoC et utilise des cœurs reconfigurables à gros grain. Un processeur ARM est utilisé comme contrôleur et des accélérateurs (Viterbi, etc.) sont connectés au processeur ARM par un bus AMBA. Les tests publiés concernent uniquement le procédé OFDM.

Les plateformes à blocs configurables que l'on vient de voir offrent des performances optimales, mais leur spécialisation limite leur évolutivité pour la programmation de protocoles non établis au moment de leur conception.

2.2.6 Approche par blocs programmables

Afin de spécialiser l'architecture au maximum tout en restant flexible, la dernière approche utilise des blocs programmables, illustrée sur la Figure 2.6. Les connexions entre les blocs sont programmables, ce qui permet de construire une architecture dédiée à l'application visée. Cette technologie reprogrammable apporte adaptabilité et évolutivité aux architectures, et se retrouve principalement dans les FPGA. Elle offre une grande puissance de calcul pour une consommation modérée.

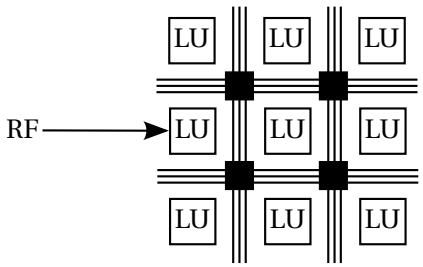


FIGURE 2.6 – Approche par blocs programmables.

La puce XiSytem [Lodi 06] est une architecture VLIW proposant 3 chemins de données parallèles, dont un PiCoGa (Pipelined Configurable Gate Array). Le PiCoGa est un FPGA à matrice déterministe ; le nombre d'interconnexions est limité et structuré comme une chaîne de traitement. Il exécute des instructions spécifiques pour le processeur. Le développement est fait en C pour programmer le VLIW et le PiCoGa. L'architecture vise le traitement du signal embarqué de manière générale, avec des expérimentations sur l'encodage MPEG2 possédant une consommation de 300 mW.

L'université Rice développe WARP [WARP 14], une plateforme ouverte de radio logicielle. La programmation est réalisée en langage VHDL. Une communauté de développement autour de la plateforme menée par l'université Rice propose des implémentations au code source ouvert. Par exemple, un design de référence pour le MIMO OFDM est disponible et il peut être

étendu à l'aide des outils Xilinx.

WINC2R est une plateforme originale pour la radio logicielle développée par l'université Rutgers. Elle est construite sur un FPGA, qui implémente des cœurs logiciels et des accélérateurs. Les cœurs logiciels sont programmés avec GNU Radio. Le calcul peut être équilibré entre les processeurs et les accélérateurs en fonction des contraintes. Contrairement à l'approche précédente, les blocs configurables permettent de choisir et de modifier les accélérateurs durant le développement. Une implémentation du 802.11a a été réalisée sur la plateforme WINC2R par Satarkar [Satarkar 09].

L'entreprise Nutaq [Nutaq 14] propose des outils de développement et des plateformes de radio logicielle basés sur des FPGA. Le développement est réalisé sur Simulink ou en VHDL. La plateforme est présentée comme supportant le MIMO WiMAX. Plusieurs autres entreprises proposent des approches similaires basées sur FPGA [Pentek 14, USRP 14, Sundance 14].

Ces plateformes à blocs programmables sont les plus à même pour l'expérimentation de nouveaux protocoles de par leur flexibilité, au détriment de leur consommation.

2.2.7 Discussion

Afin d'illustrer les différentes approches, la Table 2.1 résume les caractéristiques de plateformes représentatives de chacune des approches. Dans cette table, la consommation d'énergie n'est pas définie pour les plateformes basées sur des FPGA, car très dépendante de leur configuration. En se basant sur ces plateformes, je présente maintenant les champs applicatifs privilégiés de chaque approche.

Plateforme	Disponibilité	Application	Langage	Conso (mW)
USRP [USRP 14]	commercial	N/A	C++	≈ PC
TI C64+ [Agarwala 07]	commercial	base station	C/ASM	6000
MuSIC [Ramacher 07]	commercial	WCDMA	C/ASM	≤ 382
Sandblaster [Schulte 04]	IP licence	WCDMA	C	171
ARDBEG [Woh 08]	prototype	WCDMA	C	≤ 500
BEAR [Derudder 09]	IP licence	MIMO OFDM	Matlab/C	231
Magali [Clermidy 09a]	prototype	MIMO OFDM	C/ASM	236
ExpressMIMO [Schmidt-Knorreck 12]	prototype	MIMO OFDM	C	N/A
WARP [WARP 14]	commercial	MIMO OFDM	VHDL	N/A
Nutaq [Nutaq 14]	commercial	N/A	Matlab/VHDL	N/A
ASAP [Truong 09]	prototype	802.11a/g	N/A	198
Genepy [Jalier 10]	prototype	MIMO OFDM	C/ASM	192

TABLE 2.1 – Comparaison des principales plateformes de radio logicielle selon les résultats de performances publiés.

L'approche CPU généraliste est la plus simple à utiliser, mais elle ne permet pas d'étudier la consommation d'énergie ou *l'Adéquation Algorithme Architecture* (AAA). En effet, les CPU généralistes possèdent un comportement radicalement différent des plateformes dédiées. Il est alors difficile de comparer la puissance de calcul et la consommation énergétique pour cette approche et les autres. Par exemple, pour un protocole donné, les besoins en puissance de calcul mesurés en nombre d'opérations par seconde peuvent varier d'un facteur 100 dans

la littérature, selon la granularité de l'architecture choisie.

Afin d'étudier la puissance de calcul et d'avoir une consommation d'énergie limitée, l'utilisation d'un MPSoC (*Multiprocessor System-on-Chip*) hétérogène semble être un bon point de départ. Dans cette optique, les approches multiprocesseurs comme BEAR [Derudder 09] ou par blocs configurables comme Magali [Clermidy 09a] sont un choix pragmatique, ces plateformes étant dédiées, et donc optimisées, pour la radio logicielle.

Le problème de ce type de solution est qu'elle rend dépendant de l'architecture de la plateforme. En effet, le portage d'une application vers une architecture différente peut s'avérer compliqué. Le développement d'une couche d'abstraction commune est un défi qui permettrait de rendre possible le développement multiplateforme en radio logicielle.

L'approche par blocs programmables offre une plateforme à la fois flexible et efficace pour le prototypage grâce à l'utilisation de la technologie FPGA. Elle permet une polyvalence dans le choix de l'architecture de calcul, avec des choix radicalement différents pour WARP [WARP 14] ou WINC2R [Satarkar 09] par exemple.

Il est à noter qu'aucune de ces approches ne s'est imposée jusqu'à maintenant, des développements récents ont lieu autant dans l'industrie que le monde académique. À partir de ce constat, la conclusion évidente de cette section est qu'il n'existe pas de modèle commun d'architecture pour fournir, comme pour les processeurs généralistes, une abstraction matérielle permettant de programmer des applications de radio logicielle de manière portable. Dans la section suivante, je décrirai les différentes approches utilisées pour répondre au problème de la programmation de ces plateformes.

2.3 Modèles de programmation

Comme nous venons de le voir, des efforts considérables ont été mis dans la réalisation de plateformes matérielles dédiées, dans l'industrie comme dans le monde académique. De ces travaux, nous pouvons conclure que i) le support des mécanismes matériels est nécessaire pour atteindre les exigences de performances et de consommation d'énergie des protocoles radio modernes ; ii) au vu de la complexité des plateformes, il n'est pas raisonnable d'écrire du code C/assembleur et de le placer manuellement. En effet, l'utilisation de processeurs dédiés hétérogènes, d'architectures mémoire non homogènes, et de réseaux d'interconnexions programmables, nécessite une connaissance approfondie des plateformes qui est difficilement appréhendable sans une couche d'abstraction, qui est absente du simple langage C.

Programmer et exécuter des *formes d'ondes*, c.-à-d. la couche physique d'un protocole de communication radio, est clairement une application du problème de la programmation des machines parallèles. On doit tenir compte de ce problème lors de la programmation d'une plateforme matérielle de radio logicielle.

Je présente dans cette section les recherches effectuées pour programmer ces plateformes de manière efficace, c.-à-d. pour atteindre les performances souhaitées avec un effort de développement raisonnable. Je commencerai par un aperçu des environnements utilisés pour programmer plusieurs plateformes. L'état de l'art se concentrera ensuite sur un aspect central de la programmation d'une radio logicielle : la programmation d'une forme d'onde. C'est un défi d'exprimer une forme d'onde dans un format de haut niveau permettant d'être compilé vers une plateforme matérielle en tirant parti de ses spécificités. Pour répondre à ce défi, il existe plusieurs solutions dans l'état de l'art. Je présenterai dans les sections suivantes les environnements de programmation de radio logicielle utilisés pour exprimer et compiler ces formes d'ondes.

2.3.1 Environnement de programmation pour la radio logicielle

Il y a deux problèmes principaux dans l'environnement de programmation. Le premier est de trouver le modèle de calcul à utiliser pour spécifier la forme d'onde (décrit dans la section suivante). Le second est de compiler cette forme d'onde efficacement sur la plupart des plateformes mentionnées dans la section précédente. Choisir le bon environnement de programmation ne dépend pas simplement de la comparaison de critères objectifs. Il dépend largement de choix stratégiques dans une entreprise et de l'acceptation culturelle par les développeurs.

Le système applicatif SCA (*Software Communication Architecture*) a été développé sous l'impulsion du département de la défense américain dans le cadre du JTRS (*Joint Tactical Radio System*) et suivi par le *Wireless Innovation Forum*. SCA utilise des technologies de système distribué comme CORBA (*Common Object Request Broker Architecture*) par exemple. Un exemple d'implémentation du SCA est OSSIE [Gonzalez 09]. Mais SCA est en général déployé sur des produits militaires par des approches propriétaires propres à chaque fournisseur (Thales, Harris, etc.). Les outils SCA, tels que ceux de NordiaSoft ou Prismtech, visent à fournir un environnement de développement pour le prototypage rapide de formes d'ondes et/ou de la couche d'abstraction intergicielle (*middleware*) nécessaire au SCA. Le SCA est un standard que les produits de communications militaires doivent respecter afin de permettre l'interopérabilité entre les différents corps d'armée, mais également entre les forces des différents pays de l'Alliance Atlantique (OTAN) [Moy 13].

Le projet *open-source* GNU Radio propose d'implémenter une radio logicielle en utilisant une bibliothèque de blocs de traitement du signal écrits en C++, et de les interfacer à l'aide du langage de programmation Python. Bien que souvent associé la plateforme matérielle USRP [USRP 14], GNU Radio a récemment gagné l'attention d'autres fabricants de radio logicielle [Nutaq 14]. Actuellement, GNU Radio est développé pour des plateformes généralistes. Il ne permet pas d'implémenter des protocoles complexes tels que le MIMO OFDM avec des contraintes temps réel. D'autres projets suivant la même approche, comme OpenAirInterface [OAI 14], utilisent des systèmes d'exploitation temps réel pour améliorer la gestion de l'exécution de la forme d'onde.

De nombreux environnements utilisés pour la radio logicielle sont basés sur une interface graphique couplée à des implémentations dédiées. Par exemple l'outil Simulink est couplé à Mathworks pour la programmation de FPGA. National Instruments propose un outil similaire pour LabView. De nouveaux développements basés sur OpenMP ou OpenCL apparaissent [Jaaskelainen 10, Wang 07], mais ils n'ont pas encore gagné suffisamment d'attention dans le domaine de la radio logicielle pour évaluer leur pertinence.

2.3.2 Programmation impérative concurrente

Pour un développeur de logiciel embarqué, la manière naturelle de développer sur une plateforme de radio logicielle est d'utiliser un langage impératif (généralement le C) associé à des fils d'exécution pour représenter le parallélisme. Cette méthode est utilisée pour programmer des formes d'ondes pour les plateformes parallèles homogènes et hétérogènes. Par exemple, les différentes unités de calcul de la plateforme BEAR [Derudder 09] sont programmées en utilisant du code C et Matlab.

La programmation et l'exécution efficace de forme d'ondes sont étroitement liées aux avancées dans la programmation des plateformes hétérogènes. Bien que n'ayant pas encore été évalués pour la programmation de formes d'ondes, l'environnement de programmation

ExoCHI [Wang 07] ainsi que le cadre logiciel Merge [Linderman 08] sont des propositions visant à faciliter la programmation de plateformes hétérogènes tout en atteignant de bonnes performances. La solution proposée est d'étendre OpenMP avec des fonctions intrinsèques et de placer dynamiquement le logiciel sur les ressources disponibles.

Cohen et Rohou [Cohen 10] proposent une approche similaire dans laquelle les programmes sont compilés dans un bytecode spécifique avant le chargement. Ils sont compilés dynamiquement pour les différents accélérateurs disponibles sur la plateforme lors de l'exécution. Cette approche n'a pas encore été évaluée pour la programmation de plateformes de radio logicielle.

De nombreux travaux isolés se concentrent sur l'utilisation d'accélérateurs matériels. Le compilateur DresC [Mei 02] permet de dérouler des boucles de manière à exécuter du code parallélisé sur un accélérateur spécifique constitué de 64 unités fonctionnelles. L'intégration du GPU dans la programmation de radio logicielle a aussi été étudiée. Horrein, Hennebert et Pétrot [Horrein 11] comparent différentes architectures systèmes pour utiliser un GPU pour la programmation de radio logicielle. Leur travail est basé sur OpenCL et GNU Radio.

2.3.3 Programmation flot de données

De nombreux travaux argumentent en faveur d'un changement de paradigme et proposent de programmer des formes d'ondes en utilisant des langages flot de données. Ces langages s'appuient sur un modèle de calcul dans lequel un programme est représenté comme un graphe dirigé, composé d'acteurs reliés par des arcs. Un graphe flot de données suit une exécution dirigée par les données : un acteur ne peut être exécuté que si suffisamment de données sont disponibles sur ses arcs d'entrées. Lorsqu'il s'exécute, il consomme un certain nombre de données de ses arcs d'entrée et produit un certain nombre de données sur ses arcs de sortie. J'étudierai plus en détails les différents modèles de calcul flot de données dans la Section 3.2. Dans cette section, je passe en revue les travaux visant à appliquer le modèle flot de données à la radio logicielle.

StreamIt [Thies 09] est un langage de programmation qui permet de décrire des programmes en CSDF (*Cyclo-Static DataFlow*), par l'utilisation d'opérateurs de filtres et de *split/-join*. Il est complété par des outils qui réalisent des analyses statiques et optimisent le graphe flot de données. Le compilateur génère du code C associé à des fils d'exécutions, que le développeur doit placer manuellement sur les ressources matérielles disponibles. Le modèle de calcul CSDF est restreint à un flot unique dans StreamIt, ce qui le rend inutilisable pour des formes d'ondes complexes et dynamiques telles que le LTE.

ΣC [Goubier 11] permet de programmer des formes d'ondes en utilisant une extension du langage C. Le modèle de calcul correspondant est plus expressif que SDF (*Synchronous DataFlow*) grâce à des extensions non déterministes, mais autorise des analyses statiques telles que la limitation de la taille mémoire. Cependant il n'est pas possible de décrire le comportement dynamique d'acteurs, ce qui limite l'approche lorsque l'on décrit des formes d'ondes telles que LTE.

Des travaux passés ont montré l'intérêt de programmer en utilisant un langage généraliste enrichi de primitives permettant de construire le graphe flot de données. Suivant l'approche Stream Virtual Machine [Labonte 04], StreamWare [Gummaraju 08] propose d'écrire des graphes flot de données dans une API C dédiée et de les ordonnancer lors de l'exécution sur une machine généraliste. La même approche est appliquée au LTE [Ben Abdallah 10] en utilisant une machine virtuelle (LUA). La forme d'onde programmée contient des primitives de

reconfiguration dédiées écrites en langage LUA et interprétées directement par le contrôleur de la plateforme. Ces travaux ne se limitent pas à un modèle de calcul particulier.

Dans une approche similaire, le flot d'outils Nucleus [Castrillon 11] comprend un ensemble d'outils capable de compiler et de placer des formes d'ondes. Il utilise le cadre logiciel MAPS [Castrillon 13] afin de décrire les acteurs dans le langage CPN. Plusieurs implémentations sont fournies pour chaque acteur, et un placement guidé par l'utilisateur calcule un ordonnancement.

L'outil SystemVue [SystemVue 14] permet de modéliser des formes d'ondes en SDF ou TSDF (*Timed Synchronous DataFlow*). Il est utilisé comme base pour un travail récent [Hsu 10] qui vise à prendre en compte le caractère dynamique du LTE en introduisant la vectorisation et la sérialisation dans un graphe flot de données.

L'approche DiplodocusDF [Gonzalez-Pina 12] propose une modélisation spécifique au domaine de la radio logicielle. Le langage proposé possède une sémantique et une syntaxe précise permettant de l'utiliser pour la simulation, la vérification et la synthèse vers le langage C. Il se base sur une API de traitement du signal pour effectuer les traitements. Le placement ainsi que le support d'exécution pour une plateforme matérielle sont en cours de développement.

Le langage CAL (*CAL Actor Language*) [Bhattacharyya 08] permet de modéliser des graphes flot de données indépendamment du modèle de calcul. Il propose un grand nombre d'outils pour ordonnancer et générer du code pour différentes plateformes. Une étude récente [Gu 10] propose d'analyser le graphe flot de données pour le limiter au modèle de calcul le plus restreint, ce qui augmente les possibilités d'analyse et d'optimisation par la suite.

2.3.4 Programmation multi-paradigmes

Un ensemble de travaux proposent de mélanger plusieurs modèles de calcul. Ces approches cherchent à tirer parti des points forts de chacun de ces modèles. Par exemple, le traitement du signal peut se baser sur un modèle flot de données, alors que la gestion du mode de réception peut être prise en charge par un programme impératif.

L'approche SPEX [Lin 06] permet de programmer des formes d'ondes en utilisant trois paradigmes. *Kernel* SPEX permet une programmation séquentielle impérative utilisée pour la compilation SIMD ou VLIW. *Stream* SPEX est utilisé selon le paradigme flot de données, suivant le modèle de calcul KPN (*Kahn Process Network*). *Synchronous* SPEX s'appuie sur le paradigme utilisé dans les langages synchrones tels que Esterel ou Signal. La distribution des paradigmes est laissée au programmeur, mais toutes les parties sont inclues dans un programme C++ dans lequel i) le choix du paradigme est indiqué par un mot clé, ii) aucune création dynamique d'objet n'est autorisée. La compilation de ce programme implique un compilateur par paradigme.

De manière similaire, IRIS [Sutton 10] propose d'écrire des composants *sPHY* et *fPHY*. sPHY implémente le modèle de calcul SDF alors que fPHY implémente le modèle de calcul KPN. Le placement de ces composants est laissé au développeur. La structure utilisée fournit un support pour la reconfiguration : les composants peuvent déclencher des signaux conduisant à la reconfiguration des cœurs de calcul.

Lime [Auerbach 10] est un langage basé sur Java avec des extensions pour représenter le parallélisme. Dans Lime, la même méthode peut être utilisée comme une fonction standard ou comme un acteur pour une programmation flot de données. Dans ce cas, Lime permet d'exécuter les acteurs à différents rythmes pour communiquer, étendant ainsi SDF tout en gardant des capacités d'analyse. Il est associé à un compilateur qui génère du bytecode Java, du

C ou du Verilog, afin de permettre de choisir entre différentes implémentations pour chaque acteur.

Finalement, il est important de mentionner qu'un grand nombre de laboratoires travaillent sur la conception de systèmes complets depuis une spécification de haut niveau dans le domaine du *hardware-software co-design*. Ces travaux apportent des avancées sur des aspects spécifiques tels que la conception de plateforme assistée ou des outils de synthèse de haut niveau tels que CatapultC par exemple. Ces travaux n'ont pas encore conduit à des environnements dédiés à la radio logicielle.

2.3.5 Discussion

La revue des environnements de programmation présentée montre que, comme c'est le cas pour les plateformes matérielles, il n'y a pas de consensus sur ce que devrait être un environnement de programmation pour la radio logicielle. Cependant, il y a une tendance claire vers un changement de paradigme de manière à prendre en charge des protocoles tels que le LTE. Ces nouveaux protocoles sont largement différents des applications de traitement du signal antérieures, qui pouvaient être programmées de manière statique (SDF et/ou impératif classique).

Les arguments en faveur des modèles de calcul flot de données sont les suivants :

— Les formes d'ondes sont intrinsèquement flot de données car elles opèrent une suite de traitements sur de longues séquences de données. Bien que non infinies, elles sont regroupées par trames, les formes d'ondes requièrent encore des filtres (matériel ou logiciel) qui sont facilement exprimés par l'utilisation d'acteurs flot de données.

— Les applications de radio logicielle requièrent une puissance de calcul considérable et doivent donc utiliser de manière efficace le parallélisme disponible dans le matériel. Le formalisme flot de données permet des implémentations parallèles car il expose le parallélisme de tâches et de données.

— Les programmes flot de données ont une expressivité restreinte qui permet de les analyser afin de vérifier des propriétés comme l'absence d'interblocage, ou pour améliorer les analyses temporelles. Ces analyses sont importantes car les formes d'ondes deviennent de plus en plus complexes.

Bien qu'elles demandent un effort supplémentaire pour leur mise en œuvre, les approches flot de données semblent prometteuses. À mon avis, les approches hétérogènes entre ce paradigme et les langages impératifs concurrents permettront de trouver un compromis entre la programmabilité, la performance, la portabilité et la démontrabilité.

Ce chapitre a présenté un état de l'art autour de la radio logicielle. Il pose de nombreux problèmes ouverts, dont certains appartiennent plus généralement à la programmation des systèmes parallèles. Je vais chercher à les résoudre dans la suite de cette thèse. Le dynamisme des applications a poussé à l'adoption de nouveaux modèles de calculs plus dynamiques. Ils nécessitent de repenser les méthodes d'analyse et de vérification de ces modèles, et aussi l'implémentation efficace de ces modèles sur des plateformes matérielles. D'autre part, il est nécessaire d'apporter une couche d'abstraction cohérente pour ces plateformes matérielles complexes, afin d'améliorer le développement et la portabilité des formes d'ondes.

Je présenterai dans le Chapitre 3 le modèle de calcul qui semble le plus adapté au dynamisme des applications de radio logicielle, ainsi que les analyses réalisables sur ce modèle. L'implémentation de ce modèle sur une plateforme de radio logicielle complexe sera ensuite illustrée dans les Chapitres 4 et 5.

3 Ordonnancements pour graphes flot de données

Obviously, simulation and testing may pinpoint some errors of this kind. It is well known, however, that testing is efficient only in the initial steps of a design, and that formal methods are necessary to find the last bugs.

— Paul Feautrier, *Scalable and Structured Scheduling*

Sommaire du chapitre

3.1 Introduction

Comme nous l'avons vu dans le chapitre précédent, de nombreux travaux de recherche motivent un changement de paradigme de programmation et prônent l'utilisation de modèles de programmation basés sur les modèles de calcul flot de données. Ces modèles de calcul proposent une abstraction naturelle du traitement du signal, ce qui permet une adoption plus aisée dans ce domaine d'application. Des environnements de développement tels que Simulink ou LabView proposent d'utiliser cette abstraction. De plus, les modèles de calcul flot de données offrent une représentation du programme de haut niveau qui autorise des analyses ainsi que des optimisations à même de permettre le niveau de performance requis par la radio logicielle. Pour ces raisons, j'ai choisi de baser mon travail sur ce modèle de calcul.

Dans ce chapitre, je commencerai par décrire le modèle de calcul flot de données dans une approche statique puis dynamique dans la Section 3.2. Je présenterai dans la Section 3.3 les différentes techniques d'ordonnancement existantes à l'heure actuelle et leurs limitations. La Section 3.4 me permettra d'introduire la notion de micro-ordonnancement. Cette méthode vise à raffiner la représentation des communications dans un formalisme flot de données, et son application au contrôle des tailles mémoires. Je discuterai enfin des apports de cette approche ainsi que de ses limitations dans le cadre de mon travail.

3.2 Les modèles de calcul flot de données

Les modèles de calcul flot de données sont issus des travaux sur la programmation parallèle développé depuis les années soixante. Ces travaux ont entre autres donné naissance aux réseaux de Petri [Peterson 77] ainsi qu'aux réseaux de processus de Kahn [Kahn 74]. Par la suite, ces modèles de calcul ont donné lieu au développement d'architectures de processeur flot de données visant à remplacer l'architecture de Von Neumann. Ces architectures cherchaient à tirer parti du parallélisme d'instructions mais ne se sont jamais imposées, à cause de la granularité trop fine de ce parallélisme. Elles ont évolué vers une forme hybride, combinant une machine de Von Neumann pour l'exécution de blocs d'instructions et la représentation flot de données pour tirer parti du parallélisme entre ces blocs. Pour un historique plus complet, voir l'article de Johnston [Johnston 04].

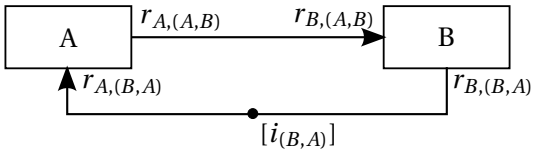


FIGURE 3.1 – Exemple de graphe flot de données.

De manière formelle, on représente un graphe flot de données comme un graphe dirigé $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Un acteur $v \in \mathcal{V}$ représente un bloc de calcul ou un sous-graphe hiérarchique. Un arc dirigé $e \in \mathcal{E}$ représente une FIFO (*First In First Out*) entre deux acteurs. Un graphe flot de donnée est illustré par la Figure 3.1. Il suit une exécution dirigée par les données : un acteur ne peut s'exécuter que s'il a suffisamment de données disponibles sur chacun de ses arcs d'entrée. L'exécution atomique d'un acteur consomme des données sur ses arcs d'entrée et en produit sur ses arcs de sortie. Cette consommation (resp. production) est fixée par un rythme $r_{v,e}$ qui associe à chaque arc e d'un acteur v un nombre de données consommées (resp. produites)

lors d'une exécution. L'état du graphe est défini comme le nombre de données présentes sur chacun de ces arcs, ce nombre est strictement positif car il représente le nombre de données dans une FIFO. L'état initial du graphe est le nombre initial de données i_e présent sur chaque arc e.

Je concentre cette étude sur les modèles de calcul flot de données dont le graphe a une topologie statique, c'est à dire que l'ensemble des acteurs et des arcs sont fixés et connus à la compilation. Pour les comprendre, je vais dans un premier temps présenter en détail le modèle statique SDF dans la Section 3.2.1. Je décrirai ensuite différents modèles qui permettent d'apporter du dynamisme dans la Section 3.2.2. Puis je détaillerai le modèle SPDF (*Schedulable Parametric DataFlow*) dans la Section 3.2.3, modèle utilisé dans la suite de la thèse.

3.2.1 Modèle statique

Le modèle statique de référence utilisé en flot de données est le modèle de calcul SDF [Bhattacharyya 99]. Dans ce modèle, les rythmes de consommation et de production sont connus statiquement. Un exemple de graphe SDF est illustré sur la Figure 3.2. Un intérêt majeur de ce modèle est de pouvoir déterminer statiquement, si il existe, un ordonnancement valide de l'exécution des acteurs. Cet ordonnancement prouve que l'ensemble des acteurs est bien exécuté, ce qui garantit la vivacité du graphe. De plus, il prouve que le graphe revient dans son état initial après une certaine séquence d'exécution, ce qui garantit l'existence d'une borne à la taille des FIFO.

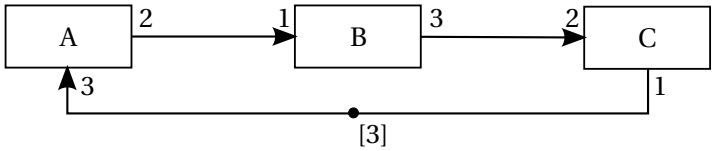


FIGURE 3.2 – Exemple de graphe SDF.

Afin de vérifier ces propriétés nous pouvons calculer le nombre d'exécutions de chacun des acteurs pour que le graphe revienne à son état initial, appelé *itération*. Le nombre d'exécutions pour un acteur, noté $\#v$, est obtenu en résolvant le système d'équations d'équilibre du graphe. Ce système est composé d'une équation (3.1) par arc e.

$$\#A \cdot r_{A,(A,B)} = \#B \cdot r_{B,(A,B)} \tag{3.1}$$

Un graphe est cohérent si le système d'équations d'équilibre a des solutions non nulles. La solution minimale de ces équations est appelé le *vecteur d'itération*. Dans l'exemple de la Figure 3.2, nous obtenons les solutions $\#A = 1$, $\#B = 2$, $\#C = 3$, soit le vecteur d'itération (A^1, B^2, C^3). Ces résultats permettent de calculer statiquement un ordonnancement selon plusieurs critères d'optimisations, dont la taille des mémoires, le temps d'exécution, la taille du code, etc., comme nous le verrons dans la Section 3.3.1.

L'intérêt du modèle SDF vient donc de sa grande prédictibilité. Cependant, la limitation à des nombres de données connus statiquement le rend inutilisable pour un grand nombre d'applications. Pour pallier à cet inconvénient, un grand nombre de modèles dynamiques ont été développés. Je détaille quelques-uns de ces modèles dans la section suivante.

3.2.2 Modèles dynamiques

Les différents modèles de calcul flot de données varient en expressivité et analysabilité. La Figure 3.3 représente cette variation pour un ensemble de modèles de calcul flot de données.

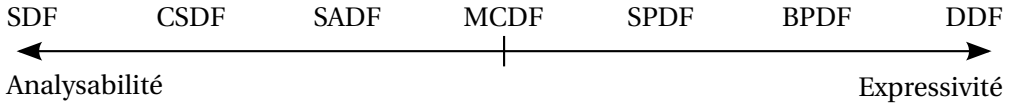


FIGURE 3.3 – Équilibre entre expressivité et analysabilité des modèles de calcul flot de données.

Il est à noter qu'aucune échelle n'est donnée de manière volontaire, car certains modèles sont très proches voire équivalents. L'idée est qu'on peut réaliser plus d'analyses et garantir plus de propriétés en limitant l'expressivité d'un modèle, par exemple l'absence d'interblocage. Les extrêmes sont constitués par le modèle SDF dont la consommation et la production sont statiques, et le modèle DDF (*Dynamic DataFlow*) dont la consommation et la production varient dynamiquement à l'exécution.

Flot de données cyclique

Les modèles flot de données cycliques du type CSDF [Bilsen 96] permettent de représenter la consommation (resp. production) d'un acteur comme une suite finie de consommations (resp. productions). Bien qu'équivalent au modèle SDF en expressivité, CSDF permet de faciliter l'expression de certaines applications, et de limiter la consommation mémoire [Bhattacharyya 96].

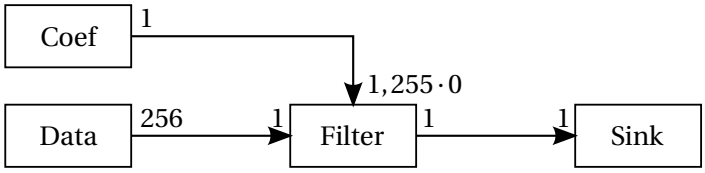


FIGURE 3.4 – Exemple de graphe CSDF.

Un exemple de graphe CSDF est donné dans la Figure 3.4. Dans cet exemple, un acteur filtre change de coefficient toutes les 256 exécutions. Ce changement est explicite grâce à l'utilisation de la syntaxe $1,255 \cdot 0$, qui veut dire : lire une donnée lors de la première exécution, suivi de 255 exécutions sans lecture de données sur ce port. Si nous voulions exprimer la même application dans un graphe SDF, il serait alors nécessaire de dupliquer 256 fois le coefficient et de le lire à chaque exécution du filtre.

Flot de données avec scénario

Les modèles flot de données qui utilisent un scénario comme SADF (*Scenario Aware DataFlow*) [Theelen 06] ou MCDF (*Mode Controlled DataFlow*) [Moreira 12] permettent d'augmenter l'expressivité du modèle en proposant plusieurs comportements par acteur. Les différents états d'un graphe sont représentés sous la forme d'une machine d'états. À chaque état est associé une partie de graphe ou des consommations et productions différentes. À l'aide de ce modèle, nous pouvons représenter différentes phases d'une application. Par exemple, nous

pouvons découper un protocole radio en deux scénarios, la synchronisation et la réception d'une trame.

Flot de données paramétrique

Les modèles flots de données paramétriques à l'image de PSDF (*Parametrized Synchronous DataFlow*) [Bhattacharya 01] ou SPDF [Fradet 12] se servent de paramètres symboliques pour définir le rythme de consommation ou de production d'un acteur. L'utilisation d'une représentation symbolique permet de conserver des capacités d'analyse comme le calcul du nombre d'itérations symboliques tout en augmentant largement l'expressivité. Grâce au paramètre, nous pouvons représenter un acteur produisant un nombre variable de données en fonction de la modulation utilisée. Une étude plus approfondie du modèle de calcul SPDF est proposée dans la Section 3.2.3.

Flot de données dynamique

Les modèles de calcul dynamique du type BDF (*Boolean DataFlow*) et DDF [Buck 93] relâchent les contraintes sur la production et la consommation des acteurs. Ces modèles permettent une expressivité maximale mais n'offrent cependant pas les mêmes capacités d'analyse et de garantie. Leur intérêt par rapport à la programmation séquentielle est alors d'exposer le parallélisme de tâche comme de données, propriétés à priori présentes dans tous les modèles flot de données.

3.2.3 Le modèle SPDF

J'ai choisi comme base de travail, parmi les nombreux modèles flot de données présentés, le modèle SPDF [Fradet 12]. Ce choix est motivé par la présence de paramètres qui permettent de représenter le dynamisme existant dans les applications de radio logicielle visées par cette étude. De plus, il offre un certain nombre d'analyses et de garanties à la compilation. Afin de comprendre ce modèle et ses garanties, je les résume dans cette section. Pour plus de détails sur le modèle, vous pouvez vous référer à l'article de Fradet, Girault et Poplavko [Fradet 12].

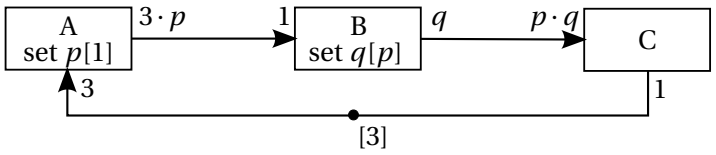


FIGURE 3.5 – Exemple de graphe SPDF.

Un exemple de graphe SPDF est donné Figure 3.5. Par rapport à un graphe SDF, nous remarquons deux particularités : les rythmes de consommation et de production peuvent être composés de produit de paramètres selon la grammaire (3.2). $\mathscr{P}$ est un ensemble de variables symboliques ; un paramètre p est modifié par un acteur à chaque itération ou après α exécutions avec l'annotation set $p[\alpha]$.

$$\mathscr{F} \quad ::= \quad k \mid p \mid \mathscr{F}_1 \cdot \mathscr{F}_2 \quad \text{où } k \in \mathbb{N}^* \text{ et } p \in \mathscr{P} \tag{3.2}$$

De manière formelle, Fradet, Girault et Poplavko proposent la Définition 1 pour un graphe SPDF.

Définition 1 *Un graphe SPDF est un tuple* $(\mathscr{G}, \mathscr{P}, i, d, r, M, \alpha)$ *où :*

- $\mathscr{G}$ *est un graphe connecté* $(\mathscr{V}, \mathscr{E})$ *avec* $\mathscr{V}$ *un ensemble d'acteurs et* $\mathscr{E} \subseteq \mathscr{V} \cdot \mathscr{V}$ *un ensemble d'arcs dirigés,*
- $\mathscr{P}$ *un ensemble de paramètres ;*
- $i : \mathscr{E} \to \mathbb{N}$ *associe à chaque arc un nombre de données initiales ;*
- $d : \mathscr{P} \to 2^{\mathbb{N}^*}$ *retourne l'intervalle de chaque paramètre ;*
- $r : \mathscr{V} \cdot \mathscr{E} \to \mathscr{F}$ *rythme de production ou de consommation d'un port (représenté par un acteur et un arc) ;*
- $M : \mathscr{P} \to \mathscr{V}$ *modificateur d'un paramètre ;*
- $\alpha : \mathscr{P} \to \mathscr{F}$ *période de modification d'un paramètre.*

La modification d'un paramètre entre chaque itération est garantie par les analyses de vivacité et de borne de taille mémoire (voir Section 3.2.1). Lorsque ce paramètre change plus d'une fois par itération, il est nécessaire de réaliser des analyses supplémentaires, car toutes les périodes de changement ne garantissent pas l'intégrité du graphe. Dans l'exemple de la Figure 3.5, une période de changement de 1 pour le paramètre q ne serait pas correcte. En effet, l'acteur C s'exécute après p exécutions de B. La consommation de C est de $p \cdot q$, où q peut avoir p valeurs différentes. q n'est alors pas correctement défini. Trois notions sont introduites par Fradet, Girault et Poplavko pour assurer que les périodes de changement d'un paramètre sont sûres : *l'influence*, les *régions* et les *itérations locales*.

Un paramètre p *influence* un arc e, noté $\mathrm{Infl}(e, p)$ s'il apparaît dans son rythme de production ou de consommation, ou dans l'équation d'équilibre de son acteur source ou puits. On définit une *région* d'influence (3.3) d'un paramètre comme l'ensemble des arcs influencés par ce paramètre. Dans notre exemple, la région d'influence de q est $\mathscr{R}(q) = \{(B, C)\}$, et les acteurs de cette région sont B et C.

$$\mathscr{R}(p) = \left\{ e \;\middle|\; \mathrm{Infl}(e, p) \right\} \tag{3.3}$$

Comme le calcul d'itération global ne permet pas d'exprimer le changement d'un paramètre p au cours d'une itération, *l'itération locale* définie une itération sur un sous-ensemble du graphe, qui a une période d'itération différente. L'itération locale $\#_L X$ d'un acteur est le nombre d'exécutions d'un acteur pour que le sous-ensemble du graphe $\{X_1, \ldots, X_n\}$ auquel il appartient revienne à son état initial. Elle est calculée pour chaque acteur X_i en divisant son itération globale $\#X_i$ par le plus grand commun diviseur des itérations des acteurs de la région d'influence (3.4). Pour revenir à l'exemple, l'itération locale dans la région de q est $B^p C$.

$$\#_L X_i = \frac{\#X_i}{\mathrm{pgcd}\left(\#X_1, \ldots, \#X_n\right)} \tag{3.4}$$

Pour être correcte, la période de changement d'un paramètre p doit être cohérente avec l'itération $\#X$ des acteurs de la région d'influence $\mathscr{R}(p)$ du paramètre. Pour cela, il est nécessaire que l'itération des acteurs soit un multiple de $\#M(p)/\alpha(p)$. Dans l'exemple, l'itération des acteurs dans la région d'influence de q doit être multiple de $\#M(q)/\alpha(q) = \#B/\alpha(q) = 3p/p = 3$. Cette condition est vérifiée avec $\#B = 3p$ et $\#C = 3$.

La décomposition en régions et sous-régions se fait de manière hiérarchique. Chaque région contient l'ensemble des sous-régions qui la composent. Ainsi, on peut vérifier qu'une

sous-région est correcte. Si c'est le cas, la région englobante répète une ou plusieurs fois cette sous-région, et la valeur de paramètre est toujours bien définie.

L'ensemble de ces analyses permet de nous assurer qu'un graphe SPDF est cohérent, malgré un changement de paramètres au cours d'une itération. Cependant, cette propriété augmente la complexité lors de l'ordonnancement. Pour cette raison, plusieurs travaux se basant sur SPDF limitent le modèle à un changement de valeur durant l'itération [Bebelis 13a, Bebelis 13b]. Dans la suite de ce travail, je travaille selon le même modèle avec un seul changement de paramètres par itération.

Bebelis *et al.* proposent un modèle de calcul dérivé de SPDF, nommé BPDF (*Boolean Parametric DataFlow*) [Bebelis 13a]. Ce modèle apporte, en plus des paramètres entiers, des paramètres booléens qui permettent d'activer ou de désactiver des parties du graphe. La valeur de ces paramètres booléens peut varier durant une itération. Les auteurs détaillent les analyses permettant de définir les périodes de variation de ces paramètres pour que l'exécution du graphe soit correcte. Dans leur modèle d'exécution, les auteurs résolvent le problème de l'ordonnancement en exécutant tous les acteurs systématiquement. Ceci est possible car la valeur du paramètre booléen définit si l'exécution est symbolique ou non.

La première partie de ce chapitre nous a permis d'étudier le modèle de calcul flot de données. Dans un premier temps, j'ai présenté ce modèle de manière générale, puis ses réalisations statiques et dynamiques. J'ai ensuite présenté de manière plus approfondie le modèle flot de données paramétrique SPDF sur lequel se base la suite de ce travail. Après avoir ainsi posé les fondements théoriques, je vais maintenant étudier les analyses rendues possibles par ce modèle, et plus particulièrement les méthodes d'ordonnancement existantes.

3.3 Ordonnancement d'acteurs flot de données

Un des intérêts des modèles de calcul flot de données est d'exposer toutes les dépendances d'un programme de manière explicite. Comme nous l'avons vu dans la section précédente, cela permet de s'assurer de l'absence d'interblocage et de l'exécution dans une taille de mémoire finie de notre programme. Une autre possibilité offerte par ces communications explicites est l'ordonnancement statique d'un programme. En effet, contrairement à d'autres modèles de calcul utilisés pour la programmation de systèmes embarqués (par exemple C + fils d'exécution), les modèles de calcul flot de données permettent cet ordonnancement durant la compilation. De cette manière, l'ordonnancement peut être optimisé à la compilation, et réduit le coût lié à un ordonnancement dynamique.

Cette section a pour objectif de mieux comprendre les enjeux et les limitations de l'ordonnancement de graphes flot de données. Je commencerai dans la Section 3.3.1 par l'ordonnancement de graphes statiques, sujet largement exploré et qui me permettra de fixer des notions d'ordonnancement. j'étudierai ensuite dans la Section 3.3.2 l'ordonnancement dans des graphes flot de données paramétriques, appliqué au modèle de calcul SPDF.

3.3.1 Ordonnancement d'acteurs statique

L'ordonnancement d'une application à plusieurs fils d'exécution, sans connaissance des dépendances de données entre les parties du programme, est réalisé de manière dynamique. Les tâches sont toutes exécutées en concurrence, le système d'exploitation se charge de les ordonnancer dynamiquement. Ce type de programmation est largement utilisé aujourd'hui et permet une grande expressivité. La communication entre les tâches est réalisée à l'aide de primitives de synchronisation de manière dynamique. Ces synchronisations ont un coût

temporel non négligeable, et l'ordonnancement dynamique n'ayant pas connaissance de ces dépendances, peut produire un ordonnancement peu efficace. De plus, il appartient au développeur de vérifier l'absence d'interblocage dû à des dépendances de données ou à une taille mémoire insuffisante.

Dans le cas d'un programme suivant un modèle de calcul flot de données, les dépendances sont explicites. Le compilateur peut tirer parti de ces informations pour générer un ordonnancement efficace, ainsi que selon le modèle vérifier l'absence d'interblocage dû aux communications. Cet ordonnancement peut être réalisé entièrement statiquement dans le cas d'une application flot de données statique (SDF), comme nous allons le voir dans cette section. Si au contraire l'application suit un modèle de calcul flot de données dynamique, les informations de dépendance peuvent être utilisées pour réaliser une partie de l'ordonnancement. Autrement, ces informations permettent de guider l'ordonnanceur dynamique. Nous verrons ce type d'approche dans la prochaine section.

L'ordonnancement d'un graphe flot de données revient à définir un ordre statique d'exécution des acteurs. Commençons par définir une grammaire formelle pour représenter cet ordonnancement avec la Définition 2.

Définition 2 *Un ordonnancement statique d'un graphe contenant les acteurs $v_1 \dots v_n$ est défini comme :*

$$sched \quad ::= \quad v \mid sched; sched \mid (sched)^x \quad où\ x \in \mathbb{N}^*\ et\ v \in \mathcal{V}$$

Pour illustrer les notions d'ordonnancement nous utiliserons la Figure 3.6. Un ordonnancement correct de ce graphe est $\left(A^2; (B^2; C)^3\right)$. Il se comprend comme : exécuter A deux fois, puis exécuter la séquence B deux fois suivi de C, et répéter cette séquence trois fois.

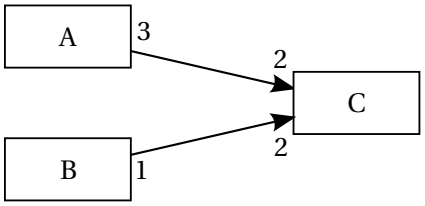


FIGURE 3.6 – Exemple de graphe SDF à ordonnancer.

Une approche naïve pour l'ordonnancement d'un graphe flot de données acyclique est le tri topologique. La méthode est de partir d'un acteur source du graphe, et d'exécuter cet acteur le nombre de fois requis pour son itération. Nous pouvons alors retirer l'acteur du graphe, et recommencer jusqu'à ce que tous les acteurs aient été exécutés. Cette méthode permet d'obtenir une SAS (*Single Apparence Schedule*), car chacun des acteurs est présent une seule fois dans l'ordonnancement. Pour l'exemple de la Figure 3.6, nous obtenons l'ordonnancement $\left(A^2; B^6; C^3\right)$ ou $\left(B^6; A^2; C^3\right)$. Cette méthode fonctionne pour tout graphe acyclique.

Dans le cas d'un graphe cyclique, il est nécessaire que ce graphe soit *faiblement connecté* [Bhattacharyya 99]. Un graphe est faiblement connecté si, pour chacun de ses cycles, le nombre de données initiales sur un arc est supérieur à la consommation cumulée sur un cycle de l'acteur connecté à cet arc, c'est à dire $i_{(A,B)} \geq r_{B,(A,B)} \cdot \#B$. Si cette condition est vérifiée,

nous pouvons ignorer l'arc contenant les données initiales dans le tri topologique, et l'appliquer sur le graphe acyclique obtenu. Dans l'exemple de la Figure 3.2 $i_{(C,A)} \geq r_{A,(C,A)} \cdot \#A$. Nous pouvons donc ignorer l'arc (C,A) dans le tri topologique, pour obtenir l'ordonnancement $\left(A;B^2;C^3\right)$.

L'ordonnancement par tri topologique génère une grande consommation mémoire. Cette consommation mémoire est définie comme le nombre de cases minimum nécessaire dans les FIFO reliant les acteurs pour l'exécution correcte du graphe. Pour le graphe de la Figure 3.6 avec un ordonnancement de $\left(A^2;B^6;C^3\right)$, nous obtenons une taille mémoire sur l'ensemble des arcs de 12 données. Nous pouvons toutefois réduire cette taille mémoire dans le cas d'un ordonnancement sous forme SAS en *factorisant* cet ordonnancement. Cette transformation est prouvée correcte par Bhattacharyya, Murthy et Lee [Bhattacharyya 96]. Elle consiste à factoriser le nombre d'itérations de plusieurs acteurs dans une boucle. Par exemple, l'ordonnancement $\left(A^2;B^6;C^3\right)$ peut être factorisé pour obtenir $\left(A^2;\left(B^2;C\right)^3\right)$. La factorisation permet de réduire la taille mémoire à 8 données, mais pas de la minimiser.

Une autre approche pour ordonnancer un graphe est l'exécution symbolique. Une exécution symbolique d'un graphe est une exécution dans laquelle on modélise uniquement le nombre de consommation et de production de données des acteurs dans le graphe. Ce modèle permet d'explorer les exécutions possibles d'un graphe de manière simplifiée. Il autorise l'optimisation selon divers objectifs dont la minimisation de la mémoire. Dans l'exemple du graphe de la Figure 3.6, un ordonnancement optimal en mémoire est $(A;B;B;C;A;B;B;C;B;B;C)$, ou plus simplement $\left(\left(A;B^2;C\right)^2;B^2;C\right)$.

L'exécution symbolique peut être réalisée à l'aide d'outils de *model checking* pour représenter le graphe. Dans les travaux de Geilen, Basten et Stuijk [Geilen 05], le langage PROMELA ainsi que le *model checker* SPIN sont utilisés pour minimiser la consommation mémoire d'un graphe flot de données. Cependant, cette approche est inapplicable pour des graphes de grande taille, l'ordonnancement d'un graphe flot de données étant un problème NP-complet. Les travaux de Liu *et al.* [Liu 09] proposent l'utilisation d'heuristiques associées au *model checking* afin de parer à ce problème.

Pour cette raison, de nombreuses heuristiques sont développées pour l'ordonnancement de graphes flot de données [Bhattacharyya 99, Bouakaz 13]. Je laisse le lecteur intéressé explorer la large recherche à ce sujet, et je décris simplement l'approche par regroupement proposé par Bhattacharyya [Bhattacharyya 99] à titre d'exemple. Cette heuristique cherche à maximiser la factorisation dans un ordonnancement. Pour cela, elle rassemble deux par deux les acteurs connectés en calculant le plus grand diviseur commun (PGCD) de leur itération. La paire qui possède le plus grand PGCD est choisie, à condition que ce nouvel acteur ne crée pas de cycle dirigé. Cette paire constitue un sous-graphe, vu comme un nouvel acteur hiérarchique dans la suite de l'algorithme. On applique sur le graphe résultant le même algorithme, jusqu'à obtenir un graphe ayant un seul acteur. Pour reprendre l'exemple de la Figure 3.6, le PGCD de chacune des paires d'acteurs est $\{A,C\} = 1$ et $\{B,C\} = 3$. On rassemble donc les acteurs B et C, puis les deux acteurs restants, pour obtenir l'ordonnancement $\left(A^2;\left(B^2;C\right)^3\right)$.

Toutes les méthodes d'ordonnancement présentées dans cette section se limitent à un seul ordre statique d'exécution. Comme vu dans le Chapitre 2, la majorité des plateformes matérielles sont multicœurs. Pour cette raison, un ordonnancement unique n'est pas satisfaisant, et il est nécessaire d'étudier l'ordonnancement d'acteurs sur plusieurs unités de calcul. Avec la multiplication des ressources de calcul, la tentation est grande de réserver une unité de calcul par acteur. Le risque est alors de sous-utiliser une grande partie des ressources de la

plateforme matérielle, tout en créant des goulots d'étranglement sur d'autres ressources. Pour cette raison, nous allons étudier l'ordonnancement d'acteurs sur des architectures multicœurs avec partage de ressources.

Une première méthode envisagée durant ce travail de thèse consiste à dériver l'ordonnancement multicœurs depuis un ordonnancement monocœur [Dardaillon 14b]. Cet ordonnancement monocœur peut être obtenu à l'aide d'une méthode décrite précédemment. Pour cette approche, faisons l'hypothèse d'un placement connu des acteurs sur les différentes unités de calcul. Nous pouvons alors séquencer chacun des acteurs sur son unité associée, dans l'ordre dans lequel l'acteur est séquencé dans l'ordonnancement global. Le respect de cet ordre permet de conserver des propriétés de l'ordonnancement initial, dont l'absence d'interblocage. La conservation de ces propriétés repose sur le modèle d'exécution associé. Ce modèle doit permettre une synchronisation entre producteur et consommateur, pour éviter un accès concurrent critique à la mémoire.

L'ordonnancement généré initialement ne tient pas compte du contexte d'exécution multicœurs, ce qui limite les possibilités d'optimisation. Le principal atout de cette méthode est de proposer un algorithme simple pour générer un ordonnancement correct. Dans la pratique, je l'associe à un ordonnancement SAS pour la génération de l'ordonnancement utilisé dans la suite de mes travaux.

Afin d'optimiser l'ordonnancement en tenant compte du contexte d'exécution multicœurs, de nouveaux algorithmes sont proposés dans la littérature. Tout d'abord, un modèle de la plateforme multicœurs peut être associé à l'exécution symbolique pour optimiser l'ordonnancement selon différents objectifs [Moreira 12]. En plus de l'optimisation de la mémoire, le temps d'exécution global peut être optimisé en parallélisant l'exécution entre les différents cœurs de calcul [Bouakaz 13]. Le problème de l'ordonnancement peut être divisé en sous-parties pour réduire sa complexité [Feautrier 06]. Les travaux de Gu *et al.* [Gu 10] sur le langage CAL proposent d'ordonnancer les parties statiques du graphe à l'aide de l'algorithme de regroupement d'acteurs présenté précédemment [Bhattacharyya 99], et l'étendent aux architectures multicœurs. Nous noterons que dans l'approche multicœurs, l'ordonnancement et le placement des acteurs sont des problèmes liés. Ces deux problèmes étant NP-complets, nous trouvons dans la littérature des heuristiques pour chacun d'entre eux afin de les résoudre de manière itérative [Bouakaz 13, Castrillon 13].

3.3.2 Extension à l'ordonnancement d'acteurs paramétriques

La section précédente m'a permis de dresser un tour d'horizon des méthodes d'ordonnancement pour le modèle flot de données statique (SDF). Les travaux développés durant cette thèse visent le modèle SPDF décrit dans la Section 3.2.3. Pour cela, je vais maintenant me concentrer sur les travaux permettant d'ordonnancer un graphe paramétrique.

L'intérêt du modèle SPDF réside dans l'équilibre qu'il propose entre les possibilités d'analyse et le dynamisme introduit par l'utilisation de paramètres. Une des analyses proposées dans l'article de Fradet, Girault et Poplavko [Fradet 12] est la génération d'ordonnancement quasi-statique pour un graphe SPDF. Un ordonnancement quasi-statique est un ordonnancement réalisé essentiellement statiquement, il laisse uniquement l'ordonnancement non décidable à la compilation être réalisé à l'exécution. Dans le cas d'un graphe paramétrique, le nombre de répétitions d'un acteur peut être paramétrique. La Définition 3 formalise la grammaire d'un ordonnancement quasi-statique. Dans cette grammaire, v représente un l'acteur à exécuter, $\text{set}_v(p)$ la production d'un paramètre p par l'acteur v, $\text{get}_v(p)$ la consommation

d'un paramètre p par l'acteur v.

Définition 3 *Un ordonnancement quasi-statique d'un graphe contenant les acteurs $v_1 \dots v_n$ est défini comme :*

$$sched \quad ::= \quad v \mid set_v(p) \mid get_v(p) \mid sched; sched \mid (sched)^p \quad où\ p \in \mathscr{F}\ et\ v \in \mathscr{V}$$

L'algorithme proposé par Fradet, Girault et Poplavko permet de générer un ordonnancement quasi-statique par acteur, ceci dans le cas général où le paramètre peut changer plusieurs fois dans la même itération et avec plusieurs paramètres imbriqués. Si l'on reprend l'exemple de la Figure 3.5, nous obtenons les ordonnancements suivants pour chacun des acteurs : $\big(A; \text{set}(p)\big)$, $\Big(\text{get}(p); \big(B^p; \text{set}(q)\big)^3\Big)$ et $\Big(\text{get}(p); \big(\text{get}(q); C\big)^3\Big)$. Nous omettons de préciser l'acteur qui produit ou consomme le paramètre en l'absence d'ambiguïté.

Cet ordonnancement permet de fixer la synchronisation des paramètres grâce à des FIFO, de manière similaire aux données. Cependant, cette méthode produit un ordonnancement par acteur, ce qui n'est pas satisfaisant dans l'objectif de partage de ressources. L'ordonnancement de plusieurs acteurs dans un seul ordonnancement quasi-statique est rendu difficile par le changement de valeur des paramètres au cours d'une itération. Les travaux de Bhattacharyya [Bhattacharya 01] se heurtent à un problème similaire avec le modèle PSDF. Dans celui-ci l'ordonnancement quasi-statique ne peut pas être entièrement déterminé à la compilation. Je limite donc le changement de valeur des paramètres à la fin d'une itération pour permettre la génération d'ordonnancement quasi-statique.

La condition d'un changement de valeur par itération permet de réutiliser la méthode du tri topologique pour créer un ordonnancement séquentiel. En effet, tous les paramètres sont fixés durant l'itération, le graphe peut être analysé avec les valeurs symboliques. Si le graphe possède un cycle dirigé, il faut appliquer la méthode vue pour les graphes statiques dans la section précédente pour supprimer ce cycle. Reprenons l'exemple de la Figure 3.5 en fixant le changement de valeur du paramètre q une fois par itération, soit set $q[3p]$. Nous obtenons alors l'ordonnancement global $\big(A; \text{set}(p); B^{3p}; \text{set}(q); C^3\big)$. Dans celui-ci, je suppose que la valeur d'un paramètre produit par un acteur, set(p), est réutilisée par les acteurs suivants dans l'ordonnancement.

La question de l'ordonnancement multicœurs avec partage de ressources se pose de la même manière que pour le flot de données statique. La solution proposée précédemment pour partager un ordonnancement unique sur plusieurs cœurs peut être étendue à l'ordonnancement paramétrique. En effet, la production et la consommation de paramètre sont associées à un acteur. Nous pouvons donc ordonner cette production ou consommation sur l'unité de calcul associée à l'acteur concerné, selon l'ordre global. La production et consommation de données et de paramètres permettent la synchronisation entre les unités de calcul.

Mon approche de l'ordonnancement de graphe SPDF se distingue par l'utilisation d'un ordonnancement quasi-statique. Cette approche ne permet pas d'optimiser l'ordonnancement en fonction de l'architecture, mais permet néanmoins de grandement réduire l'ordonnancement dynamique. Nous verrons dans le Chapitre 6 sur les expérimentations que le contrôle de l'application dans ses parties dynamiques est un élément critique sur la plateforme Magali. Cela justifie cette approche utilisée lors de la génération du code de contrôle dans le Chapitre 5.

Les travaux sur le modèle BPDF [Bebelis 13a] utilisent un ordonnancement quasi-statique par acteur. L'ordonnancement global sur une plateforme multicœurs est réalisé à l'exécution selon une méthode d'ordonnancement par *créneau*. À chaque créneau, tous les acteurs qui disposent de suffisamment de données sont ordonnancés sur les unités de calcul disponibles. Durant l'exécution du créneau, un nouvel ordonnancement est généré en se basant sur la connaissance des productions et consommations des acteurs.

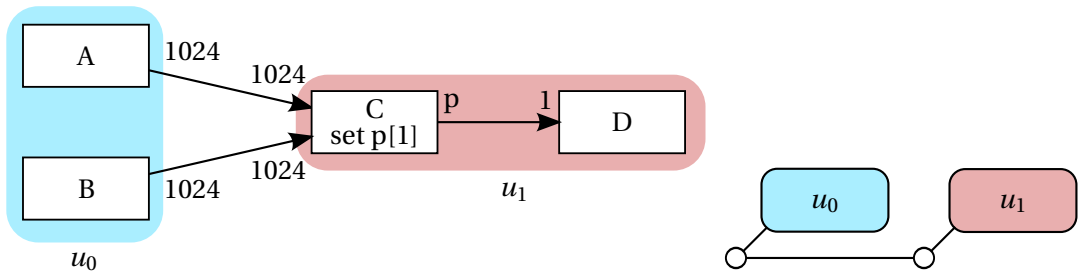
Les ordonnancements présentés jusqu'à maintenant se sont basés sur le modèle flot de données, modèle qui suppose la consommation et la production de l'ensemble des données par un acteur de manière atomique. Dans la prochaine section, je relâche cette hypothèse afin d'analyser plus finement la consommation mémoire d'une application flot de données.

3.4 Micro-ordonnancement de communications flot de données

Dans cette section, je présente un raffinement de l'ordonnancement des communications dans un graphe flot de données développé durant cette thèse, et que je nomme micro-ordonnancement. Je commencerai par décrire les motivations qui ont poussé à développer ce nouveau paradigme dans la Section 3.4.1. Je le définirai formellement dans la Section 3.4.2. Je décrirai ensuite son application au contrôle des tailles mémoires dans un graphe flot de données dans la Section 3.4.3.

3.4.1 Motivation

Un grand nombre d'ordonnancements présentés dans la Section 3.3 cherchent à minimiser la consommation mémoire ou le temps d'exécution sur des plateformes multicœurs. Pour cela, ces ordonnancements reposent sur la représentation flot de données pour déterminer une exécution parallèle des acteurs qui minimise un critère donné. Cependant, la représentation flot de données abstrait l'exécution d'un acteur. Elle modélise toutes les consommations et les productions de données d'un acteur comme une seule opération atomique. Cette abstraction à l'intérêt de permettre des analyses à haut niveau comme vu précédemment, mais elle présente toutefois des limites pour l'analyse du comportement des acteurs.



(a) Application (b) Architecture

FIGURE 3.7 – Exemple d'application flot de données placée sur une plateforme à deux cœurs.

Pour comprendre ces limites, je vous propose de nous plonger dans l'exemple de la Figure 3.7. J'ajoute la notion d'architecture et de placement pour illustrer cet exemple, tel que défini dans la Définition 4. Dans cet exemple, le graphe flot de données composé des acteurs $\mathcal{V} = \{A, B, C, D\}$ est placé sur une architecture $\mathcal{U} = \{u_0, u_1\}$ selon le placement $m(A, B) = u_0$ et $m(C, D) = u_1$.

Définition 4 *Un graphe flot de données placé sur une architecture est défini comme :*
- *— $\mathcal{G}$ est un graphe connecté $(\mathcal{V},\mathcal{E})$ avec $\mathcal{V}$ un ensemble d'acteurs et $\mathcal{E} \subseteq \mathcal{V} \cdot \mathcal{V}$ un ensemble d'arcs dirigés,*
- *— $\mathcal{U}$ un ensemble d'unités de calculs ;*
- *— $m : \mathcal{V} \rightarrow \mathcal{U}$ le placement d'un acteur sur une unité de calcul.*

La Figure 3.8 représente une itération de ce graphe sur l'architecture décrite selon le modèle flot de données. L'axe des ordonnées représente les unités de calcul. L'exécution des acteurs sur une unité de calcul est représentée en abscisse, les arcs qui relient les acteurs représentent les communications. L'hypothèse de consommation et de production de l'ensemble des données telle que formulée par le modèle flot de données séquentialise partiellement l'exécution du graphe. L'acteur C doit attendre l'exécution des acteurs A *et* B. Nous avons dans cet exemple une latence sur une itération égale à la somme des temps d'exécution de A, B, C et D, et une mémoire requise sur l'architecture égale à la somme des productions de A et B, soit 2048 (on suppose les valeurs possibles de p comprises entre 1 et 4).

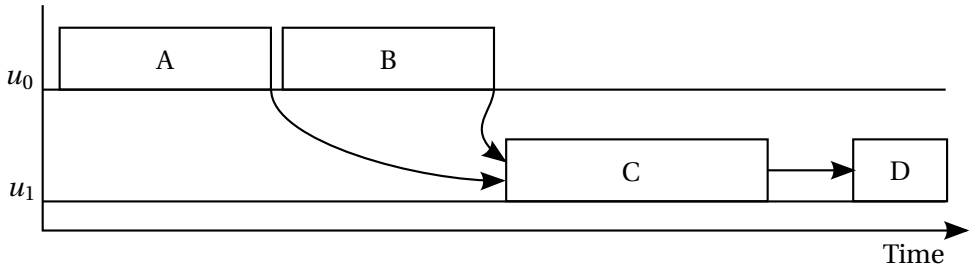


FIGURE 3.8 – Chronogramme d'une application selon le modèle flot de données.

Ces hypothèses sont tout à fait valides dans certains domaines d'applications. L'exécution de l'itération suivante sur une unité de calcul permet de conserver le parallélisme. Ici l'unité u_0 peut exécuter l'itération $n+1$ de A et B pendant que l'unité u_1 exécute l'itération n de C et D. Cependant, le domaine étudié durant cette thèse, les télécommunications, pose des contraintes de temps de réponse qui n'autorisent pas une telle latence durant une exécution. D'autre part, la mémoire représente un coût important dans une architecture matérielle, tant en termes de surface que de consommation. L'objectif de basse consommation pousse les constructeurs à minimiser la taille mémoire, ce qui rend les hypothèses de travail du modèle flot de données difficiles à adapter au domaine d'étude.

Je représente sur la Figure 3.9 l'exécution du graphe avec communications à grain fin entre les acteurs. L'exécution des acteurs A et B est entrelacée, je suppose que l'acteur C associe les données des acteurs A et B. Nous observons le plus grand nombre de communications entre les acteurs. Ces communications sont de plus petite taille que sur le modèle précédent pour prendre en compte la faible quantité de mémoire présente sur la plateforme. Cela crée une synchronisation fine entre les différents acteurs, et nous observons un large recouvrement entre l'exécution des acteurs sur l'unité de calcul u_0 et u_1.

Cet exemple illustre la différence entre une exécution selon le modèle flot de données et une exécution avec communications à grain fin plus réaliste pour une plateforme visant les télécommunications. Au delà des gains de performances pour le type de plateforme visé, il

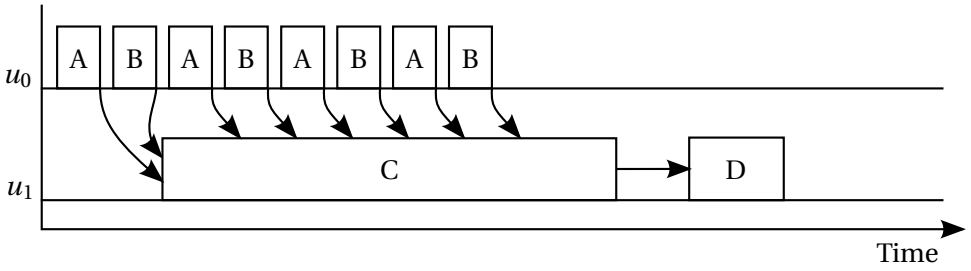


FIGURE 3.9 – Chronogramme d'une application avec communications à grain fin.

faut retenir de cet exemple que le modèle de calcul flot de données propose une modélisation des communications non adaptée aux plateformes matérielles visées. Afin de pouvoir réaliser une analyse plus réaliste des applications de télécommunications, je propose de compléter la modélisation flot de données en représentant les communications entre les acteurs non pas comme un événement atomique, mais comme une suite de productions et de consommations de données. J'introduis cette représentation dans la prochaine section.

3.4.2 Définition et exemple

Le concept central du micro-ordonnancement est de représenter de manière explicite chacune des communications durant l'exécution d'un acteur. Je propose de représenter ces communications comme une séquence de productions et de consommations sur les arcs d'un acteur. Cette définition des communications permet d'intégrer le micro-ordonnancement dans l'ordonnancement flot de données sous la forme d'un nouveau niveau hiérarchique.

Définition 5 *Un micro-ordonnancement de graphe flot de données paramétrique est défini comme :*

$$sched \quad ::= \quad micro \mid get_v(p) \mid sched; sched \mid (sched)^p \quad où\ p \in \mathscr{F}\ et\ v \in \mathscr{V}$$

$$micro \quad ::= \quad push_e(p) \mid pop_e(p) \mid set_v(p) \mid n\,?\,micro \mid micro; micro \mid (micro)^p$$
$$où\ n, p \in \mathscr{F}, v \in \mathscr{V}\ et\ e \in \mathscr{E}$$

La Définition 5 redéfinit l'ordonnancement d'un graphe flot de données paramétrique ainsi que le micro-ordonnancement des communications dans un acteur. Le micro-ordonnancement *micro* remplace l'exécution de l'acteur dans l'ordonnancement. Les calculs réalisés par les acteurs ne sont pas représentés intentionnellement, car le micro-ordonnancement se concentre sur les communications. Le micro-ordonnancement, au même titre que l'ordonnancement, permet de représenter une séquence $micro_1; micro_2$ ainsi que de répéter cette séquence un nombre paramétrique de fois $(micro)^p$. L'opérateur $push_e(p)$ (resp. $pop_e(p)$) désigne la production (resp. consommation) de p données sur l'arc e de manière atomique (un paquet indivisible de p données). Si cette production (resp. consommation) est divisible, elle est représentée comme la répétition d'une production (resp. consommation) unitaire $push_e(1)^p$ (resp. $pop_e(1)^p$). La notation n ? *micro* désigne l'exécution d'un micro-ordonnancement à condition que ce soit la $n^{ème}$ exécution de l'itération.

Nous remarquons que seule la production de paramètres est intégrée dans le micro-ordonnancement, contrairement à la consommation. En effet, un paramètre est produit par un acteur au cours de son exécution. La production de ce paramètre appartient donc à l'acteur représenté par le micro-ordonnancement. Si l'acteur n'est exécuté qu'une seule fois, l'inclusion de la production du paramètre est évidente. Si l'acteur est exécuté plusieurs fois durant une itération, la notation n ? permet de préciser durant quelle exécution il doit produire ce paramètre. Généralement le paramètre est produit au début ou à la fin d'une itération (1 ? ou $\#X$?), mais les autres cas sont permis. Cette notation fait l'hypothèse d'un moyen de connaître la position de l'exécution courante dans l'itération. Elle est nécessaire pour représenter la périodicité x de production d'un paramètre p, notée set $p[x]$.

La consommation d'un paramètre est définie dans l'ordonnancement pour deux raisons. Lorsqu'un acteur est exécuté un nombre paramétrique de fois (ex. $\left(\text{get}(p); X^p\right)$), la consommation du paramètre est faite de manière explicite durant l'itération. Lorsque l'acteur utilise la valeur du paramètre durant son exécution, par exemple pour une production de données paramétrique, l'accès au paramètre pourrait être placé dans le micro-ordonnancement. Cependant, le rythme de production d'un paramètre est fixé par l'acteur produisant ce paramètre. Pour cette raison, la consommation de paramètre est placée dans l'ordonnancement et non le micro-ordonnancement. On dissocie donc la synchronisation imposée par l'ordonnancement de l'utilisation du paramètre par l'acteur.

Illustrons le micro-ordonnancement à l'aide de l'exemple de la Figure 3.7. Un exemple de micro-ordonnancement des acteurs est donné par (3.5). Dans cet exemple, les acteurs A et B produisent leurs données par blocs de 256. L'acteur C entrelace les consommations de données provenant de A et B, toujours en bloc de 256 données. L'ordre de consommation de l'acteur C est une information importante pour l'ordonnancement des acteurs. Celle-ci était auparavant masquée par l'abstraction du modèle flot de données. Elle permet de déterminer l'ordonnancement des acteurs A et B pour minimiser le temps d'exécution et la mémoire utilisée.

$$\begin{gathered}\mu S_A = \left(\text{push}_{A,(A,C)}(256)^4\right) \qquad \mu S_B = \left(\text{push}_{B,(B,C)}(256)^4\right) \qquad \mu S_D = \left(\text{pop}_{D,(C,D)}(1)\right) \\ \mu S_C = \left(\left(\text{pop}_{C,(A,C)}(256); \text{pop}_{C,(B,C)}(256)\right)^4; \text{set}(p); \text{push}_{C,(C,D)}(p)\right)\end{gathered} \tag{3.5}$$

La spécification du micro-ordonnancement, contrairement à l'ordonnancement, ne se fait pas à partir du graphe, mais à partir du comportement des acteurs. Dans les travaux actuels, cette spécification est réalisée manuellement. Cependant, elle gagnerait à être automatisée afin de refléter au plus près le programme et la plateforme associée. Dans le cas d'un acteur implémenté sur une unité de calcul programmable, l'analyse du logiciel permettrait de retrouver le micro-ordonnancement. Dans le cas d'un acteur implémenté sur un accélérateur matériel, ce micro-ordonnancement est associé à la plateforme et pourrait être inclus dans le modèle de cette plateforme.

$$\begin{gathered}\mu S_{u_0} = \left(\text{push}_{A,(A,C)}(256); \text{push}_{B,(B,C)}(256)\right)^4 \\ \mu S_{u_1} = \left(\left(\text{pop}_{A,(A,C)}(256); \text{pop}_{B,(B,C)}(256)\right)^4, \text{set}(p); \text{push}_{C,(C,D)}(p); \text{get}(p); \text{pop}_{D,(C,D)}(1)^p\right)\end{gathered} \tag{3.6}$$

À partir du micro-ordonnancement des acteurs et leur placement sur la plateforme, je définis un micro-ordonnancement par unité de calcul (3.6). Ce nouveau micro-ordonnancement

doit respecter l'ordre défini dans le micro-ordonnancement de chacun des acteurs, ainsi que l'ordre imposé par la topologie du graphe. Si cette condition n'est pas respectée, alors un acteur pourrait produire une donnée ou un paramètre avant d'avoir consommé les données permettant de le calculer. Le micro-ordonnancement obtenu comprend bien l'ensemble des communications de la Figure 3.9. Cette nouvelle représentation permet de raffiner les analyses réalisables sur un graphe flot de données. Dans la prochaine section, je propose de l'appliquer au contrôle des tailles mémoires.

3.4.3 Application au contrôle des tailles mémoires

Le micro-ordonnancement permet de modéliser les communications entre les acteurs durant une itération, et raffine ainsi le modèle flot de données en explicitant chaque production et consommation d'un acteur. Dans un graphe flot de données, les communications entre les acteurs passent par des FIFO, représentées par les arcs du graphe. Nous avons vu dans la section 3.3 que l'ordonnancement d'un graphe flot de données permet de connaître la quantité de mémoire nécessaire sur chacun des arcs, et donc des tailles des FIFO dans une implémentation. Je propose de raffiner cette modélisation de la consommation mémoire à l'aide du micro-ordonnancement. Cette proposition vise à réduire la quantité de mémoire nécessaire à l'exécution d'un graphe tout en garantissant l'absence d'interblocage dû aux communications.

Cette nouvelle approche vient répondre à un besoin d'implémenter des applications flot de données sur des plateformes matérielles embarquées fortement contraintes en mémoire, dans cette thèse des plateformes de radio logicielle. Pour illustrer cette problématique, reprenons l'exemple de la Figure 3.7 ainsi que le micro-ordonnancement (3.6). L'acteur C entrelace la consommation des données de A et B. Le micro-ordonnancement de l'unité de calcul u_0 permet de réduire la taille mémoire de la FIFO entre les unités de calcul à 256 données par arc, au lieu de 1024 données par arc selon le modèle de calcul flot de données. Si au contraire l'ordonnancement de l'unité de calcul u_0 est $S_{u_0} = \left(\text{push}_{A,(A,C)}(256)^4; \text{push}_{B,(B,C)}(256)^4\right)$, alors la quantité de mémoire minimale nécessaire est de 1280 données. Sur une plateforme fortement contrainte possédant une mémoire limitée à 512 données, notre approche apporte la garantie d'absence d'interblocage, jusque là impossible à obtenir avec le modèle flot de données. Cette garantie permet de réduire le temps de développement en mettant à profit des informations de haut niveau. Elle évite un débogage potentiellement long et complexe sur la plateforme.

Le contrôle des tailles mémoires demande de modéliser un ensemble d'éléments. Tout d'abord, il est nécessaire de modéliser le comportement de chacun des acteurs à l'aide de son micro-ordonnancement. Comme nous venons de le voir, l'ordonnancement de ces acteurs sur les unités de calcul a une grande influence, il est aussi nécessaire de le modéliser. Enfin, il est nécessaire de modéliser la plateforme matérielle pour tenir compte de l'agencement et de la taille de ces mémoires.

Pour porter cette modélisation, je propose d'utiliser le langage PROMELA. Ce langage de vérification est utilisé dans l'outil de *model checking* SPIN. Le *model checking* consiste à représenter un programme sous la forme d'une machine d'états. À chaque nœud correspond un état du système, ici le nombre de données dans chaque FIFO. À chaque transition correspond une intruction, qui fait évoluer le système vers un nouvel état. Ici, une instruction est une production ou consommation de données ou de paramètre. Le *model checker* explore la machine d'états jusqu'à la vérification d'une formule de logique temporelle. C'est cette formule qui fixe l'objectif de vérification, par exemple une taille mémoire ou un état d'interblocage.

Ce type de modélisation est déjà utilisé pour l'ordonnancement de graphes, comme nous l'avons vu dans la section 3.3. Le modèle utilisé dans cette thèse est dérivé des travaux de Geilen, Basten et Stuijk [Geilen 05], qui modélise un graphe flot de données pour l'ordonnancer en fonction des tailles mémoires. Leur modèle suit le modèle flot de données qui utilise une exécution atomique de chaque acteur. Dans mon travail, le modèle d'acteur atomique est remplacé par l'exécution séquentielle du micro-ordonnancement.

```
#define PUSH(c,n) atomic{ch[c]+n <= max[c] -> ch[c] = ch[c]+n;}
#define POP(c,n) atomic{ch[c] >= n -> ch[c] = ch[c]-n;}
#define SET(c,v) p[c]!v;
#define GET(c,v) p[c]?v;

int ch[3];  int max[3];  chan p[1]=[1] of {int};


proctype Unit0(){
  do :: {
    int i;
    for(i:1..4) {
      PUSH(0,256);
      PUSH(1,256);
    }
  } od
}

init{
  max[0] = 256;
  max[1] = 256;
  max[2] = 4;
  atomic{
    run Unit0();
    run Unit1();
  }
}
```

```
proctype Unit1(){
  do :: {
    int i,p_c,p_d;
    for(i:1..4) {
      POP(0,256);
      POP(1,256);
    }
    select(p_c:1..4);
    SET(0,p_c);
    PUSH(2,p_c);
    GET(0,p_d);
    for(i:1..p_d) {
      POP(2,1);
    }
  } od
}

ltl deadlock {[]<> np_}
```

FIGURE 3.10 – Modèle PROMELA du graphe de la Figure 3.7.

La Figure 3.10 illustre la syntaxe du modèle PROMELA de l'exemple de la Figure 3.7, selon le micro-ordonnancement (3.6). Chaque processus **proctype** modélise une unité de calcul avec son micro-ordonnancement. Dans le code présenté, `Unit0` modélise l'unité de calcul u_0. L'exécution de `Unit0` produit des données sur l'arc (A,C) puis (B,C) ; la boucle **for** permet de répéter cette opération 4 fois. L'unité u_1 est modélisé par `Unit1`. Son exécution consomme les données sur les arcs (A,C) et (B,C) 4 fois, génère le paramètre p, produit p données sur l'arc (C,D) et consomme p fois 1 donnée sur l'arc (C,D).

Durant l'exploration des états du code PROMELA, le tableau d'entier `ch` encode le nombre de données présent dans les FIFO, et `max` la taille mémoire de chacune des FIFO. Les macros `PUSH(c,n)` et `POP(c,n)` modélisent la production et la consommation de données dans une FIFO. L'exécution de `PUSH(c,n)` (c.-à-d. produire `n` données dans la FIFO `c`) contrôle la disponibilité en mémoire dans la FIFO `c` pour ajouter `n` données, et ajoute les données

si possible. Le contrôle de taille mémoire est bloquant, le processus est arrêté jusqu'à ce que suffisamment de mémoire soit disponible. La macro `POP(c,n)` (c.-à-d. consommer `n` données dans la FIFO `c`) contrôle si au moins `n` données sont disponibles dans la FIFO `c`, et les consomme si possible. Le contrôle de disponibilité est aussi bloquant. Pour réduire la complexité du modèle, la valeur des données échangées n'est pas représentée. J'utilise un entier qui modélise uniquement le nombre de productions et de consommations sur une FIFO.

La modélisation des paramètres requiert de connaître leur valeur, pour connaître la consommation d'un acteur par exemple. Les canaux de communication **`chan`** modélisent les FIFO de transmission des paramètres. Les macros `SET(c,v)` et `GET(c,v)` modélisent la production et la consommation de paramètres. `SET(c,v)` (c.-à-d. produire un paramètre de valeur `v` dans la FIFO c) contrôle la disponibilité de mémoire dans la FIFO `c` pour ajouter une valeur de paramètre, la produisant dès que possible. `GET(c,v)` (c.-à-d. consommer un paramètre dans la variable `v` depuis la FIFO `c`) vérifie la disponibilité d'un paramètre dans la FIFO `c`, et le copie dans la variable `v` dès qu'il est disponible. Les macros `SET(c,v)` et `GET(c,v)` bloquent également le processus jusqu'à ce que la place pour produire une donnée ou que la donnée soit disponible. La méthode `select(p:range)` produit des valeurs dans tout l'intervalle `range`. Elle permet d'explorer toutes les valeurs possibles lors d'un parcours de la machine d'états.

Finalement, l'exécution correcte du programme est vérifiée par logique linéaire temporelle (**`ltl`**). L'objectif est d'éviter les interblocages dus aux communications. La condition de logique linéaire temporelle nommée `deadlock` permet de contrôler que le programme est toujours (`[]`) finalement (`<>`) dans un état de progrès (`np_`), c'est à dire qu'au moins un processus est en progression. En utilisant cette condition, le *model checking* est mis au défi de trouver un parcours valide de la machine d'état qui termine par un état de non progrès, c'est à dire d'interblocage. À titre d'exemple, la vérification de l'exemple de la Figure 3.10 est quasiment immédiate (moins d'une seconde). La machine d'état générée par l'outil pour la vérification de l'application est constituée de 640 états et 859 transitions.

J'ai présenté dans cette section un raffinement de la représentation des communications pour les applications flot de données. Cette représentation apporte de nouvelles informations pour l'ordonnancement efficace d'applications ainsi que pour la vérification d'absence d'interblocage dû à des tailles mémoires insuffisantes. Pour évaluer cette méthode de vérification avec des applications réalistes, la Section 6.3.1 présentera les résultats de vérification d'applications flot de données extraites du protocole LTE.

3.5 Discussion

Ce chapitre a permis de présenter le modèle de calcul flot de données, l'ordonnancement de graphes flot de données ainsi qu'une nouvelle représentation des communications dans ce modèle de calcul. Le modèle de calcul flot de données statique SDF est présenté dans un premier temps. Les besoins de dynamisme des nouvelles applications en télécommunications ou décodage vidéo ont poussé au développement d'évolutions dynamiques du modèle flot de données, avec les différentes familles de modèles présentées. Dans ces différents modèles, le modèle de calcul flot de données paramétrique SPDF a été choisi pour la suite des expérimentations.

La seconde partie a permis d'introduire la problématique de l'ordonnancement dans les graphes flot de données. L'ordonnancement d'une application offre l'opportunité d'optimiser

un programme selon différents objectifs de consommation mémoire ou de temps d'exécution. Ceci est rendu possible par le formalisme flot de données. En effet, il permet des analyses qui apportent des garanties sur la consommation mémoire ou l'absence d'interblocage. L'apport du dynamisme vient modifier cet équilibre entre les possibilités d'analyses et l'expressivité du modèle. Une difficulté introduite dans l'ordonnancement est la prise en compte d'un environnement d'exécution parallèle. Dans ce cadre, je propose une méthode simple pour étendre un ordonnancement à l'exécution sur plusieurs cœurs. Cette méthode fournit un ordonnancement correct, et repose sur le placement pour utiliser au mieux la plateforme.

L'ordonnancement sur une plateforme multicœurs ouvre la possibilité de tirer parti du parallélisme de données et de tâches présent dans un graphe flot de données. Cependant, la modélisation du temps d'exécution des différents acteurs à ordonnancer est un exercice périlleux. La concurrence entre les cœurs ainsi que la hiérarchie mémoire rendent cette évaluation difficile au moment de la compilation. L'autre approche, l'ordonnancement dynamique d'acteurs, peut représenter un surcoût temporel indésirable lors de l'exécution, ou ne pas être supporté par certaines plateformes. Pour ces raisons, l'ordonnancement statique ou quasi-statique est encore d'actualité. Des travaux émergeants visent à atténuer ce problème en analysant les performances d'applications flot de données à l'exécution, afin de générer un nouvel ordonnancement statique qui tient compte des performances réelles de l'application [Tan 09]. Cette approche permettrait de répondre à la complexité de l'évaluation des performances à la compilation tout en conservant un ordonnancement statique.

La troisième partie de ce chapitre introduit une nouvelle représentation de l'ordonnancement des communications pour les graphes flot de données développée durant cette thèse, le micro-ordonnancement. Cette représentation modélise chaque communication ayant lieu dans un graphe flot de données. Elle est motivée par l'inadéquation entre les tailles de mémoire dans les architectures de télécommunications et le modèle mémoire des applications flot de données. La modélisation de chaque communication entre acteurs réduit l'abstraction et rapproche le modèle de l'exécution réelle de l'application. Cette modélisation fine des communications augmente la complexité du modèle, mais donne des informations importantes pour le domaine d'application. Ces informations permettent de raffiner l'ordonnancement et la vérification d'absence d'interblocage dans une application.

Une nouvelle représentation de l'ordonnancement ainsi que du micro-ordonnancement qui utilise le langage PROMELA est proposée. Associée à l'outil de *model checking* SPIN, cette nouvelle modélisation permet de vérifier l'absence d'interblocage dû à l'ordonnancement des communications sur une application flot de données en fonction de la mémoire présente sur une plateforme. En particulier, elle permet de modéliser et vérifier des applications flot de données sur des plateformes fortement contraintes en mémoire et qui ne pouvaient être vérifiées jusqu'à maintenant.

Cette approche est encore aujourd'hui limitée. D'une part, les communications sont modélisées comme une séquence de productions et consommations de données. Cette hypothèse n'est pas vérifiée pour des applications telles que le H.264. En effet, ce type d'application consomme un nombre statique de données sur plusieurs ports, mais ces consommations se font selon un ordre différent en fonction des données pour des besoins d'optimisation. D'autre part, il est nécessaire d'évaluer les performances de cette modélisation sur des applications réelles. Cette évaluation a été réalisée sur des applications de télécommunications, et les résultats seront présentés dans le Chapitre 6. Au final, ce nouveau modèle est un premier pas, qui ouvre la vérification d'applications à des plateformes fortement contraintes.

4 Front End : du format source à la représentation intermédiaire

Remind me of that black fella back home who fell off a ten-story building. As he was falling, people on each floor kept hearing him say, "So far, so good". Heh, so far, so good.

— Steve McQueen as Vin, *The Magnificent Seven*

Sommaire du chapitre

4.1 Introduction à la compilation d'application flot de données

Dans le chapitre précédent, j'ai présenté le modèle de calcul flot de données paramétrique SPDF, son ordonnancement ainsi que les analyses possibles sur ce modèle. J'ai de plus introduis un nouveau formalisme, le micro-ordonnancement, qui permet de nouvelles analyses sur l'ordonnancement et la consommation mémoire. Dans ce chapitre et le suivant, je mets en œuvre ces propriétés dans un nouveau flot de compilation. Une vue d'ensemble de ce flot de compilation est présentée sur la Figure 4.1.

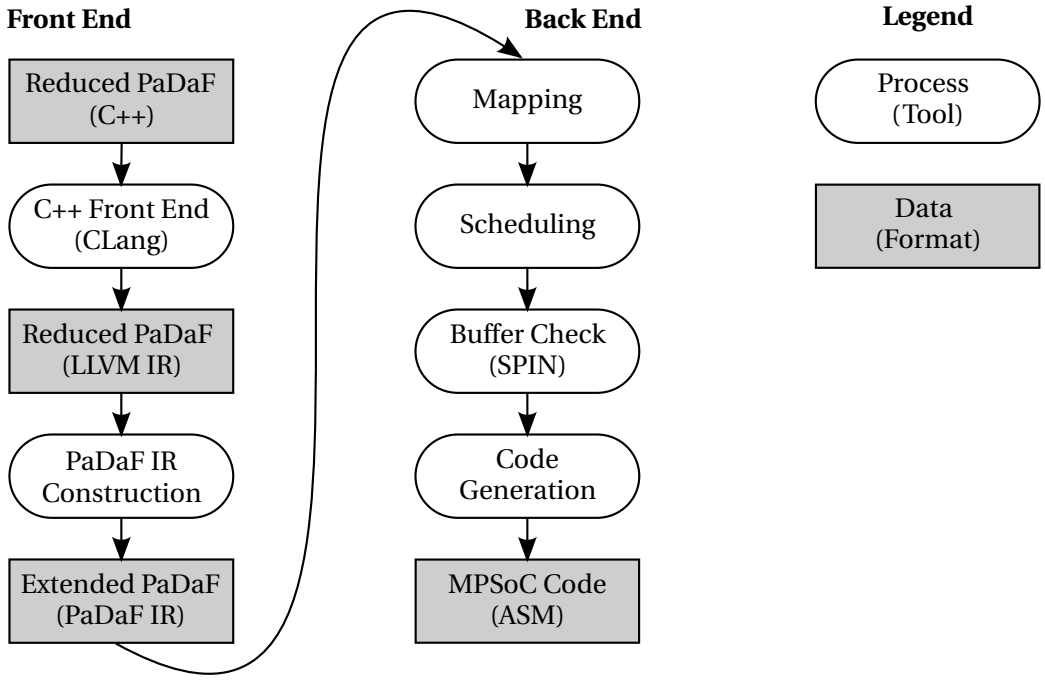


FIGURE 4.1 – Vue d'ensemble du flot de compilation.

Le flot de compilation est découpé en deux parties, le *Front End*, présenté dans ce chapitre, et le *Back End*, présenté dans le chapitre suivant. Avant de plonger dans la description du *front end*, je décris les notions abordées dans ces deux chapitres. La partie *front end* d'un compilateur permet de transformer le code source d'une application en une représentation intermédiaire (IR). L'IR d'un compilateur est commune à tous les langages en entrée du compilateur, et aussi indépendante de la cible du compilateur. Les optimisations sont réalisées sur l'IR, de manière indépendante du langage et de la plateforme. Une fois optimisée, l'IR est spécialisée par le *back end* pour une plateforme donnée. Dans le cadre de ce travail, le format d'entrée PaDaF réduit sera défini dans la Section 4.2. La transformation de ce format d'entrée vers l'IR PaDaF étendu sera présenté dans la Section 4.3. La spécialisation vers la plateforme cible, autrement appelée *back end*, sera présentée dans le Chapitre 5.

Les notions de format réduit et étendu correspondent à la représentation du graphe dans ces formats. Le format réduit contient la spécification du graphe à construire sous la forme d'un programme C++. Le format étendu contient le graphe construit avec l'ensemble de ces acteurs instanciés, dans un objet mémoire. En absence d'ambiguïté, je ferai référence au format PaDaF dans la suite du manuscrit.

Une information importante lorsque l'on parle d'un compilateur est le temps de compilation. En effet, la compilation peut avoir lieu à plusieurs moments :

— En compilation statique, l'application est compilée sur une machine hôte, puis exécutée sur une machine cible. La machine cible peut être différente de la machine hôte, on parle alors de compilation croisée. Comme la compilation a lieu à l'avance, elle peut réaliser des optimisations complexes, qui sont trop coûteuses à réaliser lors de l'exécution.
— En compilation juste à temps (JIT), l'application est compilée à la demande lors de son exécution sur la machine cible. Le JIT peut être réalisé à la granularité de la fonction, et permet de tirer parti du contexte d'exécution pour optimiser le code.
— En compilation dynamique, tout ou parti du binaire de l'application peut être recompilé durant l'exécution pour se spécialiser selon le contexte d'exécution.

Dans le cadre de ce travail, le compilateur proposé réalise une compilation croisée statique de l'application PaDaF sur la machine hôte. Le code binaire résultant est ensuite chargé sur la cible : une plateforme de radio logicielle. Ce choix est motivé par les contraintes d'une plateforme de radio logicielle, tant en mémoire qu'en performance et en consommation mémoire. La combinaison de plusieurs de ces approches permettrait d'apporter les bénéfices de la compilation dynamique à la radio logicielle. Par exemple, Cohen et Rohou [Cohen 10] proposent de combiner l'analyse poussée de la compilation statique avec la compilation dynamique pour tirer parti du contexte d'exécution, dans le cadre d'applications embarquées. Ces travaux sont cependant hors du cadre de cette thèse.

Cette introduction a permis de dessiner le cadre général du compilateur présenté. Je présente dans la section suivante les contraintes sur le langage et le flot de compilation, suivi d'une discussion sur les travaux existants au vu de ces contraintes.

4.1.1 Contraintes

Les contraintes posées dans le choix du langage et du flot de compilation sont liées ; Elles sont présentées dans cette section avant de discuter des travaux existants répondant à ces contraintes dans la section suivante.

Tout d'abord, le langage doit permettre d'exprimer une application selon le modèle de calcul flot de données paramétrique SPDF. Il est donc nécessaire de pouvoir représenter un acteur. Un acteur est composé d'un ou plusieurs ports de données en entrée et en sortie, ainsi que de ports de paramètres en entrée et en sortie. Un acteur possède une fonction de traitement sur les données. Cette fonction doit pouvoir consommer et produire des données et des paramètres. Le langage doit aussi permettre d'exprimer la construction d'un graphe, c'est à dire la connexion d'acteurs entre eux pour communiquer des données ou des paramètres.

Le domaine des télécommunications offre des contraintes supplémentaires sur l'expressivité du langage. En effet, les applications dans ce domaine sont potentiellement grandes, impliquant un grand nombre d'acteurs, et peuvent posséder une structure régulière. Prenons par exemple le récepteur MIMO simplifié de la Figure 4.2. Ce récepteur nécessite autant de chaînes de traitement qu'il y a d'antennes en réception, 4 dans l'exemple. La construction manuelle de ces chaînes s'avère fastidieuse et source d'erreurs. Le langage peut remédier à cette difficulté de deux manières. L'utilisation de la hiérarchie dans la construction du graphe permet de reproduire un sous-graphe commun aux différentes chaînes, factorisant l'effort de construction du sous-graphe. L'utilisation de structures de contrôle (ex. boucle `for`) permet de programmer la construction d'un graphe complexe, automatisant la construction de

graphes à structure régulière.

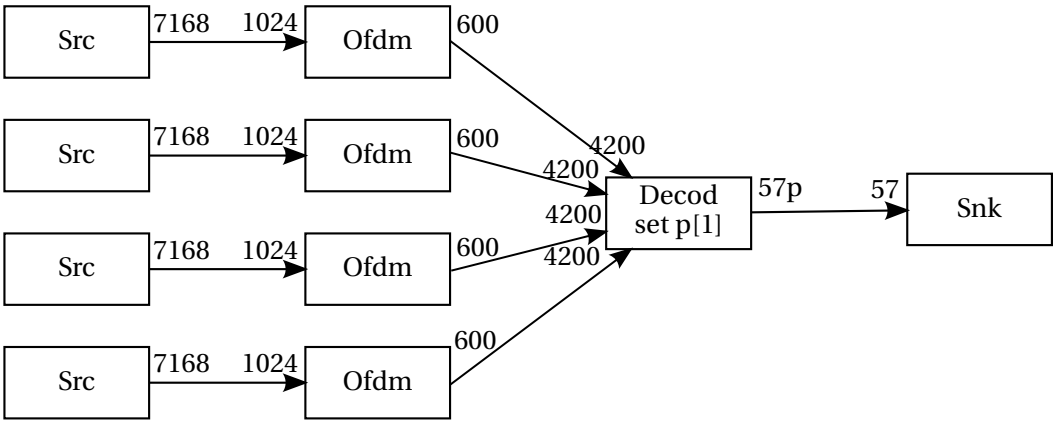


FIGURE 4.2 – Exemple de récepteur MIMO simplifié.

Une fois le langage défini, il est nécessaire d'avoir un flot de compilation supportant ce langage. La première contrainte sur cet outil vient du modèle de calcul. Comme nous l'avons vu dans le chapitre précédent, les modèles de calcul flot de données permettent de réaliser un grand nombre d'analyses sur le graphe de l'application. Mais pour cela, la topologie du graphe doit être statique, c'est à dire que tous les acteurs ainsi que les arcs les connectant doivent être connus à la compilation. Il est donc nécessaire que le graphe soit statique et dans une forme exploitable par le compilateur, ceci afin de pouvoir l'analyser puis l'optimiser durant la compilation.

D'autre part, il semble nécessaire d'utiliser un compilateur existant. En effet, le développement d'une nouvelle chaîne de compilation demande un effort colossal. Au vu de l'effort de développement effectué sur les outils existants, tant en termes d'optimisation, de plateformes supportées, de débogage que d'environnement, il semble illusoire de vouloir recréer un compilateur de toute pièce. Pour cela, je détaille dans la prochaine section plusieurs langages présentés dans l'état de l'art du Chapitre 2 et significatifs dans le domaine, et motive ensuite mon choix.

4.1.2 Travaux existants

StreamIt Le langage flot de données StreamIt [Thies 02] est développé au MIT, et se base sur un modèle de calcul équivalent à SDF. Un graphe StreamIt est composé de filtres et de primitives de construction pour structurer le graphe. Un filtre peut être un acteur effectuant un traitement, ou un sous-graphe hiérarchique. Les primitives de construction sont le *Pipeline* qui connecte plusieurs acteurs en chaîne, le *SplitJoin* qui sépare le flot de données en plusieurs branches puis les fusionne, et la *FeedBackLoop* est une boucle de rétroaction.

La structuration du langage StreamIt permet d'augmenter les possibilités d'analyse et d'optimisation du graphe obtenu, mais limite fortement l'expressivité du langage. Si l'on reprend l'exemple de la Figure 4.2, toutes les sources devraient être rassemblées en un seul filtre, avec une structure *SplitJoin* pour séparer le traitement des différentes antennes. Il faut donc multiplexer toutes les sources sur un seul canal, ce qui rend le langage contre productif et l'optimisation d'une telle application plus complexe. Une telle structuration n'est donc pas

souhaitable dans un langage destiné au domaine de la radio logicielle.

ΣC Le langage de programmation ΣC [Goubier 11] du CEA utilise un modèle de calcul équivalent à CSDF. Un intérêt de ce langage est de définir la topologie du graphe dans une section *map* exécutée lors de la compilation. Cette section permet d'instancier des acteurs et de les connecter en utilisant les structures de contrôle du langage C. Ce langage permet de créer des topologies de graphes complexes facilement, rendant l'approche intéressante pour les télécommunications. L'implémentation de ce langage est cependant propriétaire, et ne peut donc pas être réutilisée. De plus, ΣC a été pensé comme un nouveau langage, il nécessite la création d'une nouvelle chaîne de compilation complète.

CAL Actor Language Le langage CAL [Bhattacharyya 08, Lucarz 08] permet l'implémentation de graphes flot de données, indépendamment du modèle de calcul. Les acteurs sont décrits dans un langage spécifique impératif. Le graphe est décrit sous un format XML, ce qui permet l'implémentation multiple d'acteurs ainsi que la hiérarchisation en sous-graphes. Le langage CAL est utilisé pour la spécification de décodage vidéo, et fait partie intégrante du standard ISO/MPEG. De plus, il est utilisé par le flot de compilation ouvert ORCC (*Open RVC-CAL Compiler* [Gorin 10].

CAL est donc un excellent candidat pour implémenter une radio logicielle. Cependant, nous avons choisi de ne pas l'utiliser pour plusieurs raisons. Tout d'abord, le langage ne supporte pas à l'heure actuelle l'utilisation de paramètres variant à l'exécution, condition nécessaire de développement de nos travaux autour du modèle de calcul SPDF. D'autre part ces travaux visent la plateforme Magali, présentée dans la Section 5.2, qui n'est pas supportée par le flot de compilation ORCC.

J'ai donc choisi dans la suite de mes travaux de développer un nouveau format ainsi qu'une nouvelle chaîne de compilation. L'objectif n'est cependant pas de re-développer toute un flot de compilation. J'ai donc fait le choix d'adapter des éléments existants à la compilation d'applications flot de données paramétrique pour la radio logicielle. Le format source se base sur le langage C++, sans modification de sa syntaxe, ce qui permet de réutiliser un *front end* existant. La construction du graphe suit une approche similaire à celle de ΣC, en se basant sur les structures de contrôle du C++ pour la construction de graphes complexes. Ce format sera présenté dans la Section 4.2.

L'utilisation du langage C++ permet de s'appuyer sur des outils existants en compilation. Le flot de compilation développé dans cette thèse utilise l'infrastructure de compilation LLVM. Cette infrastructure supporte plusieurs langages en entrée dont le C++. Je présenterai LLVM et les adaptations au *front end* existant pour le support du modèle de calcul flot de données paramétrique dans la Section 4.3.

4.2 Format source

Dans cette section, je présente le format PaDaF. Ce format s'appuie sur le langage C++, en particulier sur la notion d'objet présent dans ce langage. J'introduis les notions générales du langage dans la Section 4.2.1. Je décris ensuite la représentation des acteurs dans la Section 4.2.2, suivi de la représentation du graphe dans la Section 4.2.3.

4.2.1 Langage

Le langage C++ est utilisé comme base pour le format PaDaF. Ce langage est largement répandu pour le développement d'applications et de systèmes, y compris dans le domaine de l'embarqué. Il constitue une base connue pour le développeur. De plus, il permet de réutiliser du code C ou C++ existant. Ce choix permet d'adapter de manière graduelle une base de code existant vers un format flot de données.

Comme nous l'avons vu en introduction, de nombreux langages flot de données proposent une syntaxe modifiée du langage C pour porter les informations sur le langage flot de données. Ce choix implique un nouveau flot de compilation, ou au moins des modifications importantes qui impactent l'ensemble d'un flot de compilation existant. J'ai donc fait le choix de travailler avec des outils existants, et d'utiliser la notion de classes pour représenter mon graphe flot de données. En effet, l'abstraction par classes permet au langage C++ de supporter différents modèles de calcul.

Un exemple répandu qui utilise cette approche est SystemC [Panda 01]. SystemC comprend un ensemble de classes C++ qui permettent de représenter une architecture logicielle et matérielle, et un ordonnanceur en vu de la simuler. Il rend possible la simulation de MPSoC depuis le niveau transactionnel jusqu'au niveau électronique. Les principales classes en SystemC sont : les *modules* pour structurer une architecture ; les *ports* pour communiquer entre modules, au travers de *canaux* ; les processus pour réaliser les calculs. La communication se fait au travers des canaux, mais aussi par des événements temporels ou discrets.

Un programme SystemC permet donc de représenter une architecture complexe possédant son propre modèle de calcul, au travers de l'utilisation de classes. Un programme SystemC peut être compilé statiquement par un compilateur C++, sans autre modification. Une fois compilé, le programme est associé à un *runtime* pour l'exécution, afin de simuler le système programmé.

Je propose d'utiliser une approche similaire, avec l'utilisation de classes pour représenter les acteurs et les connections entre ces acteurs. La principale différence avec SystemC est le modèle de calcul flot de données paramétrique utilisé dans mon approche. Un tel programme peut être simulé à l'aide d'un *runtime*, de manière similaire à SystemC. Cette approche permet de vérifier en simulation le programme. Le programme peut aussi être compilé vers une plateforme de radio logicielle pour son exécution en temps réel, qui est la visée de ce travail.

4.2.2 Représentation des acteurs

L'exemple de la Figure 4.3 permet d'illustrer un exemple simple d'acteur en PaDaF. Cet exemple correspond a une implémentation possible des acteurs Ofdm de la Figure 4.2. En premier lieu, tous les acteurs sont des classes dérivées de la classe de base `Actor`. Cette classe de base permet de spécifier que la classe actuelle est un acteur. Cette information est utilisée lors de la construction de l'architecture, comme nous le verrons dans la Section 4.3. Dans notre exemple, la classe `Ofdm` est donc dérivée de la classe `Actor`.

Un acteur contient des ports d'entrée et de sortie, `PortIn` et `PortOut`. Ces ports permettent de communiquer entre les différents acteurs. Le type de données communiqué est défini par un mécanisme de *template*. Le problème du typage de données est un problème complexe ; Ce problème n'est pas étudié dans ce travail, les données sont définies comme des entiers **`int`**. Le nombre de données consommées ou produites est fixé dans le constructeur. Dans l'exemple de la Figure 4.3, le port d'entrée `Iin` consomme `1024` données. Le constructeur permet de spécifier le noms des objets (port, acteurs, etc.). Les noms dans l'exemple sont

```
class Ofdm : public Actor {
  public:
    PortIn<int> Iin;
    PortOut<int> Iout;
    void compute();
    Ofdm()
      :Actor("Ofdm",trx_ofdm_20),Iin("in",1024),Iout("out",600){}
};
```

FIGURE 4.3 – Exemple de l'acteur OFDM en PaDaF.

similaires aux noms des variables pour plus de simplicité. La dernière indication donnée au constructeur est le placement de l'acteur sur la plateforme. En effet, comme nous le verrons dans le chapitre suivant, le placement des acteurs est actuellement réalisé manuellement. Ici, l'acteur est placé sur l'unité de calcul `trx_ofdm_20`.

```
class Decod : public Actor {
  public:
    std::vector<PortIn<int>*> Iin;
    PortOut<int> Iout;
    ParamOut p;
    void compute();
    Decod()
      :Actor("Decod", dmdi_23s), p("p", 1), Iout("out", p*57) {
      for(int i=0; i<NB_ANT; i++) {
        Iin.push_back(new PortIn<int>("in", 4200));
      }
    }
};
```

FIGURE 4.4 – Exemple de l'acteur Decod en PaDaF.

Dans le cas d'une application flot de données paramétrique, la consommation ou production d'un acteur peut aussi être un produit d'entiers ou de paramètres. L'exemple de la Figure 4.4 illustre cette possibilité. L'acteur Decod de cet exemple est aussi un exemple d'implémentation de la Figure 4.2. L'objet `ParamOut p` est le paramètre produit par cet acteur. Il est utilisé dans la production du port de sortie `Iout`, qui est de `p*57`. Cet acteur possède aussi un vecteur de ports d'entrées `Iin`, vecteur instancié dans le constructeur de l'acteur par la méthode `push_back`.

Cet exemple illustre l'expressivité de l'approche utilisé pour le langage, et précise aussi ses limites. En effet, la spécification et la construction du graphe peuvent être réalisées avec toute l'expressivité du C++. Cela permet par exemple de créer un vecteur de ports d'entrées instancié durant la construction de l'acteur par une structure arbitrairement complexe, de calculer le nombre d'itération lors de la compilation du graphe, de lire des paramètres sur la structure du graphe dans un fichier de configuration, etc. La seule condition que l'on fixe à ce langage est que la spécification des acteurs et du graphe doit aboutir à une topologie statique. Cette condition permet l'analyse du graphe durant la compilation selon les règles du modèle

de calcul flot de données. La construction du graphe par exécution de sa spécification est une solution utilisée dans le langage ΣC. Contrairement à ΣC, je propose dans ce travail de réutiliser des outils existants pour mettre en place cette construction, et décrit la méthode utilisée dans la Section 4.3.

```
void Decod::compute() {
  [...]
  int coef = NB_ANT * var;
  for(int i=0; i<NB_ANT; i++) {
    val[i] = Iin[i]->pop();
    [...]
  }
  p.set(size);
  Iout.push(res, 57*size);
}
```

FIGURE 4.5 – Exemple de traitement de l'acteur Decod en PaDaF.

Une fois la structure de l'acteur fixée, il est nécessaire de connaître le traitement que celui-ci doit réaliser. Pour cela, une fonction `compute()` est associée à chacun des acteurs. Cette fonction, illustrée sur la Figure 4.5 pour l'acteur Decod, contient le code de l'acteur à exécuter. La fonction `compute()` à besoin d'accéder aux données. Pour cela, les méthodes `pop()` et `push()` permettent respectivement de consommer et produire des données. L'accès aux données peut se faire de manière unitaire, voir le `pop()` en exemple, ou par paquets, voir le `push()` en exemple. Aucune contrainte n'est fixée sur la consommation et la production de données. Le développeur est responsable de consommer un nombre de données cohérent avec la spécification de l'acteur.

En plus de consommer et de produire des données, les acteurs d'un graphe flot de données paramétrique peuvent faire de même avec les paramètres. Dans l'exemple, une nouvelle valeur du paramètre `p` est produite par la méthode `set()`. De manière similaire, un acteur peut consommer une valeur de paramètre pour l'utiliser dans son calcul à l'aide de la fonction `get()`. Les paramètres sont produits et consommés de manière unitaire.

Le traitement d'un acteur peut être spécifié de deux manières, par un algorithme C++ et par l'utilisation d'une API de traitement du signal. Le format PaDaF s'appuie sur le langage C++ pour spécifier les acteurs et leurs interconnexions, tout comme les traitements à réaliser. Le langage C++ est un langage généraliste qui doit permettre de spécifier tout type de traitement. De cette manière, le format PaDaF garantit l'évolution vers des traitements non envisagés lors de la conception du format. Contrairement à l'usage d'une API dédiée, le C++ ne peut pas s'exécuter sur des unités de calcul dédiées. Pour cette raison, une API dédiée au traitement du signal est intégrée dans le format PaDaF. Cette API rend le format PaDaF spécifique au domaine d'application. L'utilisation d'une API est une solution répandue en radio logicielle. Le format CPN proposé par Castrillon *et al.* [Castrillon 11] se base sur la même dualité. Le langage C est utilisé comme base pour exprimer les traitements de manière générique. La bibliothèque *Nucleus* exprime les traitements sous la forme de blocs indépendants de la plateforme (*nuclei*), ainsi que leur implémentation sur des unités de traitements (*flavor*).

L'utilisation conjointe d'un langage expressif et d'une API spécifique à un domaine d'application permet de répondre à la problématique de la radio logicielle. Le langage C++ rend le

format extensible vers de nouveaux protocoles. L'API apporte l'implémentation efficace de traitements sur des accélérateurs dédiés. La difficulté dans cette approche est de proposer une API réellement indépendante de la plateforme, tout en réalisant une implémentation efficace sur chaque plateforme visée. En effet, chaque plateforme possède des unités de calcul dédiées avec leurs propres caractéristiques. Par exemple, la granularité des traitements réalisés par une unité peut varier selon la plateforme, ou les paramètres du traitement tels que la taille d'une FFT peuvent être variables ou fixes. Je présente une solution à la spécialisation des traitements et la démontre sur la plateforme Magali dans les Sections 5.3 et 5.4.

4.2.3 Représentation du graphe

La deuxième étape, après avoir créé les acteurs, est de les connecter pour former le graphe de l'application flot de données. La Figure 4.6 illustre la construction du graphe de la Figure 4.2. Cette construction est définie dans la fonction `main()` de l'application. Les acteurs sont instanciés en tant qu'objets lors de l'exécution de la fonction `main()`. C'est le cas des acteurs `decod` et `snk` dans l'exemple. Ils peuvent aussi être instanciés indirectement, avec comme exemple la classe `Antenna`. Cette classe permet de hiérarchiser la construction du graphe, et contient les acteurs `src` et `ofdm` d'une branche de réception.

```
class Antenna {
  private:
    Src src;
    Ofdm ofdm;
  public:
    PortOut<int>* Iout;
    Antenna() {
      ofdm.Iin <= src.Iout;
      Iout = & ofdm.Iout;
    }
};
```

```
int main (void) {
  Antenna ant[NB_ANT];
  Decod decod;
  Sink snk;
  for(int i=0; i<NB_ANT; i++) {
    decod.Iin[i] <= ant[i].Iout;
  }
  snk.Iin <= decod.Iout;
  buildGraph();
}
```

FIGURE 4.6 – Exemple de construction de graphe en PaDaF.

Une fois instanciés, les acteurs doivent être connectés entre eux pour former le graphe. Le format PaDaF utilise la notation `sink <= source`, similaire à celle utilisée dans les langages de description matérielle comme le VHDL. Cette notation permet de connecter un port de sortie vers un port d'entrée. Tout comme pour l'instanciation des acteurs, cette connexion peut se faire de manière simple, comme dans la connexion entre `decod` et `snk`. Elle peut aussi être faite de manière itérative à l'aide d'une boucle **`for`**, ou de manière hiérarchique comme dans l'exemple avec la classe `Antenna`.

Cet exemple illustre toute l'expressivité gagnée par l'utilisation du langage C++ associé à un ensemble de classes. Il est ainsi possible de construire le graphe de manière itérative et hiérarchique, ainsi que tout autre méthode imaginable dans le langage C++. La seule condition est, pour rappel, que le graphe final doit avoir une topologie statique et connue lors de la compilation.

Une fois l'ensemble des acteurs instanciés et connectés, la fonction `buildGraph()` est appelée. Cette fonction marque la fin de la construction du graphe. Elle est similaire à la

fonction `sc_start()` utilisée en SystemC pour démarrer la simulation. La prochaine section détaille la première étape de la compilation, c'est à dire la construction et l'analyse du graphe flot de données de l'application.

4.3 Construction de la représentation intermédiaire du graphe flot de données

Cette section traite de la construction de la représentation intermédiaire (IR) utilisée par le compilateur. L'apport principal de ce travail par rapport à l'état de l'art en compilation est la construction du graphe flot de données à partir du code C++ à compiler, et ceci durant le processus de compilation. Ce travail implique à la fois des analyses de l'IR durant la compilation et aussi, de manière beaucoup moins orthodoxe, à *l'exécution* de l'IR durant la compilation. J'explique dans cette section l'approche utilisée au travers de ces différentes étapes.

La construction du graphe présentée dans cette section est tirée des travaux de Marquet et Moy [Marquet 10] sur PinaVM, un *front end* pour l'analyse de modèles SystemC. Il reprend le processus de construction d'architecture et le lien avec l'IR LLVM. Je présente cette approche et son adaptation à la compilation d'application flot de données dans les prochaines sections. Le lecteur intéressé trouvera plus de détails sur l'analyse de l'IR et la construction des liens dans l'article original.

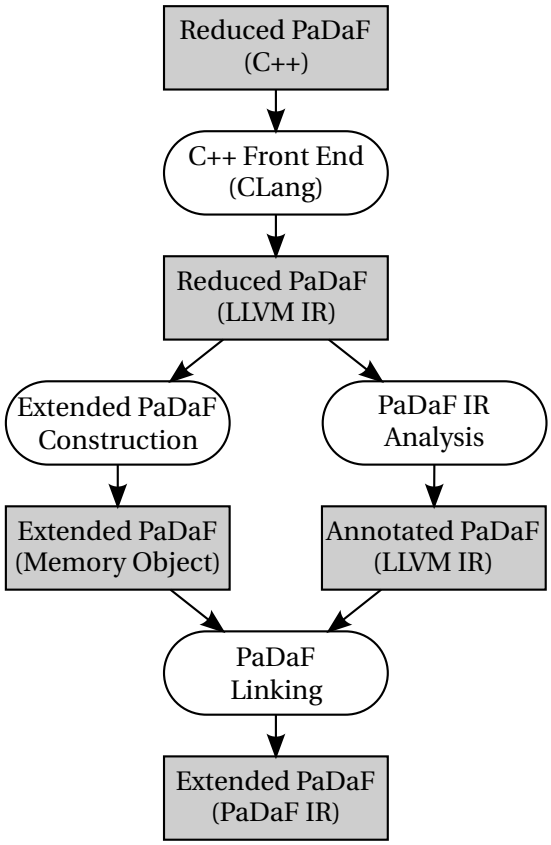


FIGURE 4.7 – Détail du *front end* du flot de compilation.

La Figure 4.7 présente une vue détaillée du *front end*. Le compilateur se base sur l'infrastructure de compilation LLVM, présentée dans la Section 4.3.1. La construction du graphe (*Extended PaDaF construction*) est présentée dans la Section 4.3.2, suivi de l'analyse de l'IR (*PaDaF IR analysis*) dans la Section 4.3.4, et la liaison de l'IR avec le graphe (*PaDaF linking*) dans la Section 4.3.4. L'infrastructure mise en place offre de nouvelles possibilités d'optimisation, que je présente dans la Section 4.3.5.

4.3.1 Infrastructure de compilation LLVM

Le compilateur développé durant cette thèse se base sur l'infrastructure de compilation LLVM. LLVM est un projet initialement développé à l'Université de l'Illinois pour l'expérimentation sur les techniques de compilation dynamique. Il a évolué vers une infrastructure de compilation importante, supportée par l'industrie. À titre d'exemple, les outils de développement d'Apple sont entièrement basés sur LLVM. Le projet LLVM est de plus entièrement *open-source*, ce qui garantit sa pérennité et la pertinence du choix pour un démonstrateur.

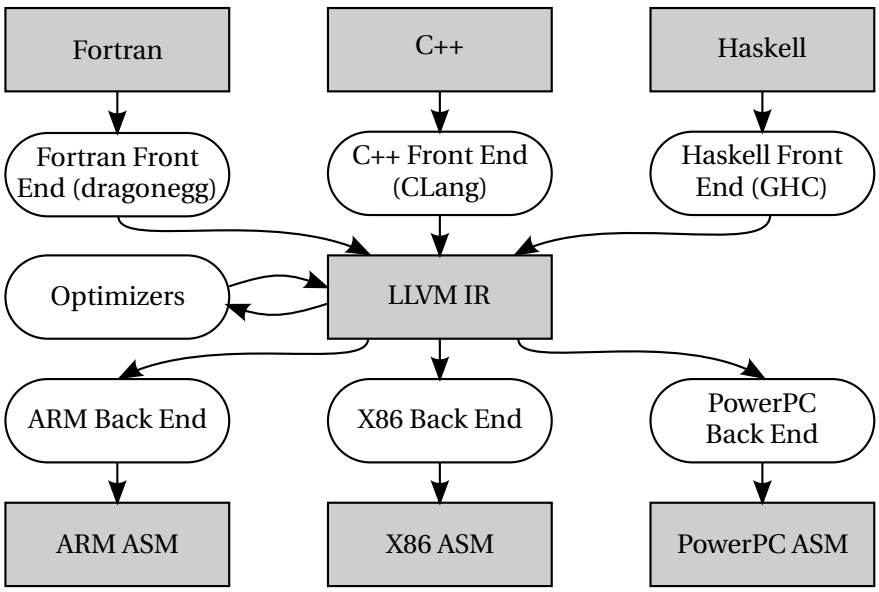


FIGURE 4.8 – Vue d'ensemble du flot de compilation LLVM.

La Figure 4.8 présente une vue générale du flot de compilation LLVM. LLVM est construit comme une infrastructure de compilation extensible et centrée autour de la représentation intermédiaire LLVM (IR LLVM). Dans ce flot de compilation, un *front end* transforme un langage donnée vers l'IR LLVM. Les optimisations sont réalisées sur l'IR LLVM, commune à tous les langages. Depuis cette représentation commune, un *back end* génère du code pour une plateforme donnée. L'intérêt de cette structure réside dans son extensibilité. L'ajout d'un nouveau langage nécessite uniquement le développement d'un *front end* pour bénéficier de toutes les optimisations et cibles supportées par LLVM. Réciproquement, l'ajout d'une nouvelle cible tire parti de l'ensemble des langages et de leurs optimisations uniquement via le développement d'un nouveau *back end*.

L'IR LLVM est une représentation bas niveau à assignation unique (SSA) ; C'est une représentation unique qui peut être vue sous 3 formes différentes :

— une IR en mémoire exploitable par un compilateur, sur lequel s'appuient les analyses et la génération de code ;
— un bytecode exécutable par un compilateur juste à temps (JIT), qui permet d'exécuter des parties du code durant la compilation ;
— un assembleur lisible par le développeur, utilisé pour le débogage durant le développement, et pour présenter les concepts dans ce travail.

Cette représentation est typée statiquement, avec une affection à gauche. Par exemple, l'expression `%res = add i32 %val, 2` additionne la variable `val` et la constante 2, et stocke le résultat dans la variable `res`. Elle est présentée par la suite dans une version simplifiée pour faciliter la compréhension.

Le flot de compilation présenté reprend plusieurs éléments de l'infrastructure LLVM. Le *front end* Clang transforme le programme PaDaF en IR LLVM pour analyse. Les optimisations de LLVM sont utilisées tout au long du flot de compilation. Les *back ends* existants sont conservés pour la génération de code, comme nous le verrons dans le chapitre suivant avec le *back end* ARM. À cela sont ajoutées des étapes spécifiques à la construction d'une IR du graphe flot de données, introduites dans la Figure 4.7 et maintenant détaillées.

4.3.2 Construction du graphe flot de données

La première phase de la compilation est la construction du graphe flot de données. Cette phase s'apparente à une *élaboration* dans les langages de description d'architecture tels que SystemC ou VHDL. Elle vise à construire une architecture, ici un graphe, à partir d'une spécification, ici le programme au format PaDaF. Ce programme est préalablement transformé en IR LLVM par le *front end* Clang.

La construction du graphe flot de données se fait donc à partir de l'IR LLVM. Une analyse du code pour reconstruire le graphe à partir de sa spécification semble voué à l'échec. Pour illustrer la complexité potentielle, j'utilise la Figure 4.6 qui présente le programme de construction du graphe du récepteur MIMO simplifié. Cette construction implique une boucle **`for`** pour la liaison des acteurs Decod et Snk. On peut dérouler cette boucle pour faciliter l'analyse de ces liaisons. Elle utilise aussi une construction hiérarchique avec la classe `Antenna`, construction beaucoup plus difficile à analyser statiquement. Cet exemple démontre que, à moins de limiter fortement l'expressivité du langage de construction du graphe, il n'est pas possible de l'analyser statiquement.

Pour cette raison, je propose d'exécuter le code de construction du graphe durant le processus de compilation. Ce code instancie l'ensemble des acteurs et les connecte entre eux. Le résultat de cette exécution est un ensemble d'objets C++ instanciés en mémoire ; Ces objets constituent le graphe étendu de l'application. Le compilateur récupère cette représentation du graphe en mémoire pour la suite de la compilation. De cette manière, il est possible de construire le graphe étendu à partir du format PaDaF réduit avec des outils existants, et en conservant la large expressivité du format PaDaF réduit.

Cette opération est réalisée à partir de l'infrastructure de compilation LLVM. L'IR LLVM possède une forme exécutable par un compilateur JIT, compilateur intégré dans l'infrastructure LLVM. Ce JIT est utilisé dans notre cas pour exécuter la fonction `main()` de construction du graphe. Une fois construit, il est nécessaire de récupérer le graphe en mémoire. La fonction `buildGraph()` placée à la fin de la construction du graphe appelle le compilateur, qui

récupère le graphe en mémoire.

Les classes du format PaDaF utilisées sont conçues pour fournir une représentation mémoire du graphe exploitable par le compilateur. Cette représentation mémoire inclut chaque acteur instancié, ainsi qu'un lien vers sa fonction de traitement en IR LLVM. Elle comprend aussi les connexions entre ces acteurs, et leur rythme de production et de consommation statique ou paramétrique, sous forme symbolique. Une fois le graphe construit, un autre problème se pose : comment relier le graphe aux fonctions de traitement, et plus spécifiquement aux communications sur le graphe ?

4.3.3 Analyse de la représentation intermédiaire LLVM

Chaque acteur possède une fonction de traitement ; Elle définit les consommations de données, les opérations réalisées sur ces données, et la production des résultats. Les production et les consommation de données sont réalisées sur les arcs du graphe connectés à l'acteur. Il est nécessaire de lier ces opérations avec les arcs du graphe construit à l'étape prépédente. Pour rappel, ces productions et consommations sont représentées par des méthodes des ports de données et de paramètres dans le format PaDaF (voir Section 4.2.2). La première étape est de retrouver les appels à ces méthodes dans l'IR LLVM.

Rôle	Prototype C++	*Mangling* IR LLVM
Conso. donnée	`PortIn<int>::pop()`	`_ZN6PortInIiE3popEv`
Conso. données	`PortIn<int>::pop(int)`	`_ZN6PortInIiE3popEi`
Prod. donnée	`PortOut<int>::push(int)`	`_ZN7PortOutIiE4pushEi`
Prod. données	`PortOut<int>::push(int*,int)`	`_ZN7PortOutIiE4pushEPii`
Conso. paramètre	`ParamIn::get()`	`_ZN7ParamIn3getEv`
Prod. paramètre	`ParamOut::set(int)`	`_ZN8ParamOut3setEi`

TABLE 4.1 – Fonctions d'accès au graphe traités par le *front end*.

La construction du graphe flot de données associe chaque acteur et sa fonction de traitement. Pour trouver les accès au graphe, la fonction de traitement est parcourue et chaque appel de méthode est inspecté. Cette inspection se base sur le nom pour associer les appels trouvés et les méthodes d'accès au graphe. Ces noms sont définis de manière unique dans l'IR depuis leur prototype C++ par une opération de *mangling*. Par exemple, la méthode `PortIn<int>::pop()` est connue sous le nom `_ZN6PortInIiE3popEv` dans l'IR, c.-à-d. un nom après *mangling* (`_Z`) avec un nom de classe (`N`) de 6 lettres (`6PortIn`), un argument de *template* (`I`) de type entier (`i`), un nom de méthode (`E`) de 3 lettres (`3pop`) et un argument (`E`) de type *void* (`v`). Chaque méthode d'accès au graphe a donc un nom unique, présenté dans la Table 4.1. Le compilateur associe chaque nom au rôle de cette méthode (ex. accéder aux données, produire un paramètre, etc.) contenu dans une métadonnée liée à l'IR LLVM.

4.3.4 Liaison du graphe à la représentation intermédiaire LLVM

Une fois les appels de méthode trouvés, la seconde étape consiste à connecter ces accès à l'objet du graphe en mémoire avec lequel ils communiquent. Pour cela, il est nécessaire de calculer l'adresse du port accédé, adresse qui est passée en paramètre de l'appel de méthode. Dans le cas général, il est difficile de calculer cette adresse par une analyse statique. J'illustre cette difficulté avec l'exemple de la Figure 4.5. Dans cet exemple l'acteur Decod possède

autant de ports d'entrées que d'antennes, ici 4. L'accès au port de données est réalisé dans une boucle `for(int i=0; i<NB_ANT; i++) {val[i] = Iin[i]->pop(); [...]`.

L'adresse du port accédé qui nous intéresse, `Iin[i]`, est fonction de la valeur du compteur de boucle `i`. Ce port est lié au graphe construit selon le code de la Figure 4.6, plus précisément les arcs définis par `decod.Iin[i] <= ant[i].Iout`. Il faut donc déterminer quel arc est lié au port de données accédé par l'appel de méthode dans l'IR LLVM. Pour cela, je propose de calculer l'adresse du port `Iin[i]` accédé tout comme pour le graphe, par l'exécution du code qui calcule cette adresse.

```
define void @_ZN5Decod7computeEv(%class.Decod* %this) {
  [...]
  %coef = mul i32 4, %var
  %Iin = getelementptr %class.Decod* %this, i32 0, i32 1
  %call1 = call %class.PortIn** @_ZNSt6vectorIP6PortInIiESaIS2_EEixEm(
      %"class.std::vector.18"* %Iin, i64 0)
  %addr = load %class.PortIn** %call1
  %call2 = call i32 @_ZN6PortInIiE3popEv(%class.PortIn* %addr)
```

FIGURE 4.9 – Extrait de l'IR LLVM du traitement de l'acteur Decod.

La fonction de traitement contient un ensemble d'opérations, dont seul une partie est utilisée pour le calcul de l'adresse du port accédé. Je souhaite exécuter uniquement les instructions nécessaires au calcul de cette adresse, et non tout le traitement. J'illustre cette notion d'instructions nécessaires avec l'exemple de l'accès au port d'entrée de l'acteur Decod. Cet accès utilise l'itérateur de la boucle **for** pour calculer l'index du port d'entrée. Cette boucle est déroulée pour fixer statiquement le port accédé sur chaque appel de la méthode. La Figure 4.9 présente un extrait de l'IR LLVM du traitement de l'acteur Decod après avoir déroulé la boucle. L'adresse du port de données `Iin[i]` que l'on souhaite connaître est définie par la variable **addr**, variable passée en argument de l'appel à la méthode `_ZN6PortInIiE3popEv`.

À partir de la variable d'intérêt, Marquet et Moy [Marquet 10] proposent d'utiliser un algorithme de *slicing* pour déterminer les instructions nécessaires à son calcul. Pour rappel, l'IR LLVM est une représentation SSA. Cette représentation facilite l'analyse du graphe de flot de contrôle qui définit les liens de dépendance entre les instructions. Ce lien est représenté en rouge pour la variable **addr** dans l'exemple. Dans l'exemple de la Figure 4.9, l'instruction `%coef = mul i32 4, %var` n'est pas utilisée pour le calcul de l'adresse du port accédé ; Elle est donc ignorée par l'algorithme de *slicing*. Le lecteur est invité à se référer à l'article original pour plus de détails sur l'algorithme utilisé.

Une fois l'algorithme exécuté, les instructions nécessaires sont placées dans une nouvelle fonction. Cette fonction prend comme argument l'acteur auquel appartient la fonction de traitement analysée. L'exécution de cette nouvelle fonction par le JIT de LLVM retourne l'adresse du port concerné. De cette manière, chaque appel de méthode est lié au port accédé. Le résultat est enregistré dans la métadonnée liée à l'appel de méthode pour utilisation future dans le flot de compilation.

La principale limitation de cette approche est qu'elle n'autorise pas le calcul de l'adresse d'un port du graphe dynamiquement. Dans l'exemple précédent, la boucle doit être déroulée pour fixer cette adresse. De manière générale, cette approche ne permet pas l'arithmétique de pointeurs impliquant des données connues uniquement à l'exécution. Elle autorise cependant

que l'accès soit placé dans une structure de contrôle. Par exemple, l'appel de méthode peut se trouver dans une structure conditionnelle, tant que l'adresse de l'élément du graphe est déterminable statiquement.

4.3.5 Optimisations à la construction du graphe

L'infrastructure mise en place dans la section précédente, en plus de lier le graphe à l'IR LLVM, offre des possibilités d'optimisation intéressantes. Elle peut être comparée à la spécification `constexpr` introduite dans le C++11, spécification qui permet de définir des fonctions comme constantes et de les évaluer à la compilation. Je décris dans cette section quelques applications de cette nouvelle manière de propager les constantes.

Lien statique entre méthode et graphe Nous avons vu dans la section précédente comment lier statiquement un appel à une méthode et le port concerné dans le graphe. Une fois ce lien déterminé, l'information est stockée dans la métadonnée. La variable qui passe l'adresse à la méthode est alors supprimée. Le code de calcul de cette variable n'est plus utile, et supprimé par une optimisation d'élimination de code mort.

Paramètre statique dans le graphe La définition d'un acteur générique peut se faire à l'aide de paramètres symboliques. Lors de son implémentation, ces paramètres peuvent être définis statiquement. Par exemple, un acteur FFT a une taille paramétrique, et son implémentation la fixe statiquement. Le compilateur remplace alors l'accès à ce paramètre par sa valeur statique. Le compilateur tire parti de cette valeur statique par une optimisation de propagation de constante.

Paramètre statique dans la fonction de traitement Le calcul d'un paramètre dans la fonction de traitement peut aboutir à une valeur constante. Cette valeur peut être obtenue par simple propagation de constante, ou en utilisant l'infrastructure de la section précédente pour la déterminer. Si ce paramètre est constant, l'appel de méthode produisant ce paramètre est retiré de la fonction de calcul. S'en suit alors une optimisation d'élimination de code mort. Le paramètre constant est remplacé dans tous les acteurs de la région d'influence par sa valeur constante.

Propagation de constante dans le graphe Les deux optimisations précédentes sont combinées pour propager les constantes dans le graphe. Lorsqu'un paramètre est déterminé statiquement, tous les acteurs de la région d'influence sont marqués pour être optimisés. Pour limiter le nombre d'optimisations par acteur, les acteurs sont triés topologiquement, et les optimisations sont d'abord réalisés sur les acteurs sources.

4.4 Travail réalisé

Le développement de ce *front end* a demandé un travail d'analyse important pour comprendre le concept de construction et de liaison du graphe par exécution de code dans le processus de compilation. La définition du format PaDaF, la mise en place de l'infrastructure de compilation et l'adaptation du travail existant au format utilisé a représenté un travail de 4 mois durant ma thèse. Ce travail a abouti au développement d'un *front end* de 6000 lignes de code source C++. 12 applications ont été développées pour évaluer ce *front end*.

Ces applications comprennent tous les exemples présentés dans cette thèse, des applications réelles présentées dans le prochain chapitre, et des applications synthétiques pour évaluer la prise en charge de plusieurs paramètres ou la construction de graphe hiérarchique.

4.5 Discussion

J'ai présenté, à travers ce chapitre, la chaîne de compilation développée durant cette thèse, et plus précisément le *front end*. Pour cela, j'ai commencé par spécifier les contraintes qui m'ont guidé dans ce développement. D'une part, le langage doit exprimer le modèle de calul flot de données paramétrique, et possède une topologie statique. La suite des travaux s'appuyant sur cette hypothèse, les méthodes développées ne s'appliquent pas aux graphes ayant une topologie dynamique. D'autre part, le développement d'applications d'envergure suppose l'utilisation de structures complexes pour construire un graphe hiérarchique et de manière itérative.

Sur la base de ces contraintes, j'ai introduit le format source PaDaF. Ce format utilise le langage C++, enrichi d'une API flot de données dédiée au traitement du signal. Ce format est utilisé à la fois pour la spécification des acteurs et du graphe. L'utililisation du langage C++ est un point essentiel dans l'apport de structures complexes. De plus, ce langage est déjà utilisé pour le développement d'applications embarquées, ce qui facilite la transition des développeurs et du code existant vers ce nouveau format.

J'ai proposé un nouveau *front end* pour transformer ce format PaDaF de haut niveau vers une représentation intermédiaire exploitable par le compilateur. Ce *front end* est basé sur des outils existants, ce qui réduit le temps de développement tout en conservant les capacités du compilateur développé. Plus particulièrement je me suis appuyé sur l'infrastructure de compilation LLVM, projet *open-source* et supporté par l'industrie. Cet outil fournit une base solide pour le développement du compilateur et offre le support du langage C++.

J'ai introduit dans ce chapitre le concept d'exécution durant le processus de compilation. Ce concept innovant est rendu possible par la versatilité de l'infrastructure LLVM. La construction du graphe est la première étape du compilateur à utiliser l'exécution comme support du processus de compilation. Cette méthode rend possible la construction de graphes complexes, et démontre la puissance de ce concept. L'API flot de données est utilisée pour garder le contrôle sur le processus de construction du graphe et donner, le cas échéant, des informations de débogage au développeur.

L'analyse et la liaison de l'IR au graphe flot de données constituent la principale difficulté liée à ce concept. Elles se basent sur les travaux de Marquet et Moy [Marquet 10]. La contribution de ce travail est leur adaptation à l'analyse et la liaison d'applications flot de données. Cette infrastructure a été étendue pour réaliser d'importantes optimisations intra-acteurs, mais aussi inter-acteurs par la propagation de constantes dans le graphe.

Le travail d'optimisation se révèle indispensable dans le cadre de la radio logicielle, avec des plateformes matérielles fortement contraintes. Le chapitre suivant va permettre de mieux comprendre ce besoin, avec la présentation de la plateforme matérielle Magali. Il traitera aussi de la génération de code pour cette cible, utilisée comme démonstrateur dans le cadre de cette thèse.

5 Back End : de la représentation intermédiaire à l'assembleur

It's gonna get worse as we go deeper.

— Joseph Gordon-Levitt as Arthur, *Inception*

Sommaire du chapitre

5.1 Introduction

Dans le chapitre précédent, nous avons vu la conception du *front end* du compilateur développé durant cette thèse. Ce *front end* a permis de transformer le programme depuis une représentation de haut niveau, le format PaDaF codé en C++, vers une représentation intermédiaire propice à l'analyse. Ce chapitre présente la seconde partie du compilateur, le *back end*, qui transforme la représentation intermédiaire en code assembleur pour la plateforme visée. Ce travail de thèse se concentre sur la plateforme Magali du CEA LETI [Clermidy 09a], introduite dans le chapitre d'état de l'art. Je la décrirai plus en détails dans la Section 5.2, afin de comprendre les difficultés spécifiques à cette plateforme.

Il convient d'abord d'introduire la notion de *back end*. Le *back end* est l'outil qui transforme la représentation intermédiaire d'un programme en un code assembleur pour une plateforme donnée. Pour rappel, la représentation intermédiaire en entrée du *back end* devrait être indépendante de la plateforme visée. En réalité, cette représentation contient déjà des informations spécifiques à la plateforme, comme par exemple la taille des entiers. En dehors de ces informations, la représentation intermédiaire est commune à l'ensemble des plateformes d'un compilateur. Chaque *back end* est alors en charge d'optimiser cette représentation selon les spécificités de la plateforme. On peut alors définir le rôle du *back end* comme étant de *spécialiser* la représentation intermédiaire d'un programme en un code assembleur optimisé pour une plateforme donnée.

Cette spécialisation est découpée en deux parties. La Section 5.3 traitera de la vérification et l'optimisation à haut niveau. La Section 5.4 présentera la suite de ces transformations à bas niveau. La spécialisation à haut niveau comprendra toutes les analyses et optimisations sur l'IR LLVM ; celle à bas niveau intègrera ces opérations vers de nouveaux formats pour la vérification ou la génération de code. Afin de mieux comprendre les optimisations nécessaires dans ce *back end*, commençons par étudier plus en détails la plateforme Magali.

5.2 Plateforme Magali

La plateforme Magali est un système sur puce dédié au traitement de la couche physique de protocoles de type MIMO OFDM, tels que le 3GPP-LTE (*3rd Generation Partnership Project*) ou le Wi-Fi. Selon la classification des plateformes de radio logicielles présentée dans la Section 2.2, elle fait partie des approches par blocs configurables. Cette approche vise à minimiser la consommation énergétique tout en offrant des performances permettant de décoder les protocoles actuels en temps réel. Ceci est rendu possible par l'utilisation d'unités configurables, chacune dédiée à un sous-ensemble des traitements de la couche physique (traitement MIMO, mise en symbole OFDM, etc.). Ce type de plateforme est hétérogène. Elle utilise des éléments d'accélération matérielle tant pour le calcul que pour les communications et pour le contrôle de ces opérations. Il en résulte des plateformes performantes mais complexes à programmer. Elle ne correspondent pas une architecture standard de plateforme, et demandent une connaissance approfondie de leur architecture pour tirer parti des mécanismes spécifiques d'accélération matérielle.

Pour mieux comprendre le fonctionnement de cette plateforme et le défi posé au *back end* en charge de sa programmation, je la décris dans cette section. J'introduirai son architecture matérielle dans la Section 5.2.1, avec une vue générale de la plateforme, suivie d'une description des ressources de calcul, des moyens de communication entre ces ressources, et enfin des mécanismes de contrôle. La Section 5.2.2 permettra quant à elle de décrire les travaux existants pour la programmation de cette plateforme.

5.2.1 Architecture matérielle

La Figure 5.1 présente l'architecture de la plateforme Magali. D'un point de vue technologique, elle est construite sur un NoC en deux dimensions, ce qui permet de relier toutes les unités de calcul entre elles. C'est une architecture GALS (*Globally Asynchronous, Locally Synchronous*), chaque unité est localement synchrone alors que le NoC est globalement asynchrone. Elle est construite avec une technologie CMOS 65 nm, et consomme un maximum de 236 mW pour les traitements de la couche physique d'un protocole LTE en MIMO 4x2.

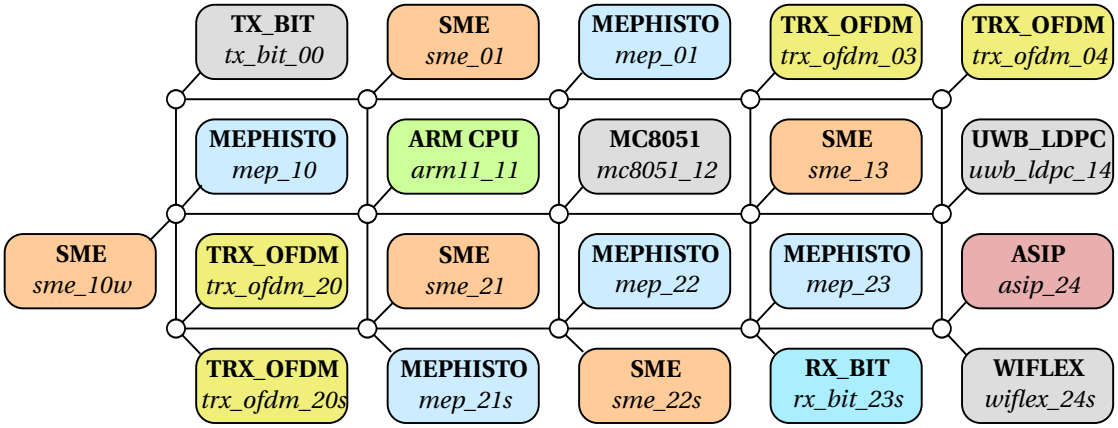


FIGURE 5.1 – Architecture de la plateforme Magali.

Afin de supporter l'architecture GALS, une interface standard fait le lien entre la partie synchrone et asynchrone. De plus, elle contient les mécanismes pour configurer le cœur de calcul, communiquer les données sur le réseau sur puce, et contrôler ces opérations localement. Cette interface, représentée sur la Figure 5.2, est homogène sur toutes les unités de calcul.

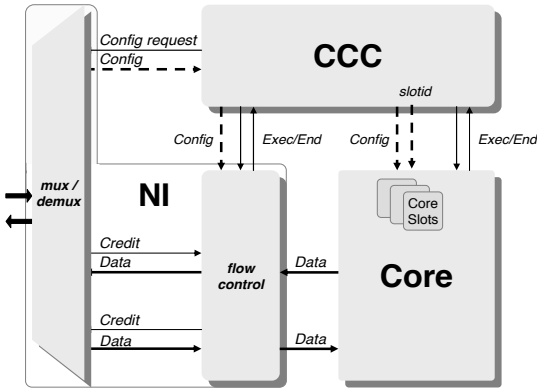


FIGURE 5.2 – Interface entre le cœur de calcul et le NoC sur Magali (de [Clermidy 09a]).

Les ressources de calcul sont hétérogènes, plusieurs unités de calcul dédiées à différents types de traitements sont utilisées sur Magali. Je commence par décrire ces ressources, appelées *Core* sur la Figure 5.2. Je décrirai ensuite l'interface réseau, nommée *NI*, et enfin le

contrôleur local *CCC* (*Configuration and Communication Controller*).

Cœurs de calcul

Les unités de calcul sur la plateforme Magali peuvent être réparties en trois catégories : les unités configurables, les unités programmables dédiées, et les unités programmables généralistes.

Unités configurables Les unités configurables s'apparentent à des IP dédiées à un traitement. Ces unités sont configurables, ce qui permet par exemple de donner la taille d'une FFT à effectuer. Cette catégorie comprend les blocs de traitement :

— *TRX_OFDM* pour le calcul de FFT ou l'inverse et la mise en forme OFDM ;
— *TX_BIT* (resp. *RX_BIT*) pour l'entrelacement (resp. le désentrelacement) ;
— *ASIP* pour l'encodage et le décodage de type Turbocodes ;
— *UWB_LDPC* et *WIFLEX* pour l'encodage et le décodage d'autres types.

Unités programmables dédiées Les unités programmables dédiées sont des unités de calcul spécialisées pour l'exécution de programmes pour un type spécifique de traitement. Cette catégorie conserve la flexibilité d'une approche programmable, tout en apportant l'accélération matérielle pour l'obtention des performances requises pour le décodage d'un protocole MIMO OFDM. On retrouve dans cette catégorie les DSP *Mephisto* [Bernard 11], des processeurs VLIW spécialisés pour le calcul sur des nombres complexes. Il sont en particulier optimisés pour les multiplications-accumulations, nombreuses dans les traitements en télécommunications. Les mémoires distribuées « intelligentes » SME (*Smart Memory Engine*) permettent de stocker les données ainsi que les programmes des unités de calcul [Martin 09]. Ces mémoires sont dites intelligentes car dotées d'une unité de lecture des données programmable. Elles ne sont pas adressées par les autres unités de calcul, mais constituent des unités de calcul à part entière. Ces mémoires ont deux rôles : celui de DMA pour l'accès non linéaire aux données (ex. désentrelacement à grosse granularité, envoi de données sur plusieurs unités de traitement, etc.), et le rôle de serveur de configurations qui distribue les programmes aux autres unités.

Unités programmables généralistes Les unités programmables généralistes comprennent le processeur ARM et le microcontrôleur 8051. Le processeur ARM est utilisé pour réaliser les calculs non supportés par les autres unités. Cette solution permet de s'assurer que la plateforme est évolutive, les traitements non supportés par les catégories précédentes pouvant être exécutés a minima par ce processeur. Le processeur ARM est aussi utilisé comme contrôleur centralisé pour synchroniser les unités et définir les traitements à effectuer. Le microcontrôleur 8051 est dédié à la gestion de l'énergie.

Communications

Les communications entre les unités de la plateforme Magali sont prises en charge par un composant dédié, appelé NI sur la Figure 5.2. Ces communications ont lieu au travers de FIFO, et utilisent un mécanisme de synchronisation basé sur un échange crédits/données. Dans ce mécanisme, le récepteur envoie des crédits à l'émetteur en fonction du nombre de places disponibles. L'émetteur réalise l'envoi dès qu'il a suffisamment de crédits et de données disponibles.

Ce mécanisme implique la configuration de l'émetteur et du récepteur de manière cohérente pour réaliser la synchronisation. Pour chaque communication, l'émetteur (resp. le récepteur) doit connaître le récepteur (resp. l'émetteur), le port sur lequel envoyer (resp. depuis lequel recevoir) et le nombre de données échangées. Toutes ces configurations doivent être connues statiquement, et sont exécutées par le contrôleur local.

Contrôle

La dernière partie de l'interface commune aux différentes unités de calcul est le Contrôleur de Communications et de Configuration (CCC). Le CCC exécute des *microprogrammes* pour chacune des FIFO d'entrée et de sortie, ainsi que pour chaque cœur de calcul. Il possède un jeu d'instructions limité, illustré sur la Table 5.1. Ce jeu d'instructions permet de charger une suite de configurations, et d'itérer sur ces configurations selon une valeur fixe ou stockée dans un registre.

Instruction		Description
`RC`	`c`	Charger la configuration `c`.
`RCL`	`c`	Charger la configuration `c` + pointeur de début de boucle.
`LL`	`n`	Répéter `n` fois à partir du pointeur de début de boucle.
`GL`	`n`	Répéter `n` fois à partir de la première instruction.
`LLi`	`r`	Répéter le nombre de fois stocké dans le registre `r` à partir du pointeur de début de boucle.
`GLi`	`r`	Répéter le nombre de fois stocké dans le registre `r` à partir de la première instruction.
`STOP`		Fin du microprogramme.

TABLE 5.1 – Jeu d'instructions du CCC de Magali.

L'exemple de la Figure 5.3 illustre l'exécution de microprogrammes. Dans cet exemple, le microprogramme du cœur de calcul de C exécute une première configuration 2 fois, suivie d'une seconde configuration. De manière similaire, le microprogramme de la FIFO *ICC0* reçoit 10 données du cœur A suivies de 20 données du cœur B, et répète cette séquence 3 fois.

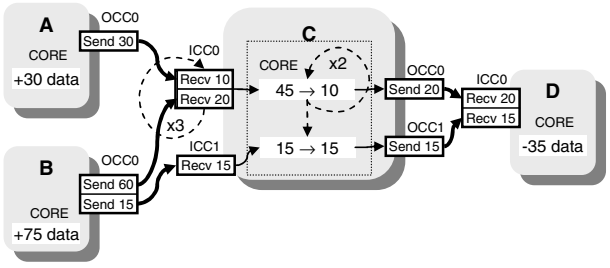


FIGURE 5.3 – Exemple de communications et calculs sur Magali (de [Clermidy 09b]).

Pour une application simple comme celle illustrée sur la Figure 5.3, il est nécessaire de programmer 10 configurations d'émetteurs et de récepteurs, 5 configurations de cœurs, ainsi que 11 microprogrammes. Programmer cet ensemble de configurations et microprogrammes devient rapidement complexe et source d'erreurs. En effet, le contrôle distribué nécessite la

programmation de configurations indépendantes avec des valeurs globalement cohérentes qui représentent l'application. Plusieurs travaux ont été proposés pour générer ces configurations automatiquement. Je détaille ces travaux sur la programmation de Magali dans la prochaine section.

5.2.2 Modèles de programmation existants

Plusieurs travaux antérieurs à cette thèse ont porté sur la programmation de la plateforme Magali. Mrabti *et al.* [Mrabti 09] proposent l'utilisation de modèles abstraits pour l'application et l'architecture, basés sur le langage XML. L'application est codée selon un modèle de calcul flot de données autorisant les flux à plusieurs entrées/sorties et spécifie les échanges de données sous une forme cyclique. La granularité des acteurs utilisés est égale à celle des unités de calcul sur la plateforme Magali. Le modèle d'architecture ainsi qu'un placement à deux phases permet d'abstraire la plateforme matérielle.

Llopard, Martin et Rousseau [Llopard 11b] ont implémenté cette approche dans le compilateur *comC*. Ce compilateur transforme le graphe de l'application pour tirer parti des mécanismes de communication de la plateforme. Il prend en charge la génération du code de l'ensemble des ressources matérielles (cœurs de calcul, communications, contrôle distribué et centralisé), et génère une implémentation optimisée.

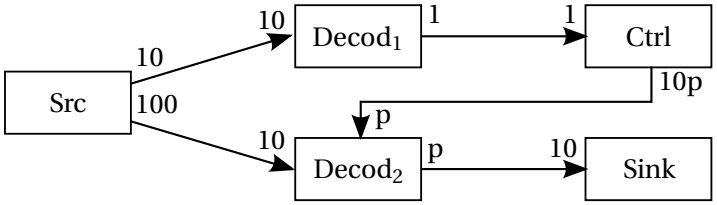


FIGURE 5.4 – Exemple de protocole dynamique sous la forme d'une application paramétrique.

Cette approche, développée autour de la plateforme Magali, est idéale pour cette dernière. Elle est cependant limitée à cette plateforme, les opérateurs utilisés pour les calculs correspondant exactement aux cœurs de calcul. Par ailleurs, l'utilisation d'un modèle de calcul statique ne permet pas de représenter des protocoles de communications dynamiques tels que le LTE. Un exemple simplifié de la dynamicité d'un tel protocole est présenté sur la Figure 5.4. Les données à décoder sont découpées en deux parties. La première partie est utilisée pour calculer le paramètre p, qui définit le décodage à utiliser dans la seconde partie.

Si le modèle de programmation ne supporte pas les paramètres, alors l'application est découpée en *phases*. Chaque phase est un graphe flot de données statique. L'enchaînement entre chacune de ces phases est défini par le développeur. La Figure 5.5 présente l'application précédente découpée en phases. La transition entre les différentes phases est représentée sous la forme d'une machine d'état englobant les graphes de chaque phase. Nous verrons dans la Section 6.3.3 que l'utilisation de phases peut générer un surcoût temporel dans le cas d'une application paramétrique.

Ben Abdallah *et al.* [Ben Abdallah 10] proposent d'utiliser une machine virtuelle radio (RVM) comme abstraction de la plateforme Magali. Ils utilisent le modèle de calcul KPN pour représenter l'application. Ce modèle de calcul possède une production et une consommation de données dynamiques, qui ne sont pas supportées par les contrôleurs distribués. En

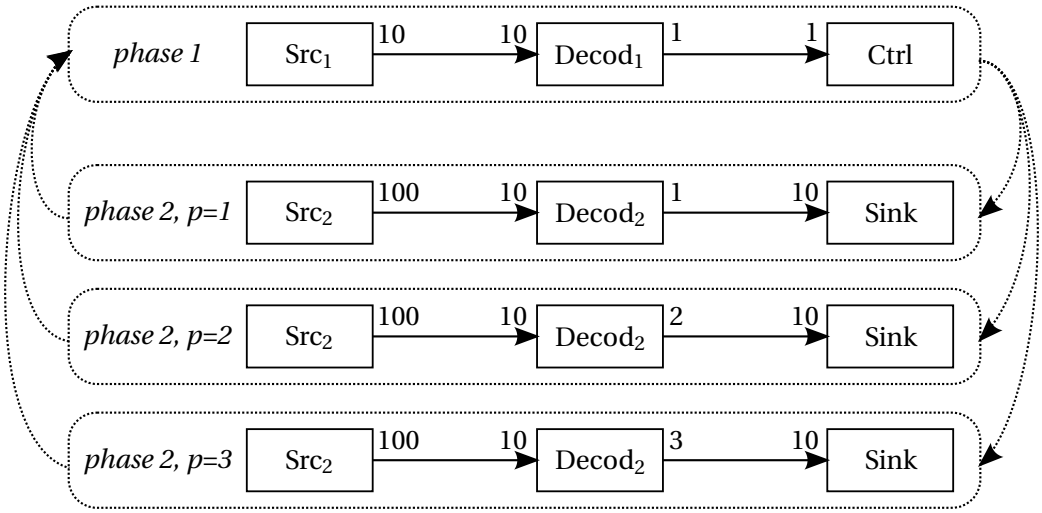


FIGURE 5.5 – Exemple de protocole dynamique découpé en phases sous la forme d'une application statique.

conséquence, les capacités de séquentialisation de ces derniers ne sont pas exploitées par cette approche, ce qui pénalise largement ses performances.

Au final, plusieurs approches ont déjà été développées pour programmer la plateforme Magali. Ces approches permettent d'abstraire la complexité de la plateforme, mais ne permettent pas d'abstraire ses opérateurs spécifiques. L'utilisation d'une API de traitement du signal standard dans ma thèse vise à améliorer cette abstraction. La Section 5.3 présentera comment spécialiser l'application pour la plateforme visée. D'autre part, les modèles de calcul utilisés viennent pénaliser les performances de ces approches. Dans mon travail, je propose d'utiliser un modèle de calcul paramétrique pour représenter le dynamisme de l'application. Je décrirai dans la Section 5.4 la génération de code optimisé pour Magali depuis une application paramétrique.

5.3 Vérification et optimisation de haut niveau

Cette section traite de la vérification et l'optimisation de la représentation intermédiaire. Je présenterai dans la Section 5.3.1 la vérification du graphe flot de données, dernière étape indépendante de la plateforme visée. Cette vérification s'appuie sur le formalisme flot de données pour analyser le graphe. Cette analyse, au delà de la vérification du graphe, est exploitée pour l'optimisation de l'application. Je décrirai ensuite l'intégration d'un outil de visualisation du graphe dans Section 5.3.2.

Je présenterai dans les Sections 5.3.3 et 5.3.4 le placement et l'ordonnancement de l'application sur la plateforme cible. Ces étapes spécialisent l'application en la mettant en relation avec la plateforme cible. Elles sont essentielles, un mauvais placement ou ordonnancement pouvant *sérialiser* l'exécution d'un graphe naturellement parallèle.

J'introduirai dans la Section 5.3.5 la fusion d'acteurs dans le graphe flot de données. Cette optimisation, largement utilisée dans la compilation d'applications flot de données, est utilisée dans ce travail pour adapter l'application aux accélérateurs matériels.

5.3.1 Vérification du graphe flot de données

Dans la Section 3.2 du Chapitre 3, j'ai introduit le modèle de calcul flot de données statique et ses différentes évolutions dynamiques. Ces évolutions jouent sur l'équilibre entre expressivité et analysabilité du modèle de calcul. J'ai choisi en particulier le modèle flot de données paramétrique pour conserver une grande analysabilité. Ces analyses permettent de vérifier statiquement la cohérence des échanges de données dans le graphe, ainsi que celle des paramètres dans le cas d'une application paramétrique. Je présente l'implémentation de ces analyses dans cette section.

La première étape est de vérifier la topologie du graphe. L'utilisation du *front end* décrit dans le chapitre précédent permet, lors de la construction du graphe, de bloquer certaines constructions interdites. Le format PaDaF, basé sur le langage C++, possède des primitives fortement typées. Les vérifications syntaxique (ex. nombre d'opérandes) et sémantique (ex. utilisation d'une variable après sa déclaration) sont réalisées par le front end Clang. Cette vérification évite, par exemple, la connexion d'un port de sortie d'un acteur vers un autre port de sortie. Par ailleurs, les classes de base du format PaDaF contiennent du code de contrôle exécuté lors de la construction du graphe par le *front end*. De cette manière, un port de données ne peut être relié qu'une seule fois à un autre port de données. Une fois le format intermédiaire construit par le *front end*, une vérification topologique explore l'ensemble du graphe pour vérifier que tous les ports de tous les acteurs sont connectés et que tous les acteurs ont une fonction de traitement `compute()`.

Une propriété importante d'un graphe flot de données statique est la possibilité de vérifier la cohérence du graphe statiquement. La cohérence, telle que définie dans la Section 3.2.1, garantit la vivacité du graphe et la limite finie de la taille des FIFO. Cette propriété est étendue aux graphes paramétriques dans le modèle de calcul SPDF. Pour la vérifier, l'itération globale des acteurs est calculée à l'aide d'une représentation symbolique des paramètres. Le vecteur d'itération obtenu est ensuite utilisé lors de l'ordonnancement des acteurs.

En plus de la cohérence du graphe, il est nécessaire de vérifier la période de changement du paramètre. Pour cela, l'influence, les régions et les itérations locales sont déterminées en s'appuyant sur la théorie développée autour du modèle SPDF et résumée dans la Section 3.2.3. Dans le cadre de ce travail, j'ai limité le paramètre à un changement de valeur durant l'itération. Afin d'appliquer cette contrainte, il faut vérifier que le rythme de modification d'un paramètre est égal à l'itération globale de l'acteur produisant ce paramètre. L'analyse de l'influence d'un paramètre et de sa région d'influence sont alors utilisés pour connaître les acteurs qui utilisent ce paramètre. Cette information importante est conservée pour la synchronisation des paramètres lors de l'ordonnancement, puis la génération du code de contrôle.

5.3.2 Visualisation du graphe

La programmation de l'ensemble de l'application se fait dans le format PaDaF, entièrement en C++. Il est difficile de visualiser et vérifier le graphe seulement à partir du code. Pour contrevenir à cette difficulté, j'ai intégré dans le compilateur un générateur graphique se basant sur l'IR PaDaF. La Figure 5.6 présente la visualisation du récepteur MIMO simplifié généré par le compilateur.

Ce générateur se base sur le langage dot et l'outil Graphviz [Graphviz 14] pour visualiser le graphe. Ce format est déjà utilisé dans le compilateur LLVM pour visualiser le graphe de contrôle de flot de l'IR LLVM. C'est un outil de débogage important pour le développement d'applications PaDaF. Il permet de visualiser le graphe dès la génération de l'IR PaDaF, mais

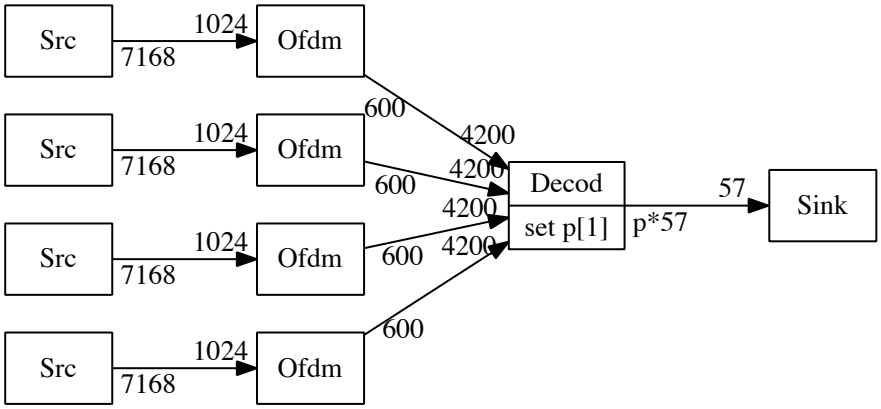


FIGURE 5.6 – Visualisation du récepteur MIMO simplifié généré par le compilateur.

aussi tout au long des transformations et optimisations réalisées dans le *back end*.

5.3.3 Placement

Afin d'optimiser l'application pour une plateforme donnée, elle est dans un premier temps placée sur la plateforme. Le placement d'acteurs sur des unités de calcul basé sur un langage de description d'architecture est un problème largement étudié [Cardoso 10, Castrillon 11, Singh 13]. Ce problème est NP-complet, il n'existe pas de solution optimisée et satisfaisante. J'ai choisi de ne pas l'étudier, et de baser mes travaux sur un placement manuel. Le placement est réalisé à la granularité de l'acteur ; un acteur dans le domaine de la radio logicielle est relativement large (un acteur pour une FFT). Le coût du placement manuel reste donc raisonnable. Cette hypothèse est commune à de nombreux travaux du domaine [Sutton 10, Gonzalez-Pina 12].

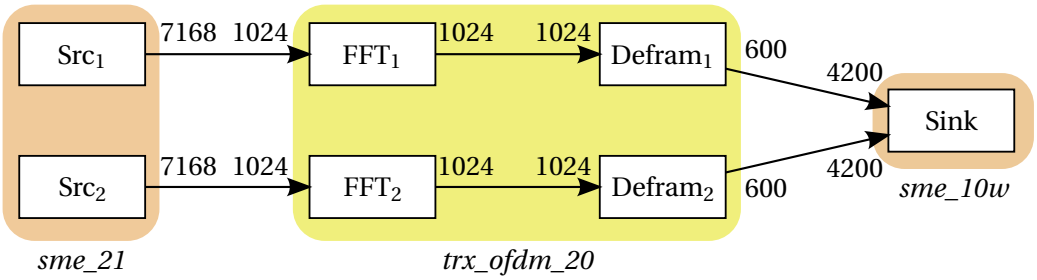


FIGURE 5.7 – Exemple de réception MIMO placé sur la plateforme Magali.

La Figure 5.7 illustre un exemple de graphe placé sur la plateforme Magali. Elle représente une partie d'une réception MIMO à deux antennes. Cette application est placée sur 3 cœurs de la plateforme Magali :

— *sme_21* est la source du signal ;
— *trx_ofdm_20* effectue la FFT et le décadrage (*deframing*), c.-à-d. la suppression des bandes de garde autour du signal d'intérêt ;
— *sme_10w* stocke le signal traité.

Un modèle de la plateforme est fourni par le langage, et chaque acteur précise son placement lors de son instanciation. Le format d'entrée PaDaF ainsi que l'exécution de la construction du graphe laisse entrevoir de nombreuses possibilités. Le code de construction du graphe peut accéder à des fichiers de configuration sur la machine hôte ainsi qu'exécuter un code C++ arbitrairement complexe pour déterminer le placement durant la phase de construction du graphe.

5.3.4 Ordonnancement

L'ordonnancement d'un graphe flot de données a été largement abordé au Chapitre 3. La théorie est développée pour les graphes flot de données statiques, ainsi que les graphes flot de données paramétriques. Dans ce cas on parle d'un ordonnancement quasi-statique, fonction de la valeur des paramètres. Ce type d'ordonnancement est formalisé par la Définition 3. Cette section s'appuie sur ces travaux pour établir l'ordonnancement de l'application.

L'implémentation retenue est un ordonnancement simple, basé sur un tri topologique du graphe. Cette méthode génère un ordonnancement SAS, qui est ensuite factorisé pour réduire la consommation mémoire. L'ordonnancement est étendu au multicœur en fonction du placement défini à l'étape de compilation précédente. On obtient ainsi un ordonnancement quasi-statique par unité de calcul. La synchronisation entre les unités de calcul est prise en charge par le modèle d'exécution. Dans le cas de la plateforme Magali, la synchronisation des données est prise en charge par le matériel. La synchronisation des paramètres est réalisée par le contrôleur centralisé. Le modèle d'exécution est décrit plus en détails lors de la génération du code de contrôle dans la Section 5.4.4.

Si l'on reprend l'exemple de la Figure 5.7, l'ordonnancement d'une branche de réception selon un tri topologique est évident. Le modèle de calcul flot de données ne permet cependant pas de discriminer la branche à exécuter en premier. La Section 5.4.1 sur le contrôle des tailles mémoires montre l'intérêt du micro-ordonnancement pour guider ce choix. Ce contrôle est effectué après la fusion d'acteurs, optimisation qui transforme le graphe pour le spécialiser pour la plateforme visée.

5.3.5 Fusion d'acteurs

Nous avons vu dans la Section 4.2.2 que le format PaDaF comprend une API décrivant les traitements du signal sous la forme d'un ensemble d'opérateurs indépendants de la plateforme. La difficulté qui se pose alors est d'adapter cette API aux accélérateurs de la plateforme matérielle. En particulier, les opérations réalisables par un seul et même accélérateur peuvent se retrouver réparties sur plusieurs acteurs, ce qui empêche de tirer parti de toutes les capacités de l'accélérateur matériel. Pour résoudre ce problème, je propose d'utiliser la *fusion* d'acteurs pour spécialiser l'application à la plateforme matérielle visée.

La fusion de filtres a été proposée par Proebsting et Watterson [Proebsting 96], puis adaptée au flot de données dans StreamIt [Thies 09]. La fusion est une transformation qui rassemble deux acteurs en joignant leurs fonctions de calcul, permettant d'optimiser le code dans la fonction obtenue. Dans les travaux précédents, les FIFO connectant les acteurs sont remplacées par des variables locales. Cette transformation autorise de nouvelles optimisations, comme la minimisation de l'utilisation de la mémoire, la fusion ou le déroulage de boucle. Dans ce travail, je propose de fusionner les appels à l'API de traitement du signal pour rendre possible le support des accélérateurs matériels [Dardaillon 14c].

L'analyse du code des acteurs permet de sélectionner les acteurs candidats pour une fusion.

Pour ce faire, le code de chaque acteur est analysé pour détecter les appels à l'API. Lorsque plusieurs acteurs sont placés sur la même unité de calcul, le schéma d'appel aux traitements est comparé à celui de l'accélérateur matériel. Si les accès sont compatibles, alors les acteurs considérés sont candidats à la fusion. Avant de réaliser la fusion des acteurs, il est nécessaire de vérifier que cette fusion ne crée pas d'interblocage dans le graphe. Pour cela, j'utilise une condition similaire à celle utilisée par Bhattacharyya dans les travaux sur l'ordonnancement d'acteurs par regroupement [Bhattacharyya 99]. Pour que des acteurs soient fusionnés, il est nécessaire que l'acteur résultant ne crée pas de cycle dirigé.

Si la condition est vérifiée, alors les acteurs sont fusionnés. Il est alors nécessaire d'ordonnancer le sous-graphe d'acteurs à fusionner. Pour cela, j'utilise la notion *d'itération locale* telle que définie dans SPDF et rappelée dans la Section 3.2.3. Elle permet de définir le nombre d'exécutions de chaque acteur pour que le sous-graphe revienne à son état initial. On utilise alors un tri topologique pour ordonnancer le sous-graphe selon l'itération locale. Dans cet ordonnancement, chaque acteur est remplacé par sa fonction de calcul. Les FIFO connectant les acteurs du sous-graphe sont remplacées par des variables locales. Les FIFO connectant le sous-graphe aux autres acteurs sont conservées sur l'acteur résultant, et le rythme de production ou de consommation de ces FIFO est multiplié par l'itération locale. Une passe d'optimisation est exécutée sur le code de calcul de l'acteur résultant. En plus des effets bénéfiques cités plus haut sur la consommation mémoire ou la fusion de boucle, cette passe réalise la fusion des appels à l'API pour spécialiser le traitement à la plateforme visée.

Dans l'exemple de la Figure 5.7, les acteurs FFT et Defram sont placés sur l'unité de calcul *trx_ofdm_20*. Or, cette unité est capable de réaliser une FFT suivie d'un décadrage. Les deux acteurs sont donc candidats à la fusion. Leur fusion ne créant pas de cycle dirigé, leurs fonctions de traitement sont assemblées selon l'ordonnancement (FFT; Defram). Cet exemple simple permet de démontrer l'intérêt de la fusion pour la spécialisation d'une application vers une plateforme utilisant des accélérateurs matériels.

5.4 Vérification et optimisation de bas niveau

Les optimisations présentées dans la section précédente ont spécialisé l'application flot de données pour la plateforme Magali. En particulier, l'application est placée, ordonnancée, et sa granularité adaptée à la plateforme Magali. Ces optimisations ont transformé la représentation intermédiaire en une forme adaptée à la génération de code pour la plateforme Magali, présentée dans cette section.

La Figure 5.8 détaille la vérification et l'optimisation de code pour la plateforme Magali. Je présenterai en premier lieu le contrôle des tailles mémoires dans la Section 5.4.1. Cette vérification passe par la génération de code PROMELA pour modéliser le graphe placé sur la plateforme Magali, puis le vérifier à l'aide du *model checker* SPIN. Je présenterai ensuite la génération de code pour la cible de ce travail, Magali. La plateforme Magali est hétérogène, elle possède un ensemble d'unités de calcul qui sont configurables ou programmables. La génération de code, ajustée aux spécificités de ces différentes unités, sera présentée dans la Section 5.4.2. Les communications et le contrôle distribué des unités de calcul possèdent une interface similaire. La génération du code pour ces interfaces sera présentée dans la Section 5.4.3 pour les communications, suivie de la Section 5.4.4 pour le contrôle.

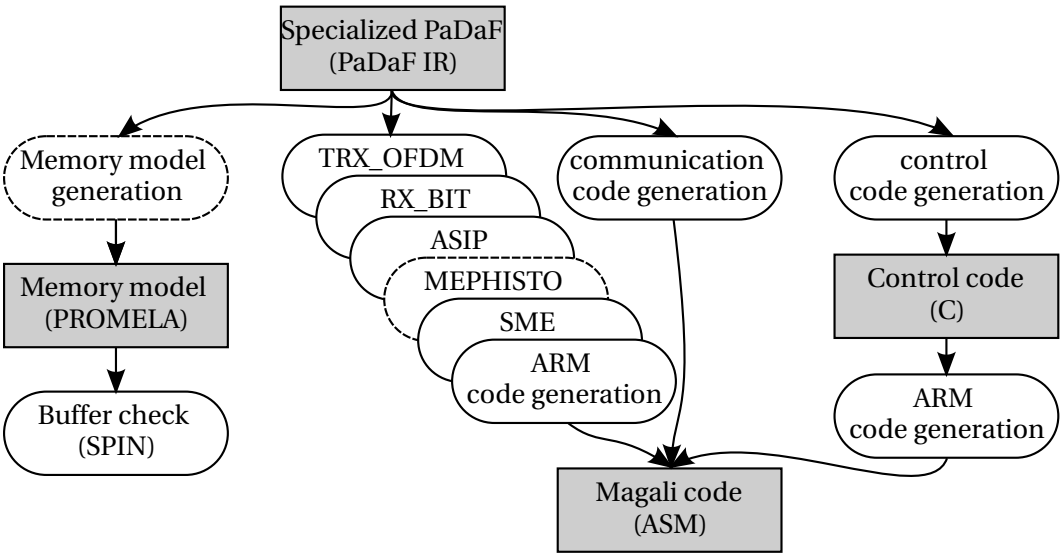


FIGURE 5.8 – Détails de la vérification et optimisation de code pour Magali.

5.4.1 Contrôle des tailles mémoires

Le contrôle des tailles mémoires permet de vérifier que l'application optimisée pour une plateforme est capable de s'exécuter sur cette plateforme en tenant compte de la mémoire disponible. Ce contrôle se base sur le micro-ordonnancement défini dans la Section 3.4. Il utilise le placement et l'ordonnancement de l'application, ainsi que la taille des mémoires disponibles sur la plateforme.

Dans la Section 3.4.3, le contrôle des tailles mémoires est effectué à la granularité d'un micro-ordonnancement par unité de calcul. Ce micro-ordonnancement comprend les productions et consommations de données et de paramètres des acteurs placés sur l'unité de calcul considérée. Il respecte l'antériorité du graphe et du micro-ordonnancement de chacun des acteurs, mais autorise l'entrelacement des actions de plusieurs acteurs. Dans le cas de la plateforme Magali, la plateforme n'est pas capable d'exécuter plusieurs acteurs de manière concurrente sur une unité de calcul autre que le contrôleur centralisé. Pour cette raison, la granularité du micro-ordonnancement est limitée à l'acteur. Cette exécution n'est cependant pas atomique, car les communications sont décomposées pour évaluer leur consommation mémoire réelle.

$$\begin{gathered}
\mu S_{\mathrm{Src}_1} = \left(\mathrm{push}(1024)^7\right) \qquad \mu S_{\mathrm{Src}_2} = \left(\mathrm{push}(1024)^7\right) \\
\mu S_{\mathrm{Ofdm}_1} = \left(\mathrm{pop}(1024); \mathrm{push}(600)\right) \qquad \mu S_{\mathrm{Ofdm}_2} = \left(\mathrm{pop}(1024); \mathrm{push}(600)\right) \\
\mu S_{Sink} = \left(\mathrm{pop}_{(\mathrm{Ofdm}_1,\mathrm{Sink})}(600)^7; \mathrm{pop}_{(\mathrm{Ofdm}_2,\mathrm{Sink})}(600)^7\right)
\end{gathered} \tag{5.1}$$

Pour l'exemple de la Figure 5.7, on obtient les micro-ordonnancements (5.1) pour les acteurs. La mention des arcs de production ou de consommation est omise lorsqu'ils sont évidents. En fonction de l'ordonnancement des branches effectué dans la Section 5.3.4, on obtient le micro-ordonnancement (5.2) ou (5.3). Le micro-ordonnancement de l'acteur Sink guide notre choix vers le micro-ordonnancement (5.2). Cette information permet non seulement d'optimiser l'ordonnancement, mais aussi d'éviter un blocage de l'application dans le

cas où la mémoire disponible pour la FIFO de l'arc (Ofdm_2, Sink) est inférieure à 4200 données.

$$\begin{array}{c} \mu S_{sme\_21} = \left(\mu S_{\text{Src}_1}; \mu S_{\text{Src}_2}\right) \\ \mu S_{trx\_ofdm\_20} = \left(\mu S^7_{\text{Ofdm}_1}; \mu S^7_{\text{Ofdm}_2}\right) \\ \mu S_{sme\_10w} = \left(\mu S_{\text{Sink}}\right) \end{array} \quad (5.2) \qquad \begin{array}{c} \mu S_{sme\_21} = \left(\mu S_{\text{Src}_2}; \mu S_{\text{Src}_1}\right) \\ \mu S_{trx\_ofdm\_20} = \left(\mu S^7_{\text{Ofdm}_2}; \mu S^7_{\text{Ofdm}_1}\right) \\ \mu S_{sme\_10w} = \left(\mu S_{\text{Sink}}\right) \end{array} \quad (5.3)$$

La génération du modèle PROMELA, telle que défini dans la Section 3.4.3, n'est pas encore implémentée dans le compilateur. Après étude des différents types d'unités de calcul présentes sur la plateforme, je présente brièvement la génération de modèle envisagée. Le micro-ordonnancement des unités de calcul serait généré selon l'ordonnancement de chaque unité de calcul, de manière directe. Le micro-ordonnancement d'un acteur placé sur un accélérateur matériel s'appuierait sur la spécification des communications de cet accélérateur, intégrée au modèle de la plateforme. Le micro-ordonnancement d'un acteur placé sur une unité programmable serait quant à lui réalisé à partir d'une analyse du code de l'acteur pour en extraire l'ordre des communications. Cette dernière génération semble la plus complexe, et demanderait une évaluation plus complète. Le modèle de la plateforme permettrait de préciser la répartition des tailles de mémoire pour la représentation des FIFO.

Le modèle PROMELA des applications est actuellement réalisé manuellement, comme dans l'exemple de la Figure 3.10. Il permet dès à présent d'évaluer le bon fonctionnement des applications, ou de les corriger dans le cas où un ordonnancement créerait un interblocage. Cette correction se fait par l'ajout manuel de contraintes sur le graphe applicatif, sous la forme d'arcs supplémentaires entre les acteurs dont on souhaite contraindre l'ordre d'exécution. Ces arcs supplémentaires sont définis comme des contraintes d'ordonnancement et n'impliquent pas de communication de donnée ou de paramètre. Cette méthode permet une vérification autrement absente lors du développement sur la plateforme Magali ; Elle réduit le temps d'évaluation d'une application par rapport à une exécution sur la plateforme d'évaluation.

5.4.2 Génération des cœurs de calcul

La fusion, présentée dans la section précédente, rend possible le support des accélérateurs matériels présents sur la plateforme Magali. La représentation intermédiaire est spécialisée pour correspondre aux unités de calcul disponibles. En fonction de l'unité de calcul visée, le code des cœurs de calcul est généré de manière différente.

Dans le cas des unités configurables, un générateur de code est associé à chacun des types de cœur. Le compilateur intègre les générateurs de code développés durant cette thèse pour les unités de calcul TRX_OFDM, RX_BIT et ASIP. Ces générateurs supportent les opérations :

- de production et consommation de données ;
- de FFT et FFT inverse ;
- de filtrage de données par masque (ex. décadrage) ;
- de démodulation (décodage du signal reçu complexe en valeur numérique) ;
- de désentrelacement à la granularité du mot et du bit ;
- de *puncturing* et *depuncturing* (décimation en sortie d'un code convolutionnel et opération inverse par ajout de zéros) ;
- de décodage Turbocode.

Toutes les opérations sont paramétrables. Par exemple, on peut changer la taille d'une FFT ou le type de démodulation. Les paramètres sont cependant statiques dans la configuration des unités de calcul. Dans la majorité des cas, ces paramètres sont connus statiquement. La configuration est alors générée à partir des valeurs des paramètres, passées en argument de la fonction. Il existe cependant des cas où cette valeur de paramètre n'est pas connue statiquement. Les protocoles radio dynamiques, par exemple, peuvent recevoir un nombre variable de données ou utiliser une modulation différente selon la qualité du canal radio.

L'utilisation des paramètres est un point central de ce travail, et le *back end* Magali prend en charge le fait qu'un paramètre soit variable. Pour supporter les paramètres variables, un jeu de configurations est généré pour chacune des valeurs possibles de ces paramètres. Un code d'évaluation de la valeur des paramètres est ajouté dans le code de contrôle. En fonction de cette valeur, ce code sélectionne la configuration à charger sur l'unité de calcul. À noter que cette approche est différente de celles des communications paramétriques, qui sera présentée dans la section suivante.

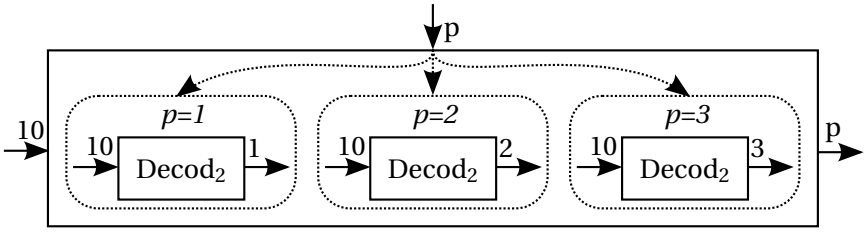


FIGURE 5.9 – Exemple de configuration paramétrique sur la plateforme Magali.

L'exemple de la Figure 5.9 illustre pour l'acteur Decod_2 de la Figure 5.4 la gestion de plusieurs configurations en fonction de la valeur du paramètre. La génération du code de contrôle permettant de sélectionner les différents modes en fonction de la valeur du paramètre est entièrement automatique. Une telle approche sans le support du compilateur impliquerait la programmation manuelle d'un ensemble de configurations pour les unités de calcul, les communications, le code de contrôle, ainsi que la vérification de l'ensemble de ce code. Au vu de la complexité, la programmation manuelle de ce type de plateforme se fait par phases, comme vu dans la Section 5.2.2. Cette étape de la génération du code rend donc possible le support de paramètres sur la plateforme Magali dans une approche globale.

```
Decod2(1) {
  cfg_mode    0x0
  nb_complex  10
  llr_rate    0
  alternate   0
  modulation  1
  q_pos       1
  end_seq     1
}
```

```
Decod2(2) {
  cfg_mode    0x0
  nb_complex  10
  llr_rate    0
  alternate   0
  modulation  2
  q_pos       1
  end_seq     1
}
```

```
Decod2(3) {
  cfg_mode    0x0
  nb_complex  10
  llr_rate    0
  alternate   0
  modulation  3
  q_pos       1
  end_seq     1
}
```

FIGURE 5.10 – Pseudo-code du cœur de calcul de l'unité *rx_bit_23s*.

Dans le cas des unités programmables, la génération de code est un processus complexe

qui nécessite l'utilisation d'un compilateur complet. Il serait illusoire de vouloir réécrire un *back end* pour un DSP ou un processeur généraliste dans la durée d'une thèse. Pour cette raison je réutilise, lorsque disponible, les outils existants.

DSP Mephisto Le DSP Mephisto est un processeur VLIW spécialisé pour le traitement du signal, avec des spécificités comme des fichiers de registres adressables. La compilation efficace pour ce type d'architecture de processeur est un problème connu [Hines 05]. Llopard *et al.* proposent un compilateur basé sur LLVM pour le processeur Mephisto [Llopard 13]. Le *back end* de ce compilateur pourrait être intégré dans mon flot de compilation basé sur la même IR LLVM. Cependant, l'état de développement du compilateur Mephisto lors de mes expérimentations n'a pas permis son intégration dans mon flot de compilation.

Mémoire distribuée SME La mémoire distribuée SME possède aussi son propre compilateur. Ce compilateur utilise un sous ensemble du langage C pour programmer la gestion mémoire. L'utilisation d'un format spécifique, un sous ensemble du langage C, ne permet pas d'utiliser le format PaDaF comme format d'entrée. Son intégration dans ce travail se fait au travers des appels à l'API. Chaque appel à l'API supporté par le SME est associé à un modèle de code dans le format spécifique au SME. Ce modèle de code est complété avec les paramètres de la fonction appelée, par exemple la taille des données à désentrelacer. Le code généré est ensuite fourni au compilateur du SME pour générer le code natif. Cette méthode, bien que primitive, permet de supporter de nouvelles unités de calcul.

Processeur ARM Enfin, le processeur généraliste ARM est utilisé pour les traitements non supportés par les autres cœurs de calcul. Le fait de pouvoir exécuter n'importe quel traitement, y compris ceux non envisagés lors de la conception des accélérateurs, est primordial pour une radio logicielle. Sans cette possibilité, une radio logicielle est incapable de s'adapter aux évolutions du protocole. Cette propriété importante est supportée par le flot de compilation. Les acteurs non supportés par les accélérateurs sont placés sur le processeur ARM. Ces acteurs sont ordonnancés statiquement, et le code résultant est placé sur un fil d'exécution. Les communications avec les autres unités sont assurées par l'interface commune. Le *back end* ARM de LLVM est utilisé pour compiler l'IR PaDaF. De plus, le processeur ARM joue aussi le rôle de contrôleur central. L'implémentation de ce contrôle est décrite dans la Section 5.4.4.

5.4.3 Génération des communications

Toutes les communications dans un graphe flot de données sont explicites par construction. Le graphe de l'application définit la source, la destination, ainsi que le nombre de données échangées, le tout dans une représentation de haut niveau. Cette représentation de haut niveau permet de s'abstraire de la plateforme, mais aussi de générer automatiquement le code pour la plateforme cible. Dans le cas de la plateforme Magali, l'interface de communication est commune à toutes les unités de calcul. Cette section s'occupe de l'optimisation et la génération de configurations pour cette interface.

Comme nous l'avons vu dans la Section 5.2.1, cette interface réseau commune est programmable. Ce programme permet d'exécuter une suite de configurations et d'itérer sur ces configurations. Dans le cas de communications, ce programme rend possible l'envoi de données à plusieurs récepteurs selon des schémas de communications complexes. Il est utilisé à deux fins lors de la génération de code pour la plateforme Magali.

D'une part, il supporte l'envoi (resp. la réception) de données vers (resp. depuis) plusieurs unités de calcul par un acteur. Sur l'exemple de la Figure 5.7, l'acteur Sink reçoit les données de l'acteur Ofdm_1 suivi de Ofdm_2. Les deux configurations générées sont illustrées sur la Figure 5.11 ; Elles sont ordonnancées dans un microprogramme pour la réception des données de l'acteur Sink.

```
icc0_0() {
  channel         1
  num_occ         0
  sel_credit      0
  path_to_target  2 SOUTH NORTH
  credit_size     8
  total_credit_nb 4200
}
```

```
icc0_1() {
  channel         1
  num_occ         0
  sel_credit      1
  path_to_target  2 SOUTH NORTH
  credit_size     8
  total_credit_nb 4200
}
```

FIGURE 5.11 – Pseudo-code de communication de l'unité *sme_10w*.

D'autre part, le support de schémas de communications complexes autorise de nouvelles optimisations. En effet, ce mécanisme est suffisamment expressif pour supporter les acteurs dédiés à la manipulation de données tels que le découpage et la fusion de flux de données (ex. *split/merge*). Dans le cas d'un découpage de données, l'acteur est fusionné avec le producteur des données ou avec le consommateur des données dans le cas d'une fusion de données. Cette optimisation tire parti des spécificités matérielles de la plateforme Magali de manière transparente pour le développeur.

L'utilisation d'un modèle de calcul paramétrique implique la communication d'un nombre paramétrique de données. L'utilisation de paramètres pose le même problème que pour les cœurs de calcul : les configurations ne supportent pas les paramètres variables. Contrairement aux configurations de calcul, les communications sont cependant divisibles. Cette propriété rend possible l'utilisation d'une seule configuration envoyant un nombre statique de données, configuration répétée un nombre paramétrique de fois dans un microprogramme.

Ces différentes techniques spécialisent les communications à la plateforme Magali. Une fois spécialisées, le compilateur génère les communications entre les unités de calcul. La communication entre les unités de calcul utilise un routage XY. Dans ce routage, la donnée est d'abord transférée horizontalement, puis verticalement. Ce routage simple a l'avantage d'être déterministe et de garantir l'absence d'interblocage.

Les communications de grande taille sont découpées en paquets, en fonction de la taille des FIFO présentes sur les unités et de la taille totale des données à envoyer. Ce découpage est généré de manière cohérente entre émetteur et récepteur, ceci pour éviter un interblocage. En effet, si le récepteur envoie un nombre de crédits insuffisant par rapport à la taille du paquet à transmettre par l'émetteur, la communication se retrouve bloquée. Ce découpage est facile à évaluer lors d'une communication simple entre deux acteurs. Le placement et la fusion de plusieurs acteurs sur la même unité peut engendrer des interblocages difficiles à évaluer. Dans ce cas, la vérification des tailles mémoires décrite dans la Section 5.4.1 permet de simuler toutes les communications, et d'assurer l'absence d'interblocage.

5.4.4 Génération du contrôle

L'ordonnancement sur la plateforme Magali est découpé en deux parties : les contrôleurs distribués supportent un ordonnancement statique ; le contrôleur centralisé prend en charge la synchronisation des unités de calcul et des paramètres. Afin de mieux comprendre ce découpage, étudions-le à la lumière de l'application de la Figure 5.12.

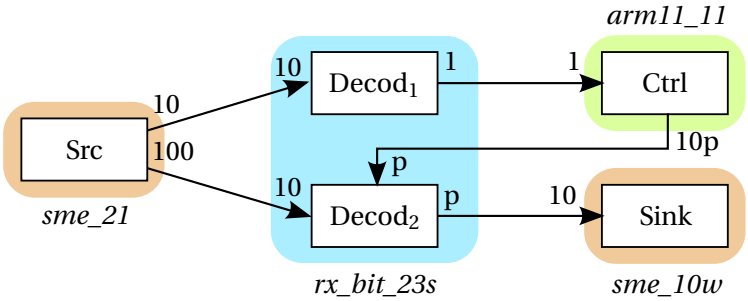


FIGURE 5.12 – Exemple de protocole dynamique placé sur la plateforme Magali.

Le compilateur génère l'ordonnancement quasi-statique (5.4), dont une partie est paramétrique.

$$S_{sme\_21} = (\text{Src}) \qquad S_{arm11\_11} = \left(\text{Ctrl}; \text{set}(p)\right) \qquad S_{sme\_10w} = \left(\text{get}(p); (\text{Sink})^{p}\right)$$
$$S_{rx\_bit\_23s} = \left(\text{Decod}_1; \text{get}(p); (\text{Decod}_2)^{10}\right) \tag{5.4}$$

Cet ordonnancement est découpé entre le contrôleur centralisé et les contrôleurs distribués. Pour rappel, les contrôleurs distribués CCC ont été présentés dans la Section 5.2.1. Ils permettent d'exécuter une séquence de configurations, et d'itérer sur cette séquence selon une valeur fixe. Le compilateur alloue tout l'ordonnancement connu statiquement au contrôleur distribué. L'objectif de cette allocation est de minimiser la charge du contrôleur centralisé ainsi que le nombre de synchronisations.

Les Figures 5.13 et 5.14 illustrent le contrôle pour les unités de calcul *sme_10w* et *rx_bit_23s*. Le contrôle est distribué entre le contrôleur centralisé `arm` et le contrôleur distribué `ccc`. Le contrôleur centralisé démarre (**`start`**) les contrôleurs distribués, il attend la fin de leurs exécutions (**`wait`**), et synchronise les paramètres (**`set`** et **`get`**).

```
armSme10w(){
  get p
  start cccSme(p)
  wait cccSme
}
```

```
cccSme10w(p){
  for i=1 to p
    Sink()
}
```

FIGURE 5.13 – Pseudo-code de contrôle de l'unité de calcul *sme_10w*.

Les CCC ne supportent pas nativement les applications paramétriques. Deux stratégies permettent de les adapter à ce type d'application. Si un paramètre influe sur le nombre d'exécutions d'une configuration, alors la borne de la boucle itérant sur la configuration est

placée dans un registre. Ce registre est modifié par le contrôleur centralisé avant le démarrage du microprogramme. C'est le cas de l'unité *sme_10w*, dont le contrôle est illustré sur la Figure 5.13.

```
armRxBit(){
  start cccRxBit1()
  wait cccRxBit
  get p
  if p=1
    start cccRxBit2()
  if p=2
    start cccRxBit3()
  if p=3
    start cccRxBit4()
  wait cccRxBit
}
```

```
cccRxBit1(){
  Decod1(1)
}

cccRxBit2(){
  for i=1 to 10
    Decod2(1)
}
```

```
cccRxBit3(){
  for i=1 to 10
    Decod2(2)
}

cccRxBit4(){
  for i=1 to 10
    Decod2(3)
}
```

FIGURE 5.14 – Pseudo-code de contrôle de l'unité de calcul *rx_bit_23s*.

Si un paramètre influe sur une configuration, un jeu de configurations est généré pour chaque valeur du paramètre. Chaque configuration est associée à un microprogramme, et le contrôleur centralisé choisit le microprogramme à exécuter en fonction de la valeur du paramètre. C'est le cas de l'unité *rx_bit_23s*, illustrée sur la Figure 5.14.

Le processeur ARM prend en charge à la fois des opérations de calcul, mais aussi le contrôle centralisé de l'ensemble des unités de calcul. Pour chaque unité de calcul, un code de contrôle indépendant est généré. Tous ces codes sont exécutés en concurrence sur le processeur ARM, sans connaissance à priori de l'antériorité.

L'ordonnancement des différents codes de calcul et de contrôle est supporté par le système d'exploitation eCos [eCos 14]. Chaque unité de calcul est associée à un fil d'exécution en charge du contrôle de cette unité, la synchronisation se faisant par une interruption associée à un sémaphore. Un fil d'exécution supplémentaire réalise les traitements ordonnancés sur le processeur ARM. La communication des paramètres entre les fils d'exécution utilise des variables conditionnelles.

La séparation du contrôle de chaque unité de calcul vise à conserver tout le parallélisme de l'application, sans introduire de contraintes d'ordonnancement dans le contrôleur. Cette approche se différencie de l'approche par phases qui séquentialise l'exécution de chacune des phases. Le modèle d'exécution s'appuie sur l'ordonnanceur du système d'exploitation pour des raisons de temps de développement. Les performances de ce modèle d'exécution sont discutées dans la Section 6.3.3.

5.5 Travail réalisé

Le développement de ce *back end* a demandé une étude approfondie des mécanismes matériels de la plateforme Magali. Il s'appuie sur l'infrastructure de compilation LLVM pour la génération de code. Ce travail a nécessité l'étude de la structure de génération de code de LLVM pour y intégrer la génération de code vers la cible Magali. Il a été réalisé en 4 mois durant ma thèse en collaboration avec un stagiaire ingénieur. Le développement du compilateur complet, en intégrant le *front end*, représente près de 15000 lignes de code C++. Son évaluation

a été réalisée à l'aide de 6 applications compilées et exécutées sur la plateforme Magali. Parmi ces applications, 3 sont extraites du protocole LTE et présentées dans le prochain chapitre.

5.6 Discussion

Ce chapitre a traité du *back end* du compilateur et de la génération de code pour Magali. La plateforme matérielle Magali est décrite en détails, avec une présentation de l'ensemble des unités de calcul, des mécanismes de communication et de contrôle. Les modèles de programmation existants pour cette plateforme sont aussi présentés pour connaître l'état de l'art actuel du développement sur Magali.

Dans un deuxième temps, les techniques d'optimisation et de spécialisation de la représentation intermédiaire sont présentées. Le problème du placement est laissé au développeur, le travail se concentrant sur l'optimisation de l'application pour la plateforme. Dans cet objectif, je propose la fusion d'acteurs comme support des accélérateurs de calcul. Dans ce travail, la fusion est implémentée de manière spécifique pour chaque accélérateur dans le *back end* de Magali. L'automatisation de cette spécification dans le compilateur est une étape importante dans l'adoption de cette technique. Cette automatisation devra se faire en association avec le placement automatique des acteurs sur la plateforme.

Les vérifications sur la représentation intermédiaire sont aussi présentées dans cette section. En particulier, le micro-ordonnancement est appliqué au contrôle des tailles mémoires. Il est utilisé à plusieurs fins pour la plateforme Magali. L'ordonnancement des acteurs est évalué à la lumière de l'ordre de production et de consommation au sein de l'acteur. Le découpage des communications en paquets est aussi vérifié pour éviter les famines.

Enfin, la génération du code pour Magali décrit la prise en charge des cœurs de calcul hétérogènes, des communications entre ces unités, ainsi que le modèle d'exécution supportant le contrôle de l'application. La génération de code pour une plateforme hétérogène est un point sensible adressé dans ce travail. Les unités configurables sont supportées par un *back end* dédié. Dans le cas des unités programmables, il n'est pas raisonnable de développer un nouveau *back end*, ce qui pose le problème de l'intégration d'un *back end* existant dans le flot de compilation. L'utilisation de LLVM rend cette intégration dépendante du support de la représentation intermédiaire par le *back end* à intégrer.

La génération des communications est quant à elle facilitée par l'abstraction du modèle de calcul flot de données. Toutes les communications sont explicites et représentées dans un format de haut niveau. Le compilateur tire profit de ces informations pour générer les configurations pour l'ensemble des communications sur la plateforme de manière automatique. Le contrôle des applications est aussi généré automatiquement. Il utilise l'ensemble des capacités des contrôleurs distribués pour limiter la charge du contrôleur centralisé.

Les paramètres variables sont supportés par l'ensemble de la génération de code. Ce support n'est pas natif à la plateforme Magali, et nécessite l'ajout d'un certain nombre de mécanismes logiciels. En particulier, ce travail se distingue par la génération automatique de jeux de configurations couvrant l'ensemble des valeurs des paramètres, et des codes de contrôle associés. L'utilisation de plusieurs paramètres par unité de calcul peut augmenter le nombre de configurations de manière importante. Un développement possible serait alors la *compilation croisée dynamique* de ces configurations. Les contraintes applicatives rendent l'intégration d'un compilateur sur la plateforme Magali irréaliste à l'heure actuelle. La recherche sur la compilation à la volée pour les systèmes embarqués [Carbon 13] et la génération de spécialiseur de programmes [Lomüller 14] pourraient amener à reconsidérer la question.

Le prochain chapitre présentera les expérimentations réalisées sur la plateforme Magali. Ces expérimentations se basent sur le protocole de télécommunication LTE. L'analyse des résultats de performance mettra en avant les avantages et les limitations du compilateur. Elle permettra de valider le travail réalisé sur le compilateur.

6 Expérimentations et résultats

À partir de maintenant on travaille au chrono. Parce que une minute d'écart veut pas dire forcément soixante secondes. Ça peut se transformer en années de placard. Crois-moi, j'connais la question.

— Jean Gabin comme M. Charles, *Mélodie en sous-sol*

Sommaire du chapitre

6.1 Introduction

Les chapitres précédents ont traité du flot de compilation développé durant cette thèse. Nous avons vu que ce compilateur utilise le format de haut niveau PaDaF en entrée, une représentation intermédiaire basée sur l'IR LLVM, et génère le code assembleur pour la plateforme Magali. La question auquel ce chapitre vise à répondre est comment mesurer les apports de ce compilateur par rapport aux travaux existants ? Pour cela, je propose de mesurer d'une part l'apport du format de haut niveau proposé sur la programmation et d'autre part l'apport du compilateur sur les performances. Il est important de définir une référence pour évaluer ces apports. Ce chapitre présente des éléments de réponses en deux parties.

La Section 6.2 traitera des applications utilisées comme références dans ce travail. Il est tout d'abord nécessaire de définir les caractéristiques à évaluer. Ces caractéristiques doivent refléter les contraintes du domaine des télécommunications. Pour cela, je présenterai le protocole LTE et justifierai son choix comme référence pour l'évaluation de ce travail. Je présenterai ensuite les applications extraites du protocole LTE et leurs caractéristiques. Enfin je discuterai de l'apport de la programmation des applications dans le format PaDaF.

La Section 6.3 s'appuiera sur ces applications pour évaluer les performances du compilateur. Dans cet objectif, j'évaluerai dans un premier temps le coût du contrôle des tailles mémoires durant le processus de compilation. La bonne distribution et synchronisation du contrôle est un élément critique dans l'exécution d'applications sur des plateformes distribuées telles que Magali. En prélude de l'analyse des performances, je présenterai les différents modèles de contrôle, et les profils d'exécution que l'on peut attendre selon les caractéristiques de l'application et de la plateforme. J'analyserai enfin les performances des applications sur la plateforme Magali en fonction du modèle de contrôle.

6.2 Applications de référence

Dans cette section, je présente les applications de référence utilisées pour l'évaluation du compilateur. Ces applications ont pour but d'évaluer l'adéquation du flot de compilation avec le domaine des télécommunications. Pour remplir cet objectif, il est nécessaire d'utiliser une référence dans ce domaine. Cette référence doit présenter les caractéristiques existant dans l'essentiel des protocoles, mais aussi celles attendues dans les futurs protocoles pour assurer la pérennité de l'approche. Dans ce travail, j'ai fait le choix du protocole LTE comme référence. En effet, le protocole LTE (c.-à-d. *Long Term Evolution*) définit le cadre pour les protocoles de télécommunications de quatrième génération. Il constitue donc une référence représentative pour l'évaluation de ce travail, et offre de bonnes perspectives sur la compatibilité du compilateur avec les futurs protocoles. Je présente le protocole LTE dans la Section 6.2.1, en particulier la couche physique du lien descendant qui nous intéresse pour la suite.

À partir de ce protocole, il est nécessaire d'extraire des scénarios pour isoler les difficultés d'implémentation du protocole. Ces scénarios constituent autant d'applications, et chaque application reflète des caractéristiques particulières à évaluer. Je présenterai les 3 applications choisies et leurs caractéristiques dans la Section 6.2.2. Le développement de ces applications est réalisé au format PaDaF. Je proposerai une évaluation des gains apportés par ce format dans la Section 6.2.3, sur la base des applications développées et de l'expérience de développement du CEA sur la plateforme Magali.

6.2.1 Protocole LTE

LTE est un standard de télécommunications développé par le 3GPP. La Figure 6.1 représente une trame LTE (*release* 8) à décoder sur l'équipement mobile. Une trame LTE est composée de 10 sous-trames de 1 ms chacune. Chaque sous-trame est composée de 14 symboles temporels. Chaque symbole est découpé en sous-porteuses en fréquence, jusqu'à 2048 pour une bande passante de 20 MHz. La transmission de données utilise le MIMO, avec jusqu'à 4 antennes en émission sur la station de base et 2 antennes pour la réception sur l'équipement mobile. Pour le lecteur intéressé, le rapport de Zyren et McCoy [Zyren 07] constitue une bonne introduction aux concepts physiques utilisés dans le protocole LTE. Le rapport de Agilent [Agilent 09] propose une présentation plus avancée du protocole LTE.

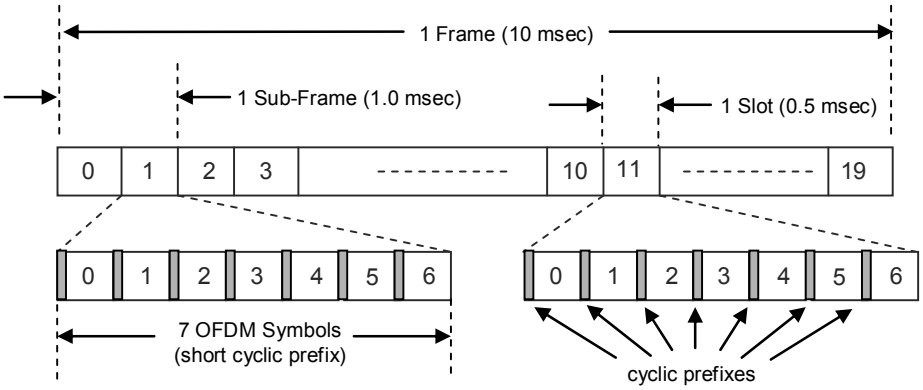


FIGURE 6.1 – Structure générique d'une trame LTE (extrait de [Zyren 07]).

Ce travail se concentre sur l'implémentation du protocole sur l'équipement mobile. En particulier, je détaille les contraintes sur le lien descendant de la catégorie 5 du protocole LTE, dont les principales caractéristiques sont présentées dans la Table 6.1.

Catégorie de l'équipement	Débit max. (Mbps)	Nombre d'antennes (émission × réception)
catégorie 1	10	1 × 2
catégorie 2	50	2 × 2
catégorie 3	100	2 × 2
catégorie 4	150	2 × 2
catégorie 5	300	4 × 2

TABLE 6.1 – Débit descendant maximum par catégorie en LTE (*Release 8*).

Le LTE illustre plusieurs caractéristiques présentes dans les protocoles de télécommunications, qui sont autant de problèmes à résoudre lors de la spécification et de la compilation de ce protocole. Le premier problème vient de la complexité des protocoles. Ils mettent en jeu un grand nombre de traitements à réaliser, potentiellement parallèles dans le cas d'une réception MIMO. La Figure 6.2 illustre une vue générale de la couche physique de réception du LTE, avec des traitements parallèles pour le décodage du signal de deux antennes. Un grand nombre de ressources de calcul matérielles sont nécessaires pour effectuer ces traitements, comme

nous l'avons vu dans l'état de l'art du Chapitre 2. Lors de leur programmation, la contrainte essentielle est de tirer parti du parallélisme de données et de tâches de ces protocoles pour atteindre l'objectif de performance. Ce problème est au cœur du domaine de la radio logicielle, comme présenté par Woh *et al.* [Woh 07], et appartient plus généralement au problème de la programmation parallèle.

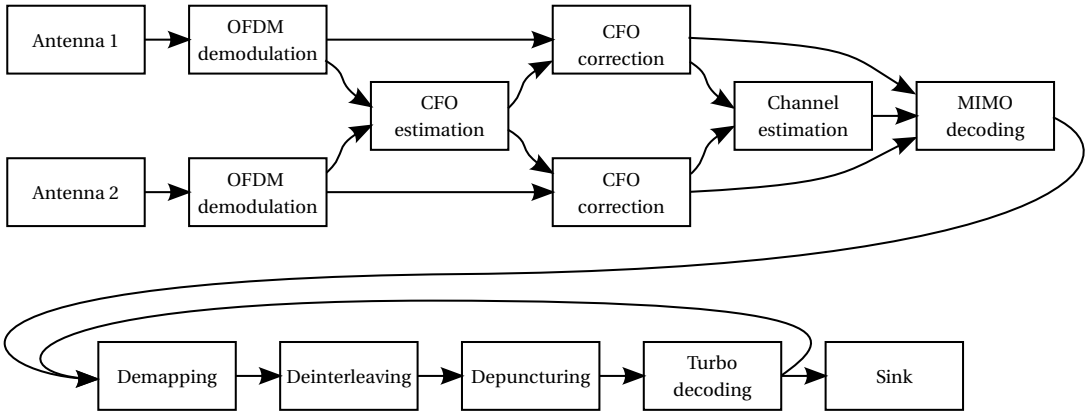


FIGURE 6.2 – Vue d'ensemble de la couche physique de réception du LTE.

En plus des communications de données entre les blocs de traitements, le domaine des télécommunications implique un grand nombre de manipulations de données au sein des blocs de traitements. Ce deuxième problème est aussi décrit par Woh *et al.* [Woh 07] dans son analyse du LTE. Dans cette analyse, ils modélisent un traitement comme une suite d'opérations impliquant le chargement de données, la permutation de ces données, la réalisation de calculs sur ces données, et le stockage de ces données. Il est important de noter dans leur modèle de traitement la présence de manipulation de données, ici appelé permutation, comme un élément constituant tout traitement.

Un exemple de ce problème dans le protocole LTE vient de la correction du décalage de fréquence CFO (Carrier Frequency Offset). Cette correction est réalisée par analyse des signaux de référence, aussi appelés *pilotes*, placés dans chaque sous-trame. Le placement de ces pilotes est illustré sur la Figure 6.3. La solution à ce problème est généralement de réaliser la manipulation de données dans un langage généraliste comme le C++. Cette solution restreint cependant les manipulations de données aux blocs supportant la génération de code depuis un langage généraliste. Le format PaDaF complète cette approche par plusieurs opérations de manipulation (par ex. filtrage de données et désentrelacement) dans son API. Cette solution n'est pourtant pas complètement satisfaisante, car elle limite l'utilisation de DMA intelligents à ces quelques opérations. Le développement d'une abstraction des manipulations de données est un élément crucial pour la radio logicielle, mais hors du champ d'étude de cette thèse.

Le troisième problème posé est le dynamisme des nouveaux protocoles. En effet, ces protocoles visent à adapter leur transmission à la qualité du canal. Cette adaptation peut passer par une allocation de ressources dynamique et un codage de canal variable, et peut aller jusqu'à une reconfiguration complète du protocole. Dans le cas du protocole LTE, les informations d'allocation de ressources et de codage de canal sont transmises lors de chaque sous-trame. Une sous-trame LTE est découpée en trois canaux logiques qui sont le canal de format, le canal contrôle et le canal partagé, illustrés sur la Figure 6.3. Le canal de format

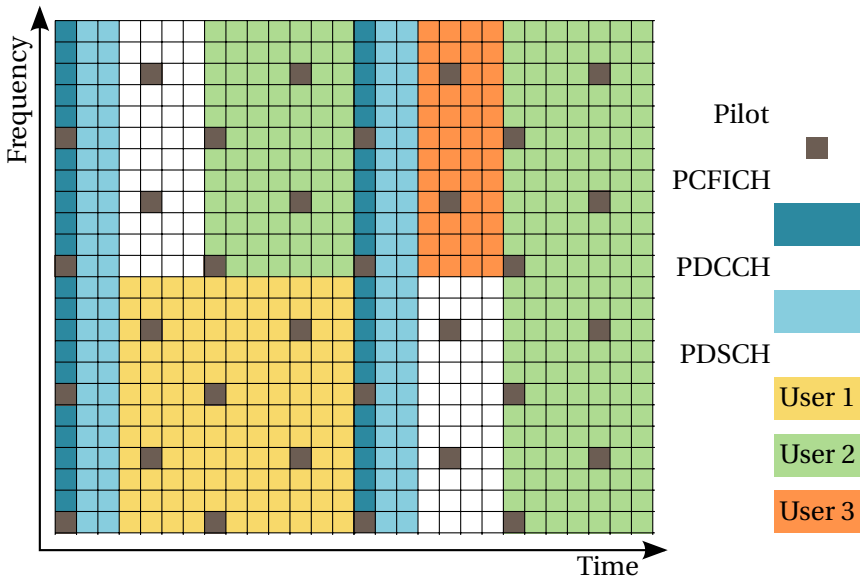


FIGURE 6.3 – Allocation de ressources en LTE.

PCFICH (*Physical Control Format Indicator CHannel*) encode le format du canal de contrôle. Le canal de contrôle PDCCH (*Physical Downlink Control CHannel*) encode l'allocation de ressources de chacun des utilisateurs et le codage des données. Le canal de données, ou canal partagé PDSCH (*Physical Downlink Shared CHannel*), contient les données allouées pour chacun des utilisateurs. Il est donc nécessaire de décoder les différents canaux logiques, et en fonction des valeurs décodées de reconfigurer la suite du traitement de manière dynamique. Cette dépendance entre les canaux est représentée par une boucle de rétroaction dans la couche physique du LTE sur la Figure 6.2.

6.2.2 Propriétés des applications

Nous venons de voir les difficultés à surmonter pour implémenter la couche physique d'un protocole radio, illustrées avec le protocole LTE. Pour évaluer le compilateur face à ces difficultés, je propose trois applications qui illustrent chacune un point particulier.

Application OFDM L'application OFDM, présentée sur la Figure 6.4, est un test simple des capacités d'adaptation du compilateur. Elle illustre la spécialisation d'une application abstraite pour la plateforme Magali, avec la fusion de deux acteurs sur l'unité de calcul *trx_ofdm_20*.

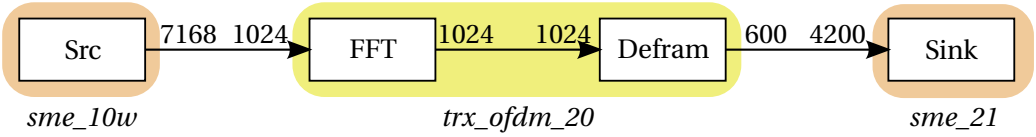


FIGURE 6.4 – Application OFDM placée sur Magali.

Application démodulation L'application démodulation, présentée sur la Figure 6.5, comprend plus d'acteurs et d'unités de calcul. La difficulté mise en avant est le nombre de communications entre les unités de calcul. Cette application évalue les capacités du compilateur à ordonnancer et à générer un code correct pour l'ensemble des acteurs automatiquement.

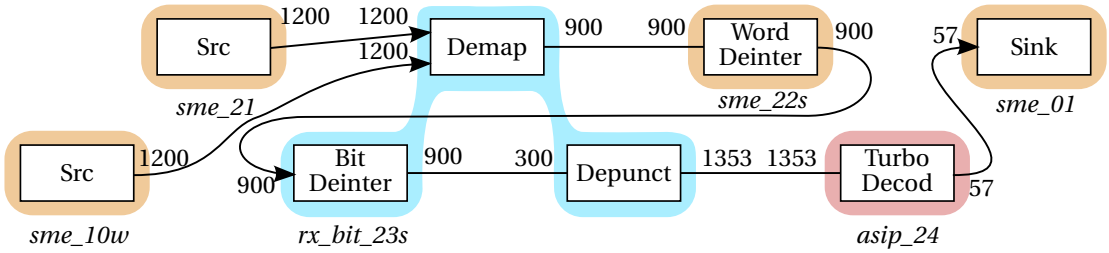


FIGURE 6.5 – Application démodulation placée sur Magali.

Application démodulation paramétrique L'application démodulation paramétrique, présentée sur la Figure 6.6 étend l'application précédente par l'ajout d'un paramètre. Elle représente le décodage des canaux logiques dans le protocole LTE. Le résultat du décodage du canal de contrôle (partie supérieure de la Figure 6.6) définit la modulation (paramètre p) pour le décodage du canal de données (partie inférieure de la Figure 6.6). Elle permet l'évaluation de la gestion des paramètres sur une application importante.

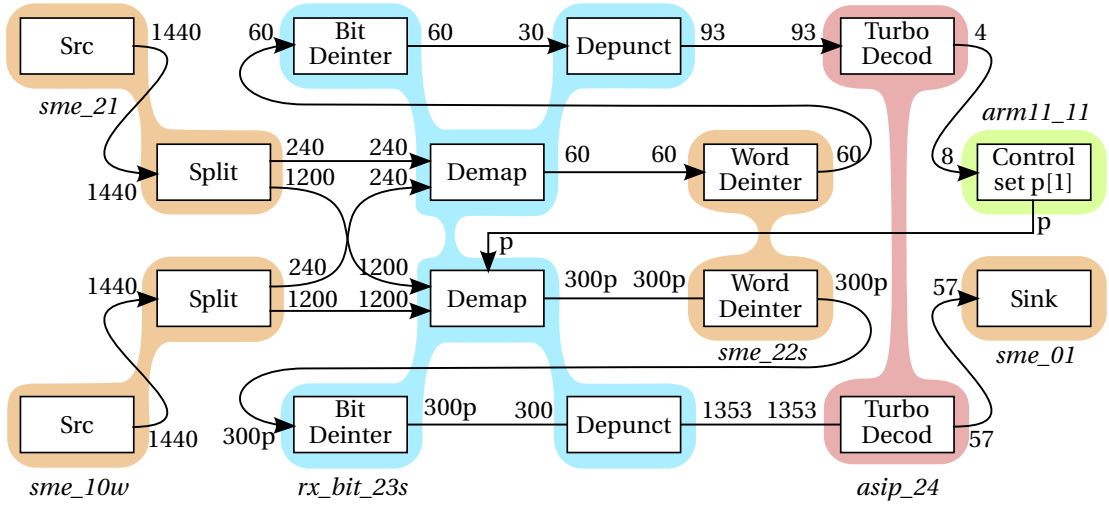


FIGURE 6.6 – Application démodulation paramétrique placée sur Magali.

6.2.3 Développement des applications

Les bénéfices en termes de complexité et de temps de développement à utiliser le compilateur sont décrits dans la Table 6.2. Ces résultats sont basés sur l'expérience des développeurs Magali. Le temps nécessaire pour écrire une application est une métrique subjective, car il

inclut le temps de réflexion, difficile à juger, et qu'il est très dépendant du développeur. Cependant, lorsque les applications sont écrites par des personnes ayant les mêmes compétences techniques et avec les mêmes connaissances de la plateforme matérielle et des protocoles radio, il donne une estimation pertinente des bénéfices à utiliser l'outil fourni. La taille du code pour la plateforme Magali est séparée entre le code C pour le contrôleur central ARM et le code assembleur pour le contrôle distribué. Le nombre de lignes relativement faible par rapport au temps de développement dans l'approche native est dû à la complexité inhérente à la programmation de la plateforme : le contrôle distribué requiert la configuration de différents blocs matériels indépendants avec des valeurs globalement cohérentes qui représentent dans leur ensemble l'application. Sans un outil dédié, assurer — et déboguer — cette cohérence globale est source d'erreurs pour le programmeur.

	PaDaF		Natif	
Applications	C++ (#lignes)	(heures)	C / ASM (#lignes)	(semaines)
OFDM	60	1	150 / 200	1
Démodulation	160	4	300 / 600	4
Démod. paramétrique	260	8	500 / 800	12

TABLE 6.2 – Comparaison du développement au format PaDaF et natif sur Magali.

En conséquence, alors que la taille du code généré par le compilateur est globalement équivalente à celle du code natif, la taille initiale du code PaDaF est divisée par 5 et le temps de développement par environ 40. Un autre avantage est que le code PaDaF est indépendant de la plateforme, et peut être compilé vers d'autres plateformes. Le développement récent d'une évolution de Magali pour l'intégration 3D [Dutoit 13] constitue une bonne illustration. Du point de vue programmation, cette évolution est principalement architecturale, avec le déplacement d'unités de calcul sur le réseau. Cette modification a toutefois rendu les applications développées nativement obsolètes. Le modèle de la plateforme Magali 3D est utilisé pour ces expérimentations de manière transparente pour le compilateur, et souligne l'intérêt de s'abstraire de la plateforme.

6.3 Évaluation des performances

Nous avons vu dans la section précédente le protocole LTE qui sert de référence pour l'évaluation du compilateur. À partir de ce protocole, j'ai défini plusieurs applications pour évaluer l'effet de différentes caractéristiques du protocole sur la compilation et l'exécution. Je présente dans cette section l'évaluation proprement dite. Cette évaluation se fera en deux temps, avec d'une part la compilation des applications et d'autre part l'exécution de celles-ci.

Pour évaluer le compilateur, j'ai compilé les applications de test présentées dans la section précédente. Cette évaluation concerne l'ensemble du compilateur, tel que présenté dans les Chapitres 4 et 5. Le temps de compilation pour ces applications est négligeable (moins de 1 s). Le temps de vérification des tailles mémoires est cependant plus important. Je reviendrai sur ce temps de vérification dans la Section 6.3.1.

Une fois l'application compilée, la deuxième évaluation concerne l'exécution sur la machine cible. La plateforme Magali utilise des mécanismes d'accélération matérielle pour le calcul et les communications. Le *back end* du compilateur spécialise les applications de test

pour tirer parti de ces mécanismes. Le contrôle sur la plateforme Magali est réparti entre le contrôleur centralisé et les contrôleurs distribués. Le choix de cette répartition est laissé au développeur, ce qui rend possible différentes stratégies et influence le niveau de performances des applications. J'analyserai différents modèles de contrôle utilisés sur Magali dans la Section 6.3.2, et leurs performances temporelles dans la Section 6.3.3.

6.3.1 Évaluation du contrôle des tailles mémoires

La vérification des tailles mémoires s'appuie sur un ensemble de techniques présentées tout au long de cette thèse. J'ai introduit dans la Section 3.4 du Chapitre 3 le formalisme micro-ordonnancement. Ce formalisme vise à représenter les communications entre acteurs dans un graphe flot de données, et à les ordonner. J'ai ensuite appliqué ce formalisme à la vérification des tailles mémoires sur les graphes flot de données paramétriques. Chacune des applications de test a été placée et optimisée pour la plateforme Magali, puis modélisée avec le langage PROMELA selon les contraintes présentées dans la Section 5.4.1. Le *model checker* SPIN parcourt ce modèle pour assurer l'absence d'interblocage dû aux communications.

Les techniques de *model checking* peuvent être limitées par des problèmes de complexité lors de l'exploration de larges espaces d'états. Pour évaluer cette complexité, les résultats de simulation utilisant le *model checker* SPIN sont présentés dans la Table 6.3. Ces simulations ont été exécutées sur la machine hôte avec un processeur Intel Core i5 à 2.4 GHz avec 8 Go de RAM et le système d'exploitation OS X 10.9.2. Les boucles `for` dans le code PROMELA sont déroulées pour réduire le nombre d'états d'un facteur 10. Comme attendu, la complexité augmente avec la taille du graphe flot de données ainsi que la présence de paramètres. Toutes les applications sont toutefois explorées dans un temps raisonnable.

Application	États	Transitions	Temps d'exécution (s)
OFDM	1.28×10^4	2.56×10^4	0.1
Démodulation	2.12×10^6	1.07×10^7	9
Démod. paramétrique	6.07×10^7	2.22×10^8	199

TABLE 6.3 – Temps d'exécution du contrôle des tailles mémoires.

Sans cette méthode, la programmation sur la plateforme Magali se fait par tests successifs pour résoudre les interblocages. Ce procédé complexe est coûteux en temps, et de plus ne permet pas de trouver tous les interblocages. Cette méthode apporte donc une analyse des tailles mémoires formelle absente du développement sur la plateforme Magali. Elle est cependant limitée, la complexité augmentant de manière exponentielle avec la taille de l'application. En prolongement de ce travail, il serait intéressant d'expérimenter d'autres modélisations pour réduire cette complexité. La programmation linéaire, utilisée par Bodin *et al.* [Bodin 14] pour minimiser la taille des mémoires dans un graphe flot de données, serait une solution à évaluer dans le futur.

6.3.2 Analyse des modèles de contrôle

Le contrôle des applications sur Magali est partagé entre le contrôleur central et les contrôleurs distribués. Le contrôleur central est nécessaire pour la synchronisation de plusieurs unités de calcul et le démarrage de ces unités. Au delà de cette contrainte, le contrôle est librement réparti par le programmeur.

Nous avons vu dans la Section 5.2.2 que la programmation native utilise le modèle de contrôle par phase. Dans ce modèle de contrôle, l'application est découpée en plusieurs graphes, chaque graphe étant une *phase*. Le contrôleur central se charge du démarrage de chaque phase et de la synchronisation entre ces phases. Le code généré par le compilateur utilise quant à lui un modèle de contrôle où chaque unité de calcul est associée à un fil d'exécution sur le contrôleur centralisé. Le contrôleur centralisé se charge de démarrer chaque unité de calcul et de la synchronisation entre ces unités. Chaque modèle de contrôle possède une granularité différente, qui a ses avantages et ses limitations. Je propose de les analyser pour mieux les comprendre et savoir interpréter les résultats de performances temporelles dans la section suivante.

Le chronogramme de la Figure 6.7 illustre le fonctionnement d'une application par phases. Pour chaque unité de calcul, le chronogramme représente les périodes pendant lesquelles une unité est activée, c'est à dire depuis le démarrage de l'unité par le contrôleur central jusqu'à son arrêt. Les temps t_{si} et t_{ei} sont respectivement le temps de démarrage et d'arrêt de l'unité i. Pour le contrôleur central (CPU), le chronogramme représente le temps δ_{si} de reconfiguration d'une unité de calcul i.

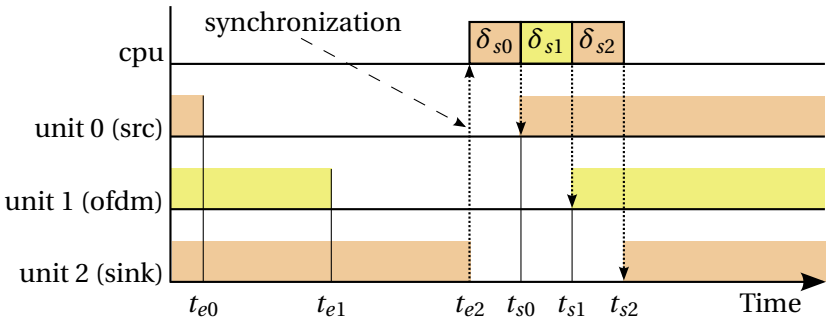


FIGURE 6.7 – Chronogramme d'une application contrôlée par phase.

Un unique point de synchronisation (trait pointillé à t_{e2}) est utilisé pour le fonctionnement par phases. Le contrôleur central est inactif depuis le démarrage de la dernière unité de calcul jusqu'à ce point de synchronisation. Il existe donc une opportunité de reconfiguration en temps masqué. Une opération en temps masqué est une opération réalisée en parallèle d'une autre opération dans un temps inférieur ou égal à celle-ci. Dans le contexte de cette analyse, je suppose un graphe linéaire, c.-à-d. avec un ordre absolu entre les unités de calcul. Le temps masqué est alors le temps de non recouvrement entre deux unités consécutives, c'est à dire le temps pendant lequel au moins une unité est inactive alors que l'unité suivante est encore active. Je désigne ce temps de non recouvrement par la suite comme $\delta_{e0} = t_{e1} - t_{e0}$.

La reconfiguration en temps masqué vise à réduire le temps de reconfiguration de l'application, ce temps pouvant représenter jusqu'à 10 % du temps d'exécution de l'application [Clermidy 09a]. Je définis le temps total de reconfiguration comme la somme des temps pendant lesquels une unité de calcul n'est pas en cours d'exécution. Dans le cas du contrôle par phase, ce temps est égal à la somme des temps de non recouvrement et des temps de reconfiguration (6.1).

$$\delta_{\text{c phase}} = \sum_{i=0}^{n-1} \delta_{ei} + \sum_{i=0}^{n} \delta_{si} \qquad (6.1)$$

$$\delta_{\text{c unit}} = \sum_{i=0}^{n-1} \delta_{si} \qquad (6.2)$$

Le code généré par le compilateur utilise un modèle de contrôle à la granularité de l'unité de calcul pour tirer parti de cette opportunité de calcul en temps masqué. Dans ce modèle de contrôle par unité, chaque unité est reconfigurée dès la fin de son exécution. Le temps de reconfiguration total est alors égal à la somme des temps de reconfiguration (6.2).

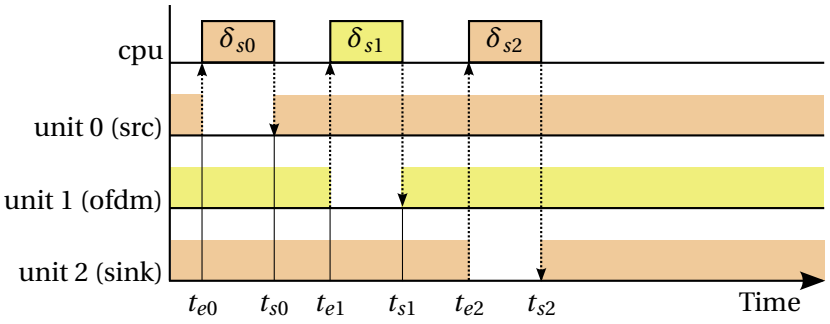


FIGURE 6.8 – Chronogramme d'une application contrôlée par unité ($\delta_{ei} > \delta_{si}$).

Le chronogramme de la Figure 6.8 montre le fonctionnement du contrôle par unité lorsque le temps de non recouvrement est supérieur au temps de reconfiguration. Dans ce cas, nous obtenons bien un gain sur le temps d'exécution, qui tire parti du temps précédemment inutilisé à cause de la synchronisation de l'approche par phase.

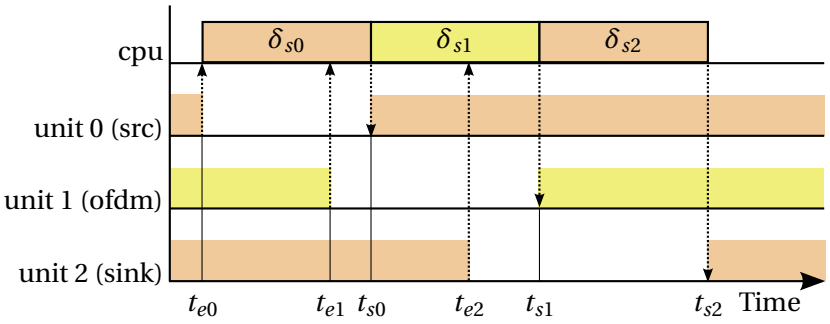


FIGURE 6.9 – Chronogramme d'une application contrôlée par unité ($\delta_{ei} < \delta_{si}$).

Le chronogramme de la Figure 6.9 illustre quant à lui le cas où le temps de non recouvrement est inférieur au temps de reconfiguration. On notera par ailleurs que le temps de reconfiguration de chaque unité est supérieur dans le contrôle par unité par rapport au contrôle par phase. Ceci est dû au changement de contexte entre les fils d'exécution du contrôleur central. Dans ce cas le contrôle par unité peut se révéler désavantageux.

Les temps qui composent le calcul des temps de reconfiguration (6.1) et (6.2) sont dépendants du modèle de contrôle utilisé et de l'application. On ne peut pas conclure à partir des valeurs théoriques quant à l'approche la plus appropriée. Il est donc nécessaire de confronter ces modèles de contrôle à l'expérimentation.

6.3.3 Évaluation des performances temporelles

En prélude de l'évaluation des performances temporelles, il est important d'en préciser le cadre. L'environnement d'expérimentation utilisé est l'infrastructure de simulation de la plateforme Magali. Cette infrastructure repose sur un modèle transactionnel (TLM) en SystemC de Magali. Les temps d'exécution du modèle TLM sont extraits des unités de calcul après synthèse en technologie CMOS 65 nm. Le contrôleur central ARM est exécuté dans une machine virtuelle QEMU connectée au modèle TLM de la plateforme. La synchronisation entre le modèle TLM et la machine virtuelle QEMU est faite à la granularité du *Transaction Level Block*.

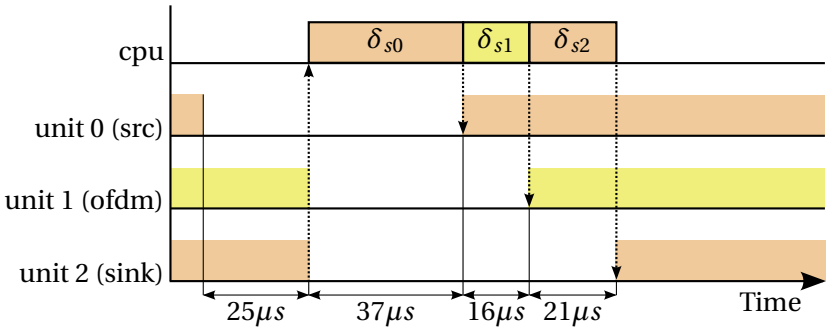


FIGURE 6.10 – Chronogramme de l'application OFDM avec contrôle par phases.

Pour analyser les performances temporelles, nous avons vu dans la section précédente l'influence des modèles de contrôle sur le temps de reconfiguration des unités de calcul. Je propose dans cette section de confronter ces modèles à l'expérimentation, et présente les résultats d'exécution sous la même forme de chronogrammes. La Figure 6.10 illustre le contrôle par phase et la Figure 6.11 le contrôle par unité. Nous observons que le temps de non recouvrement est inférieur au temps de reconfiguration entre l'unité src et OFDM. Ce temps est négligeable (moins de 1μs) entre l'unité OFDM et sink, et n'est pas représenté sur les Figures 6.10 et 6.11. Ce temps de non recouvrement faible s'explique en partie par la faible taille mémoire entre les unités de calcul, une unité étant active jusqu'à la fin de la transmission de ses données.

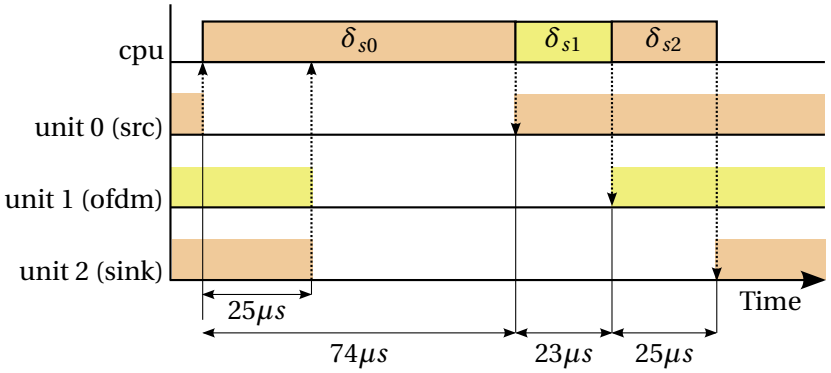


FIGURE 6.11 – Chronogramme de l'application OFDM avec contrôle par *threads*.

Le temps total de reconfiguration est de 99μs dans le cas d'un contrôle par phase, contre 122μs pour le contrôle par unité. Le faible temps de non recouvrement est une des raisons du surcoût du contrôle par unité. De plus, la prise en compte des interruptions des unités OFDM et sink par le contrôleur centralisé ralentit la reconfiguration de l'unité src. Au final, on observe donc un surcoût à l'utilisation du contrôle par unité, qui se reflète dans les résultats de performances présentés par la suite.

Les temps d'exécution des applications décrites dans la Section 6.2.2 sont présentés dans la Table 6.4. Le temps d'exécution d'une application est défini comme le temps depuis le démarrage de la première unité jusqu'à l'arrêt de la dernière unité ; le temps présenté est la moyenne de 10 exécutions consécutives. Le contrôle par phase correspond au développement natif — et coûteux en temps — réalisé sur la plateforme Magali. Ce développement est utilisé comme référence pour comparer les différentes solutions.

Application	Phase (μs)	RVM (μs)	Unité (μs)	Optimisé (μs)
OFDM	149	500 (+236%)	168 (+13%)	149 (+0%)
Démodulation	180	-	283 (+57%)	180 (+0%)
Démod. paramétrique	419	-	558 (+33%)	288 (-31%)

TABLE 6.4 – Temps d'exécution des applications selon le modèle de contrôle utilisé.

Le résultat sur le modèle de contrôle RVM [Ben Abdallah 10] montre l'importance d'utiliser le contrôle distribué, avec un surcoût de 236% pour la simple application OFDM. Le contrôle par unité correspond au code généré par le compilateur sans modification. Le surcoût du contrôle par unité est proportionnel au nombre d'unités de calcul pour les deux premières applications. Pour la troisième application, ce surcoût est équilibré par l'utilisation d'un seul graphe pour toute l'application, contrairement au contrôle par phase où l'application est découpé en deux graphes : le graphe statique et le graphe paramétrique. Dans le contrôle par unité, les unités de calcul Src ne sont pas influencées par le paramètre, ce qui réduit le nombre de reconfigurations.

En se basant sur les résultats précédents, je propose un contrôle optimisé. Ce contrôle utilise un seul fil d'exécution ; Il est équivalent à un contrôle par phase dans le cas d'une application statique. Si un paramètre est utilisé, un point de synchronisation est introduit uniquement pour les unités de la région du paramètre. De cette manière, il limite le nombre de blocs reconfigurés, permettant d'améliorer les performances par rapport au contrôle par phase. Cette optimisation est réalisée manuellement en modifiant le code C du contrôleur central généré par le compilateur. Son automatisation dans des travaux futurs permettrait la génération d'un contrôle plus adapté à la plateforme Magali.

6.4 Discussion

Dans ce chapitre, j'ai apporté des réponses à l'évaluation du compilateur développé durant cette thèse. Pour cette évaluation, je me suis appuyé sur le protocole LTE. Ce protocole, de par son objectif d'évolution à long terme, reflète les caractéristiques actuelles et à venir en télécommunications. J'ai extrait de ce protocole plusieurs applications pour l'évaluation du compilateur. Le développement des applications de test au format PaDaF démontre l'apport de ce format sur le temps de développement par rapport à l'approche native. Cet apport s'appuie sur l'ensemble des outils du compilateur, avec en premier lieu le contrôle des tailles

mémoires comme support de développement et de vérification des applications. Le coût temporel du contrôle des tailles mémoires est aussi évalué dans ce chapitre. Il se montre adapté au contrôle des applications actuelles, mais l'utilisation du *model checking* limite la taille des applications évaluables. L'utilisation d'autres modèles comme la programmation linéaire est proposé pour étendre cette vérification à des graphes de plus grande taille.

J'ai ensuite présenté les différents modèles de contrôle utilisés sur la plateforme Magali. En particulier, j'ai détaillé l'influence sur ces modèles du temps de non recouvrement et du temps de reconfiguration des unités de calcul. Ces modèles sont ensuite évalués sur le simulateur de la plateforme Magali. L'analyse des modèles de contrôle permet de comprendre les performances mitigées du compilateur. Sur la base de ces analyses, j'ai proposé un nouveau modèle de contrôle qui réduit le nombre de synchronisations. Ce modèle tire profit du modèle de calcul flot de données paramétrique utilisé par le compilateur, et apporte un gain de 31% par rapport à l'application native découpée en phases. Au final, le choix du modèle de contrôle dépend à la fois de la plateforme et de l'application. La génération automatique d'un contrôle adapté à ceux-ci est un défi à relever pour le développement futur des applications flot de données.

La principale limitation de ce chapitre est l'étendue réduite de l'évaluation du compilateur. En effet, cette évaluation est réalisée pour la plateforme Magali en simulation uniquement. Ces résultats demanderaient à être validés sur la plateforme réelle. Cette validation nécessiterait une présence physique au CEA pour accéder à la plateforme matérielle. Elle n'a pu aboutir par faute d'outils de déploiement et par manque de ressources. Par ailleurs, l'expérimentation sur d'autres plateformes de radio logicielle serait un autre point important pour démontrer la portabilité des applications. Ce chapitre a toutefois permis une démonstration des concepts développés durant cette thèse avec l'implémentation de parties du protocole LTE sur la plateforme de radio logicielle Magali.

7 Conclusion et perspectives

The isolated man does not develop any intellectual power. It is necessary for him to be immersed in an environment of other men, whose techniques he absorbs during the first twenty years of his life. He may then perhaps do a little research of his own and make a very few discoveries which are passed on to other men ... the search for new techniques must be regarded as carried out by the human community as a whole, rather than by individuals.

— Alan Turing, *Intelligent Machinery*

Le travail présenté dans cette thèse part du problème général de la programmation des plateformes parallèles, appliqué au domaine de la radio logicielle. La radio logicielle est présentée comme l'évolution de l'implémentation des protocoles radios, depuis des protocoles statiques implémentés sur des circuits dédiés, vers des protocoles dynamiques implémentés sur des circuits programmables. Ces nouveaux protocoles posent de fortes contraintes aux plateformes matérielles. Au nombre de ces contraintes on retrouve le débit de données à traiter qui augmente avec chaque nouveau protocole ; le temps de traitement de ces données pour respecter la qualité de service ; le dynamisme du protocole pour s'adapter à son environnement ; la consommation pour les systèmes embarqués. Toutes ces contraintes ont entraîné la création de plateformes complexes programmables et dédiées à la radio logicielle. Nous avons dans ce travail un bon exemple de cette complexité avec la plateforme Magali, qui propose une architecture hétérogène, une mémoire distribuée non cohérente, et une

grande part d'accélération matérielle. Cette complexité pose le problème du développement de programmes capables d'exploiter le potentiel de ces plateformes. Une solution possible à ce problème est l'utilisation d'un modèle de programmation basé sur le modèle de calcul flot de données. Ce modèle de calcul expose des informations de haut niveau sur l'application, comme le parallélisme de tâches ou le *pipeline*, et explicite les dépendances de données entre les traitements. Ce format représente l'application de manière indépendante de la plateforme, ce qui rend l'application portable. La difficulté est alors de faire le lien entre ce format d'application et la plateforme matérielle. C'est dans ce cadre que se pose ce travail de thèse, la spécialisation d'applications flot de données pour une plateforme de radio logicielle.

7.1 Synthèse des travaux

Je présente la synthèse des travaux réalisés durant cette thèse de manière chronologique, de manière à mieux comprendre, rétrospectivement, la réflexion qui a emmené aux contributions de cette thèse. Cette thèse s'est déroulée en 3 ans sur la base d'une collaboration entre le laboratoire CITI de l'INSA de Lyon et le laboratoire LISAN du CEA LETI de Grenoble. Elle s'est découpée sommairement en une année d'étude au laboratoire CITI, une année d'expérimentation au laboratoire LISAN, et une année d'analyse et de valorisation des résultats au laboratoire CITI.

7.1.1 Étude préliminaire

La première année de thèse a commencé par un état de l'art de la radio logicielle. Le passage en revue des nombreuses plateformes matérielles a révélé qu'il n'existait pas un modèle d'architecture dominant, mais de nombreuses solutions différentes. J'ai établi des catégories selon les caractéristiques de l'architecture qui influent sur la programmation de ces plateformes matérielles. Chacune des catégories remplit un objectif différent selon des critères de programmabilité, d'évolutivité ou de consommation d'énergie.

La suite de cette étude a porté sur les solutions logicielles existantes pour la programmation de ces plateformes. Cette étude a montré la présence importante d'applications développées selon des modèles proches des modèles de calcul flot de données. Ce lien s'explique par l'abstraction permise par ces modèles, qui est similaire au traitement du signal largement utilisé en radio logicielle. Cette abstraction est aussi une solution pour passer outre la complexité des plateformes matérielles utilisées. L'état de l'art a été réalisé en lien avec la mise en place de la plateforme d'expérimentation de radio cognitive CorteXlab [Cardoso 12]. Il a été valorisé par une publication dans la conférence Internationale IWCMC [Dardaillon 12b], un rapport de recherche Inria [Dardaillon 13a] et un chapitre de livre sur la radio cognitive à paraître [Dardaillon 14a].

Lors de cette première année nous avons choisi une application de référence pour fixer les contraintes matérielles et logicielles. Pour cela, j'ai étudié les protocoles de télécommunications à implémenter. Le choix du protocole de quatrième génération 3GPP-LTE s'est imposé comme référence. En effet, ce protocole est proposé comme le cadre pour le développement des protocoles dans les années à venir. Le LTE est un protocole complexe qui a demandé une étude approfondie des types de traitement à réaliser, des contraintes de débit, de temps d'exécution et du dynamisme à supporter dans le protocole. Parmi les points notables qui sont ressortis de cette étude, j'ai distingué l'importance des manipulations de données dans ce protocole. Près de la moitié des traitements sont en effet des manipulations de données à grain large, comme le découpage des données à traiter, ou à grain fin, comme le désentrelacement

des données. Les communications sont explicites dans le modèle de calcul flot de données, ce qui s'avère être un avantage supplémentaire pour prendre en charge ces manipulations de données.

Un point important à prendre en compte est le dynamisme du protocole LTE. Ce dynamisme doit s'exprimer dans le modèle de programmation. L'utilisation d'un modèle statique pour représenter une telle application amène au surdimensionnement de l'application, à la non prise en charge du dynamisme par les mécanismes intégrés à la plateforme, ce qui conduit à des pertes de performances, voire à l'impossibilité d'implémenter le protocole sur une plateforme donnée. J'ai fait le choix du modèle de calcul flot de données paramétrique SPDF pour supporter ce dynamisme. Ce choix s'appuie sur une étude des différents modèles de calcul flot de données, de leur expressivité mais aussi des analyses réalisables statiquement. Ces analyses rendent possible de nouvelles optimisations ainsi que l'ordonnancement statique du graphe, ou dans le cas de SPDF quasi-statique. À partir de ces contraintes, j'ai défini le format d'entrée utilisé et les outils de compilation. J'ai choisi de développer un nouveau flot de compilation, le modèle de calcul SPDF n'étant pas présent dans les outils actuels. Ce choix m'a donné l'opportunité d'expérimenter de nouvelles méthodes pour la compilation. J'utilise le langage C++ comme base existante, utilisée pour le développement sur système embarqué. L'infrastructure de compilation LLVM est choisie comme base de travail pour le développement du compilateur de par sa maturité pour constituer une base de travail solide ; de par son extensibilité qui en fait un outil largement utilisé en recherche ; et de par son utilisation dans les développements du CEA, pour bénéficier de l'expérience acquise et des travaux existants.

7.1.2 Expérimentations

Les études réalisées au laboratoire CITI ont permis de cerner quelques unes des difficultés à surmonter pour le développement de la radio logicielle. À partir de ces études, j'ai choisi de développer un nouveau format d'entrée basé sur le langage C++ et qui utilise le modèle de calcul SPDF, ainsi qu'un nouveau flot de compilation basé sur LLVM. La seconde année passée au laboratoire LISAN a été consacré à l'évaluation de ces choix.

J'ai réalisé des expérimentations préliminaires sur la plateforme Magali pour comprendre les subtilités de cette plateforme et le travail existant. J'ai pu constater la complexité du développement sur cette plateforme, car elle met en jeu un grand nombre de programmes sous la forme de configurations de calcul, de communications et de contrôle. À cette complexité s'ajoute la cohérence entre ces configurations qui est laissée au développeur dans le format natif. J'ai aussi pu constater des contraintes mémoires fortes, causes de nombreux interblocages lors de l'exécution de programmes. La résolution de ces interblocages passe par la modification de plusieurs configurations, la cohérence entre ces configurations rend ce débogage particulièrement complexe et consommateur en temps. Les outils existants pour Magali sont basés sur un modèle flot de données statique, contrairement à l'objectif fixé d'utiliser un modèle paramétrique. J'ai fait le choix d'un nouveau flot de compilation basé sur LLVM. Ce choix aurait du rendre possible l'intégration du compilateur Mephisto dans notre flot de compilation. Son état de développement ne l'a cependant pas permis.

Le format d'entrée est basé sur le langage C++, sans modification de sa syntaxe afin de réutiliser le *front end* C++ existant Clang. Le modèle de calcul flot de données est représenté par un ensemble de classes de bases utilisées pour la création des acteurs et des liens entre ces acteurs. Le *front end* du compilateur est étendu pour intégrer le graphe flot de données dans

la représentation intermédiaire. Cette extension utilise le concept développé dans le travail de Marquet et Moy [Marquet 10] sur l'analyse de modèles SystemC, et utilisé ici pour construire le graphe flot de données. Cette intégration passe par l'exécution du code de construction du graphe durant le processus de compilation. Cette exécution s'appuie sur les capacités de LLVM, et rend possible la construction de graphes complexes, avec l'utilisation de classes pour hiérarchiser sa structure, de boucles pour répéter des structures, et d'autres constructions rendues possibles par la prise en charge du langage C++. Une fois le graphe construit, il est nécessaire de lier l'objet graphe en mémoire à la représentation intermédiaire LLVM du traitement des acteurs. Ce lien est toujours basé sur les méthodes de Marquet et Moy, et utilise un algorithme de *slicing* pour isoler le code d'accès au graphe dans le code de calcul. Ce code d'accès au graphe est exécuté, et le résultat de cette exécution est l'arc accédé dans le graphe flot de données. Au final, ce *front end* utilise un langage C++ non trivial enrichi de classes pour représenter le graphe flot de données. Par la réutilisation d'outils existants et d'une technique novatrice, le développement de ce démonstrateur a été réalisé en 4 mois durant cette thèse.

Une fois la représentation intermédiaire du compilateur construite, la seconde étape fut le développement d'un *back end* pour la plateforme Magali. Il est constitué d'une première passe de *haut niveau* pour l'analyse et l'optimisation de la représentation intermédiaire, et d'une seconde passe de *bas niveau* pour la transformation de la représentation intermédiaire en assembleur pour Magali. La passe de haut niveau vérifie la bonne construction du graphe flot de données. Elle optimise la représentation intermédiaire à l'aide des analyses précédentes, en particulier la propagation des paramètres constants dans le graphe flot de données. Elle est aussi en charge de spécialiser l'application avec le placement, réalisé manuellement dans ce travail, et l'ordonnancement pour la plateforme ciblée. À partir de ce placement, la fusion d'acteurs spécialise les traitements en fonction de la granularité des accélérateurs.

La passe de bas niveau transforme la représentation intermédiaire vers l'assembleur pour la plateforme Magali. Cette passe se base sur la structure de génération de code le LLVM. Elle a nécessité le développement de générateurs de code pour les communications entre les unités, les contrôleurs distribués intégrés sur chacune des unités, ainsi que les blocs configurables. La prise en charge des paramètres variant dynamiquement dans ces opérations a nécessité une attention toute particulière. Le compilateur génère automatiquement la synchronisation par le contrôleur central, l'utilisation de registres pour les communications et le contrôle distribué, ainsi que la génération de configurations multiples pour les configurations de calcul. La génération de code utilise le compilateur ARM pour le contrôleur central, et le compilateur SME pour les DMA intelligents. L'intégration du compilateur Mephisto n'a pas été réalisée à cause de son état de développement lors des expérimentations. Le développement de l'ensemble du *back end* a représenté 4 mois de travail en collaboration avec un stagiaire ingénieur. Il a été valorisé par une publication sur la génération de code pour cible hétérogène dans un atelier lié à la conférence internationale HiPeaC [Dardaillon 14c].

La dernière étape de cette année d'expérimentation a été la validation des applications sur la plateforme Magali. Cette validation a montré que le code généré pour le calcul et les communications est équivalent au code assembleur développé par les ingénieurs du laboratoire LISAN. De plus, l'approche de haut niveau a réduit d'un facteur 40 le temps de développement par l'automatisation de la génération et de la cohérence des configurations. Le code de contrôle des unités de calcul est différent de l'approche développée au laboratoire LISAN, et a demandé une analyse plus complète. Les applications dynamiques développées au LISAN sont découpées en *phases*, chaque phase est un graphe flot de données statique.

Les transitions entre ces phases représentent le dynamisme de l'application, et sont prises en charge par le contrôleur central. Dans mon travail les applications sont exprimées dans un graphe flot de données paramétrique. La gestion des paramètres est intégrée au contrôleur central, ce qui offre l'opportunité d'un contrôle à la granularité de l'unité de calcul. Ces différents modèles de contrôle ont été étudiés sur le simulateur de la plateforme Magali. Les résultats montrent que la granularité proposée résulte en un surcoût trop important, surcoût qui augmente avec le nombre d'unités de calcul de l'application. Un troisième modèle de contrôle a été proposé qui utilise une reconfiguration unique pour toutes les unités de calcul lors du changement de valeur d'un paramètre. Les performances obtenues avec ce modèle sont équivalentes au contrôle par phase, et surpassent ce modèle dans le cas d'applications paramétriques. Ceci s'explique par la réduction du nombre d'unités à reconfigurer, seul les unités de la région d'influence du paramètre étant reconfigurées. L'analyse des différents modèles de contrôle et de leur performance a fait l'objet d'une publication dans la conférence française ComPAS [Dardaillon 14d].

7.1.3 Analyse et valorisation

La troisième année a été dédiée à l'analyse et à la valorisation des expérimentations réalisées au CEA, ainsi qu'à la rédaction de ce manuscrit de thèse, au laboratoire CITI.

Une difficulté rencontrée lors du développement sur la plateforme Magali est la faible quantité mémoire qui entraîne des interblocages. La difficulté se retrouve dans les applications compilées, et s'explique par l'inadéquation entre l'abstraction fournie par le modèle de calcul flot de données, dans lequel un acteur produit l'ensemble de ces données de manière atomique, et l'exécution réelle dans laquelle les données sont transmises de manière continue au travers d'une mémoire limitée. J'ai proposé de répondre à cette difficulté par un raffinement de l'ordonnancement de graphe flot de données, le micro-ordonnancement. Ce micro-ordonnancement représente l'ordre détaillé des communications de chaque acteur, et offre de multiples possibilités d'analyse. Cet ordre de consommation offre des informations pour guider l'ordonnancement. Le micro-ordonnancement des acteurs et leur placement sur la plateforme matérielle rend possible la vérification de l'absence d'interblocage dû aux communications. Cette analyse a été réalisée avec une modélisation PROMELA du micro-ordonnancement et de la plateforme, et une vérification par le model checker SPIN. L'analyse a permis de trouver les interblocages présents dans les applications, et de valider l'absence d'interblocage dans les applications de tests. Le temps de vérification augmente avec la complexité des applications, ce qui limite la taille des applications analysables. Cette analyse a rendu possible la vérification des applications de test, et représente un premier pas vers la vérification d'applications qui visent les plateformes fortement contraintes comme celle de cette étude.

Cette année de thèse a été largement consacrée à la valorisation des travaux d'expérimentations effectués au CEA, avec la rédaction de plusieurs articles sur le flot de compilation [Dardaillon 14c, Dardaillon 14d]. L'ensemble du travail sur le flot de compilation et l'analyse du micro-ordonnancement a donné lieu à une publication dans la conférence internationale CASES [Dardaillon 14b]. La rédaction de ce manuscrit durant ces 3 derniers mois est la dernière étape de ce travail de valorisation.

7.1.4 Travail réalisé

Pour évaluer le travail réalisé, je présente une estimation du temps dédié à chacune des grandes étapes de cette thèse :

— État de l'art de la radio logicielle : 8 mois
— Étude du protocole de télécommunication LTE : 2 mois
— Étude des modèles de calcul flot de données et de leur ordonnancement : 6 mois
— Étude des manipulations de données : 3 mois
— Développement du *front end* du compilateur : 4 mois
— Développement du *back end* pour Magali du compilateur : 4 mois
— Valorisation du travail par des publications : 6 mois
— Rédaction du manuscrit de thèse : 3 mois

Cette estimation inclut les périodes de réflexion qui ont mené à ces développements, dont le début de l'étude des manipulations de données présentée dans les perspectives de ce manuscrit. Je présente également une synthèse du travail réalisé sur le compilateur dans la Table 7.1.

Partie du compilateur	#lignes	mois × #développeurs	#applications évaluées
front end	6000	4 × 1	12
back end Magali	9000	4 × 2	6

TABLE 7.1 – Évaluation quantitative du travail réalisé sur le compilateur.

7.2 Discussion

Les travaux présentés dans ce manuscrit ont mis en évidence la pertinence d'un modèle de programmation basé sur les modèles de calcul flot de données pour la programmation de plateformes de radio logicielle. Ces travaux présentent plusieurs contributions, sur lesquelles je propose de porter un regard critique dans cette discussion.

7.2.1 Évolution de la radio logicielle

En premier lieu, ce travail présente un large état de l'art des plateformes matérielles et logicielles pour la radio logicielle. Cet état de l'art n'a pas distingué de tendance claire, car plusieurs catégories de plateformes sont développées pour des besoins différents. Il est toutefois intéressant d'observer la commercialisation de plusieurs de ces produits, comme la plateforme NXP EVP 16 [Moreira 12] ou le X-GOLD SDR 20 par Infineon [Ramacher 11]. On observe aussi la maturation de la radio logicielle avec le rachat de *start-ups* du domaine, soit pour leur intégration dans des produits commerciaux dans le cas du rachat de Icera par Nvidia [Nvidia 13], ou pour leur technologie dans le cas du rachat de Sandbridge [Glossner 07] par Qualcomm. Une conséquence possible de cette rationalisation est la réduction des développements et de la recherche publiée par l'industrie.

D'autres pans de la radio logicielle bénéficient encore d'une recherche visible, avec des développements pour l'expérimentation rapide de nouveaux protocoles dans des conditions réalistes. Par exemple, l'arrivée des outils de radio logicielle *open-source* comme GNU Radio offrent une solution simple à programmer qui correspond à ces besoins d'expérimentation. GNU Radio reste cependant limité pour l'expérimentation sur des protocoles plus complexes avec des contraintes temporelles fortes. Par exemple, l'expérimentation sur un protocole de

partage de réseau coopératif [Simeone 08] — l'utilisateur secondaire relaie le trafic de l'utilisateur primaire en échange de temps de transmission dédié — demande une reconfiguration rapide pour la retransmission du message. Pour rendre possible de telles expérimentations, il est nécessaire d'associer la facilité de programmation d'approches telles que GNU Radio à des plateformes capables de traitements complexes et de reconfigurations rapides. Cette thèse a montré que cette association est rendue possible par l'utilisation d'un modèle de programmation flot de données.

7.2.2 Flot de compilation

Ce travail a fait la démonstration d'une nouvelle méthode de développement pour l'implémentation d'un *front end* flot de données qui possède une large expressivité. Il serait intéressant d'explorer les possibilités offertes par ce *front end* au delà de l'implémentation de radio logicielle, comme un moyen d'apporter une information d'architecture lors du processus de compilation. Un autre point important serait l'étude des limites de ce *front end*, en particulier de l'étape de liaison de la représentation intermédiaire avec le graphe. Enfin, le développement actuel d'un générateur de code C depuis la représentation LLVM — le *back-end C* intégré à l'infrastructure de compilation LLVM — pourrait ouvrir de nouvelles possibilités de compilation. Notamment, il rendrait possible la compilation du code d'unités de calcul par d'autres compilateurs, avec l'utilisation de code C comme langage intermédiaire.

Ce travail a aussi démontré la possibilité d'automatiser la génération de code pour la plateforme Magali avec la prise en charge de paramètres variables dynamiquement, absent des travaux existants. Il a montré l'intérêt d'un modèle de calcul flot de données pour offrir de larges possibilités d'analyse et d'optimisation. La génération de code pour une plateforme complexe telle que Magali reste cependant coûteux en temps de développement. De plus, la réflexion et le développement de cette génération de code ont été réalisés bien après le développement de la plateforme matérielle. L'évaluation de cette plateforme a été réalisée en assembleur lors de son développement, ce qui a limité la complexité des applications d'évaluation. Elle a empêché l'évaluation de mécanismes matériels complexes tels que le mécanisme de changement de contexte intégré aux contrôleurs distribués. Cet exemple est un rappel du besoin d'intégrer le développement du logiciel avec le développement matériel pour arriver à en exploiter le potentiel. Cette adaptation devra passer par le développement de logiciels adaptés au matériel, mais aussi de matériels exploitables par le logiciel. Ce second point est capital pour ne pas aboutir à des processeurs de traitements du signal programmés uniquement en assembleur et inexploités de par leur complexité.

7.2.3 Support d'exécution

Une des difficultés rencontrées dans les expérimentations sur Magali est le niveau de performances des applications, en fonction du modèle de contrôle. Le modèle de contrôle le plus performant sur cette plateforme, selon les mesures, est le modèle qui met le moins en jeu le contrôleur central. Ceci est dû au temps de reconfiguration du contrôleur, pris en charge par un système d'exploitation non adapté au contrôle d'application flot de données. En effet, une des propriétés importantes de ce modèle de calcul est la connaissance des dépendances entre toutes les tâches, sous la forme de données. Ces dépendances sont une information importante pour l'ordonnancement efficace des acteurs ; elles ne sont cependant pas prises en compte par l'ordonnancement du système d'exploitation utilisé. Le développement d'un système d'exploitation adapté au flot de données est un travail important pour la démocrati-

sation de ce type d'application. Ce travail devra passer par une réflexion sur le format de ces dépendances, et sur leur transformation vers une forme exploitable par l'ordonnancement de ce système d'exploitation. Un exemple de cette réflexion est fourni par le travail de Pop et Cohen [Pop 13], dans lequel l'ordonnancement des acteurs est dirigé par les données. Lors de la fin de l'exécution d'un acteur, il décrémente le compteur associé aux acteurs dépendants. Si ce compteur arrive à 0, alors l'acteur consommateur est ordonnancé.

7.3 Perspectives

Les travaux réalisés durant ces trois années ont permis d'explorer la compilation d'application flot de données pour un domaine spécifique, la radio logicielle. Au delà des contributions présentées, ce travail ouvre de larges perspectives, avec d'une part la prolongation des travaux actuels, et d'autre part l'ouverture d'une réflexion plus large sur *les manipulations des données*.

7.3.1 Développement du compilateur

Le développement d'un compilateur est un travail d'envergure que l'on peut souhaiter voir prolongé dans le futur. Il dépend du besoin d'autres développeurs pour ses capacités, comme la programmation de la plateforme Magali, et la présence d'une communauté autour de ce travail pour le maintenir. Il est toutefois intéressant de discuter des possibilités de développements envisageables comme un retour d'expérience. Ce développement a pour premier objectif de renforcer le travail réalisé, en combler les lacunes et étendre ses capacités. Comme discuté précédemment, le support d'exécution actuel limite les performances de la plateforme, et pourrait être amélioré par le développement d'un ordonnancement dédié. Le développement de la génération de code pour de nouvelles plateformes est un autre travail important pour démontrer la portabilité de l'approche développée. La prise en charge de nouveaux modèles de calcul permettrait d'étendre l'expressivité du format développé, avec par exemple l'intégration du modèle de calcul BPDF [Bebelis 13a]. BPDF reprend la base du modèle de calcul flot de données paramétrique de SPDF [Fradet 12], et limite le changement de valeur des paramètres à une nouvelle itération. Il ajoute la gestion des paramètres booléens pour désactiver des parties du graphe, et dont les valeurs peuvent changer plusieurs fois durant une itération. Un intérêt par rapport au modèle paramétrique utilisé actuellement est d'expliciter la désactivation d'éléments du graphe ; cette information est utilisable pour le placement d'acteurs qui ne peuvent être exécutés de manière concurrente, ou pour reconfigurer la topologie du graphe flot de données.

Un compilateur flot de données intéressant de ce point de vue est ORCC [Gorin 10]. En effet, il prend en charge plusieurs modèles de calcul flot de données plus ou moins expressifs. L'analyse de l'application lors de la compilation réduit l'expressivité au modèle de calcul le plus restreint [Gu 10], ce qui augmente les analyses possibles. Il serait intéressant d'étudier l'intégration de la gestion des paramètres sous la forme d'un modèle de calcul de type SPDF ou BPDF dans ce compilateur. En effet, ORCC ne prend pas en charge les paramètres variables dynamiquement à l'heure actuelle. Les travaux récents sur le modèle de calcul flot de données paramétrique PiMM [Desnos 13] sont une autre possibilité pour la gestion de ces paramètres. Un point pour lequel ce compilateur n'a pas été retenu dans ce travail est l'absence de génération de code pour une plateforme dédiée comme Magali. La mise en place de la génération de code pour une plateforme dédiée à la radio logicielle sur ORCC est un point à évaluer dans le futur.

7.3.2 Pérennité des approches dédiées à la radio logicielle

Les contributions de cette thèse se sont concentrées sur la compilation pour des plateformes de radio logicielle dédiées, avec comme démonstrateur la plateforme Magali. La fin du développement de cette plateforme pose la question de la pérennité des contributions pour ce type de plateformes dédiées. Le développement de la radio logicielle se doit de rester en contact avec les besoins des futurs protocoles. La bande passante de ces protocoles augmente à chaque génération pour satisfaire aux besoins de débits de données. De plus, la numérisation du signal est réalisée plus proche de l'antenne à chaque génération, ce qui augmente la partie programmable, et la rapproche de la radio logicielle idéale telle que définie par le *Wireless Innovation Forum*. Ces développements entraînent une augmentation du débit de données à traiter, débit qui augmente plus vite que la puissance de calcul [Ulversoy 10].

Une autre tendance est l'intégration de ces modules de télécommunications dans des équipements autonomes à basse consommation. Elle a vu depuis une dizaine d'années la démocratisation des téléphones intelligents. On observe actuellement la réalisation de *l'internet des objets*, avec comme exemple la commercialisation des premières montres intelligentes qui intègrent un module 3G. Ces contraintes de consommation d'énergie s'appliquent aussi sur les équipements non autonomes, avec d'une part des besoins d'économie d'énergie, et d'autre part une limite à la dissipation d'énergie d'une plateforme matérielle. Tous ces développements montrent le besoin de traiter un débit de données en augmentation avec une consommation maîtrisée, besoin auquel seul peut répondre un matériel dédié.

7.3.3 Manipulations de données

Une des observations faites sur les protocoles actuels est l'importance des manipulations de données, qui représentent près de la moitié des traitements. Les modèles de calcul flot de données exposent ces manipulations de données entre les acteurs. La plupart de ces modèles comprennent les opérateurs de base de jointure ou de séparation de données. Ces opérateurs de base peuvent être combinés pour représenter des manipulations plus complexes [de Oliveira Castro 10]. Les manipulations exposées par les modèles de calcul flot de données restent cependant limitées aux échanges entre les acteurs. Les manipulations de données intra-acteurs, importantes dans ces protocoles, sont absentes de la représentation flot de données. Le micro-ordonnancement expose plus d'informations sur les manipulations de ces données aux frontières entre les acteurs. C'est un premier pas qui rend possible de nouvelles analyses et optimisations sur la mémoire ou l'ordonnancement. Il révèle aussi un réel besoin d'une représentation analysable de ces manipulations pour être exploitée lors de la compilation.

Il existe plusieurs langages adaptés à la représentation des manipulations de données à grain fin au sein d'un acteur. Array-OL [Boulet 07] est un langage de modélisation de haut niveau qui représente l'accès aux données sur des structures multidimensionnelles. Il repose sur des hypothèses similaires aux modèles de calcul flot de données, c'est à dire qu'il est dirigé par les données, sans état interne aux traitements, et sans contrôle complexe. Son expression régulière limite cependant son expressivité. Il est de plus complexe, ce qui le rend difficilement utilisable. Le langage de spécification CRP (*Communicating Regular Processes*) [Feautrier 06] est basé sur le langage C. Il spécifie les traitements sous la forme de modules, reliés par des canaux. Ces canaux sont des tableaux multidimensionnels à assignation unique. Le calcul des indices est limité à des expressions affines pour le besoin des analyses, et les structures des tableaux de données statiques. Les solutions présentées font toutes l'hypothèse d'un flot de

données statique. Il serait nécessaire d'étendre ces modèles pour correspondre au besoin de dynamisme des applications étudiées.

L'analyse de ces manipulations de données serait la source d'un grand nombre d'informations. Il serait possible d'en extraire l'ordre des consommations et des productions de données, modélisé par le micro-ordonnancement dans ce travail. Ces informations sont à la base du contrôle des tailles mémoires présenté dans ce manuscrit. Elles seraient aussi utile pour guider l'ordonnancement des acteurs. L'analyse de ces manipulations rendrait aussi possible leur transformation vers une forme adaptée à la plateforme, tout comme celle des manipulations de données entre acteurs [Llopard 11a].

La représentation des manipulations de données de manière abstraite constituerait une base idéale pour la génération de code vers tout type de plateforme de manipulation de données. Par exemple sur la plateforme Magali, cette génération de code viserait les modules de communication ICC/OCC entre les unités de calcul pour les opérations simples comme la fusion de données, et les mémoires intelligentes SME pour des opérations plus complexes comme le désentrelacement. Les manipulations de données constituent un type de traitement à part, qui met souvent en jeu plusieurs unités. La proposition récente de dédier une étape de placement séparée pour les transferts de données [Enrici 14] en souligne l'intérêt. L'analyse et l'optimisation de ces manipulations de données tout au long du flot de compilation est une des clés pour arriver à passer au delà du mur de la mémoire.

Publications

Chapitre de livre

[Dardaillon 14a] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Cognitive Radio Programming Survey*. In Naima Kaabouch & Wen-Chen Hu, editeurs, Handbook of Research on Software-Defined and Cognitive Radio Technologies for Dynamic Spectrum Management. IGI Global, October 2014.

Conférences internationales

[Dardaillon 12b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset & Antoine Scherrer. *Software defined radio architecture survey for cognitive testbeds*. In International Wireless Communications and Mobile Computing Conference (IWCMC), pages 189–194, Limassol, Cyprus, August 2012.

[Dardaillon 12a] Mickaël Dardaillon, Cédric Lauradoux & Tanguy Risset. *Hardware implementation of the GPS authentication*. In International Conference on Reconfigurable Computing and FPGAs (ReConFig), pages 1–6, Cancun, Mexico, December 2012.

[Dardaillon 14b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-pierre Charles. *A Compilation Flow for Parametric Dataflow : Programming Model, Scheduling, and Application to Heterogeneous MPSoC*. In International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), New Delhi, India, October 2014.

Rapport de recherche

[Dardaillon 13a] Mickaël Dardaillon, Kevin Marquet, Jérôme Martin, Tanguy Risset & Henri-Pierre Charles. *Cognitive Radio Programming : Existing Solutions and Open Issues*. Rapport technique 8358, Inria, September 2013.

Conférences nationales et ateliers

[Dardaillon 11] Mickaël Dardaillon, Cédric Lauradoux & Tanguy Risset. *QC-SYND*. In Journées scientifiques SEmba, Valence, France, October 2011.

[Dardaillon 13b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Compilation d'application data flow paramétrique visant les systèmes sur puces dédiés*. In Rencontres de la communauté française de compilation, Dammarie-Les-Lys, France, December 2013.

[Dardaillon 13c] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Front-End pour compilateur dataflow utilisant l'infrastructure llvm*. In Journées scientifiques SEmba, Lyon, France, April 2013.

[Dardaillon 13d] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Machine Virtuelle pour la Radio Cognitive*. In École d'hiver Francophone sur les Technologies de Conception des Systèmes embarqués Hétérogènes (FETCH), Leysin, Switzerland, January 2013.

[Dardaillon 14c] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Compilation for heterogeneous SoCs : bridging the gap between software and target-specific mechanisms*. In Workshop on High Performance Energy Efficient Embedded Systems at the international conference on High Performance and Embedded Architecture and Compilation (HiPEAC), Vienna, Austria, January 2014.

[Dardaillon 14d] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-pierre Charles. *Contrôle d'application flot de données pour les systèmes sur puces : étude de cas sur la plateforme Magali*. In Conférence en Parallélisme, Architecture et Système (ComPAS), Neuchâtel, Switzerland, April 2014.

[Dardaillon 14e] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Un nouveau flot de compilation pour application flot de données paramétrique*. In Colloque du GDR SoC-SiP, Paris, France, June 2014.

Bibliographie

[Agarwala 07] Sanjive Agarwala, Arjun Rajagopal, Anthony Hill, Mayur Joshi, Steven Mullinnix, Timothy Anderson, Raguram Damodaran, Lewis Nardini, Paul Wiley, Peter Groves & Others. *A 65nm C64x+ multi-core DSP platform for communications infrastructure*. In IEEE International Solid-State Circuits Conference (ISSCC), pages 262–601, San Francisco, California, February 2007. (p. 11, 15)

[Agilent 09] Agilent. *Agilent 3GPP Long Term Evolution : System Overview, Product Development, and Test Challenges*. Application note, Agilent, 2009. (p. 79)

[Amdahl 67] Gene M Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In AFIPS Joint Computer Conference, pages 483–485, Atlantic City, New Jersey, April 1967. (p. 3)

[Arnold 14] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg & Gerhard Fettweis. *Tomahawk : Parallelism and heterogeneity in communications signal processing MPSoCs*. ACM Transactions on Embedded Computing Systems (TECS), vol. 13, no. 3s, page 107, mar 2014. (p. 4, 12)

[Asanovic 06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams*et al.* *The landscape of parallel computing research : A view from Berkeley*. Rapport technique UCB/EECS-2006-183, University of California, Berkeley, December 2006. (p. 2)

[Auerbach 10] Joshua Auerbach, David F. Bacon, Perry Cheng & Rodric Rabbah. *Lime : a Java-Compatible and Synthesizable Language for Heterogeneous Architectures*. In ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pages 89 – 108, Reno, Nevada, October 2010. (p. 19)

[Bebelis 13a] Vagelis Bebelis, Pascal Fradet, Alain Girault & Bruno Lavigueur. *BPDF : A statically analyzable dataflow model with integer and boolean parameters*. In Proceedings of the International Conference on Embedded Software (EMSOFT), pages 1–10, Montreal, Canada, September 2013. (p. 27, 32, 98)

[Bebelis 13b] Vagelis Bebelis, Pascal Fradet, Alain Girault & Bruno Lavigueur. *A Framework to Schedule Parametric Dataflow Applications on Many-*

Core Platforms. In Workshop on Compilers for Parallel Computing (CPC), Lyon, France, July 2013. (p. 27)

[Ben Abdallah 10] Riadh Ben Abdallah, Tanguy Risset, Antoine Fraboulet & Jérôme Martin. *Virtual Machine for Software Defined Radio : Evaluating the Software VM Approach*. In International Conference on Computer and Information Technology (CIT), pages 1970–1977, Bradford, United Kingdom, June 2010. (p. 4, 18, 62, 88)

[Berkel 05] Kees Van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman & Matthias Weiss. *Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices*. EURASIP Journal on Advances in Signal Processing, vol. 2005, no. 16, pages 2613–2625, January 2005. (p. 11)

[Bernard 11] Christian Bernard & Fabien Clermidy. *A low-power VLIW processor for 3GPP-LTE complex numbers processing*. In International conference on Design, Automation & Test in Europe (DATE), pages 1–6, Grenoble, France, March 2011. (p. 60)

[Bhattacharya 01] Bishnupriya Bhattacharya & Shuvra S. Bhattacharyya. *Parameterized dataflow modeling for DSP systems*. IEEE Transactions on Signal Processing, vol. 49, no. 10, pages 2408–2421, October 2001. (p. 25, 31)

[Bhattacharyya 96] Shuvra S. Bhattacharyya, Praveen K. Murthy & Edward Ashford Lee. Software synthesis from dataflow graphs. Kluwer Academic Press, 1996. (p. 24, 29)

[Bhattacharyya 99] Shuvra S. Bhattacharyya, Praveen K. Murthy & Edward Ashford Lee. *Synthesis of embedded software from synchronous dataflow specifications*. Journal of VLSI signal processing systems for signal, image and video technology, vol. 166, no. 2, pages 151–166, June 1999. (p. 23, 28, 29, 30, 67)

[Bhattacharyya 08] Shuvra S. Bhattacharyya, Gordon Brebner, Johan Eker, Carl von Platen, Marco Mattavelli & Mickaël Raulet. *OpenDF : a dataflow toolset for reconfigurable hardware and multicore systems*. ACM SIGARCH Computer Architecture News, vol. 36, no. 5, pages 29–35, December 2008. (p. 19, 45)

[Bilsen 96] Greet Bilsen, Marc Engels, Rudy Lauwereins & Jean Peperstraete. *Cyclestatic dataflow*. IEEE Transactions on Signal Processing, vol. 44, no. 2, pages 397–408, February 1996. (p. 24)

[Bodin 14] Bruno Bodin, Youen Lesparre, Jean-Marc Delosme & Alix Munier-Kordon. *Fast and efficient dataflow graph generation*. In International Workshop on Software and Compilers for Embedded Systems (SCOPES), pages 40–49, St. Goar, Germany, jun 2014. (p. 84)

[Bouakaz 13] Adnan Bouakaz & Jean-Pierre Talpin. *Buffer minimization in earliest-deadline first scheduling of dataflow graphs*. In ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), pages 133–142, Edinburgh, Scotland, June 2013. (p. 29, 30)

[Bougard 08] Bruno Bougard, Bjorn De Sutter, Diederik Verkest, Liesbet Van der Perre & Rudy Lauwereins. *A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing*. IEEE Micro, vol. 28, no. 4, pages 41–50, July 2008. (p. 11)

[Boulet 07] Pierre Boulet. *Array-OL revisited, multidimensional intensive signal processing specification*. Rapport technique, Inria, February 2007. (p. 99)

[Buck 93] Joseph T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California at Berkeley, Berkeley, California, 1993. (p. 25)

[Carbon 13] Alexandre Carbon. *Accélération matérielle de la compilation à la volée pour les systèmes embarqués*. PhD thesis, Université Pierre et Marie Curie, Paris, France, October 2013. (p. 75)

[Cardoso 10] João M. P. Cardoso, Pedro C. Diniz & Markus Weinhardt. *Compiling for reconfigurable computing*. ACM Computing Surveys, vol. 42, no. 4, pages 1–65, June 2010. (p. 65)

[Cardoso 12] Leonardo S. Cardoso, Guillaume Villemaud, Tanguy Risset & Jean-Marie Gorce. *CorteXlab : A Large Scale Testbed for Physical Layer in Cognitive Radio Networks*. In European cooperation in the field of scientific and technical research, Lyon, France, May 2012. (p. 92)

[Castrillon 11] Jeronimo Castrillon, Stefan Schürmans, Anastasia Stulova, Weihua Sheng, Torsten Kempf, Rainer Leupers, Gerd Ascheid & Heinrich Meyr. *Component-based waveform development : the Nucleus tool flow for efficient and portable software defined radio*. Analog Integrated Circuits and Signal Processing, vol. 69, no. 2-3, pages 173–190, June 2011. (p. 19, 48, 65)

[Castrillon 13] Jeronimo Castrillon, Rainer Leupers & Gerd Ascheid. *MAPS : Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs*. IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pages 527–545, February 2013. (p. 4, 19, 30)

[Clermidy 09a] Fabien Clermidy, Romain Lemaire, Xavier Popon, Dimitri Ktenas & Yvain Thonnart. *An Open and Reconfigurable Platform for 4G Telecommunication : Concepts and Application*. In 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pages 449–456, Patras, Greece, August 2009. (p. 13, 15, 16, 58, 59, 85)

[Clermidy 09b] Fabien Clermidy, Romain Lemaire & Yvain Thonnart. *A Communication and configuration controller for NoC based reconfigurable data flow architecture*. In 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS), pages 153–162, San Diego, California, May 2009. (p. 61)

[Clermidy 10] Fabien Clermidy, Christian Bernard, Romain Lemaire, Jérôme Martin, Ivan Miro-Panades, Yvain Thonnart, Pascal Vivet & Norbert Wehn. *A 477mW NoC-based digital baseband for MIMO 4G SDR*. In IEEE International Solid-State Circuits Conference (ISSCC), pages 278–279, San Francisco, California, February 2010. (p. 13)

[Cohen 10] Albert Cohen & Erven Rohou. *Processor virtualization and split compilation for heterogeneous multicore embedded systems*. In Design Automation Conference (DAC), pages 102 – 107, Anaheim, California, June 2010. (p. 18, 43)

[Dardaillon 11] Mickaël Dardaillon, Cédric Lauradoux & Tanguy Risset. *QC-SYND*. In Journées scientifiques SEmba, Valence, France, October 2011. (p. 101)

[Dardaillon 12a] Mickaël Dardaillon, Cédric Lauradoux & Tanguy Risset. *Hardware implementation of the GPS authentication*. In International Conference on Reconfigurable Computing and FPGAs (ReConFig), pages 1–6, Cancun, Mexico, December 2012. (p. 101)

[Dardaillon 12b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset & Antoine Scherrer. *Software defined radio architecture survey for cognitive testbeds*. In International Wireless Communications and Mobile Computing Conference (IWCMC), pages 189–194, Limassol, Cyprus, August 2012. (p. 92, 101)

[Dardaillon 13a] Mickaël Dardaillon, Kevin Marquet, Jérôme Martin, Tanguy Risset & Henri-Pierre Charles. *Cognitive Radio Programming : Existing Solutions and Open Issues*. Rapport technique 8358, Inria, September 2013. (p. 92, 101)

[Dardaillon 13b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Compilation d'application data flow paramétrique visant les systèmes sur puces dédiés*. In Rencontres de la communauté française de compilation, Dammarie-Les-Lys, France, December 2013. (p. 101)

[Dardaillon 13c] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Front-End pour compilateur dataflow utilisant l'infrastructure llvm*. In Journées scientifiques SEmba, Lyon, France, April 2013. (p. 101)

[Dardaillon 13d] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Machine Virtuelle pour la Radio Cognitive*. In École d'hiver Francophone sur les Technologies de Conception des Systèmes embarqués Hétérogènes (FETCH), Leysin, Switzerland, January 2013. (p. 102)

[Dardaillon 14a] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Cognitive Radio Programming Survey*. In Naima Kaabouch & Wen-Chen Hu, editeurs, Handbook of Research on Software-Defined and Cognitive Radio Technologies for Dynamic Spectrum Management. IGI Global, October 2014. (p. 92, 101)

[Dardaillon 14b] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *A Compilation Flow for Parametric Dataflow : Programming Model, Scheduling, and Application to Heterogeneous MPSoC*. In International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), New Delhi, India, October 2014. (p. 30, 95, 101)

[Dardaillon 14c] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Compilation for heterogeneous SoCs : bridging the gap between software and target-specific mechanisms*. In Workshop on High Performance Energy Efficient Embedded Systems at the international conference on High Performance and Embedded Architecture and Compilation (HiPEAC), Vienna, Austria, January 2014. (p. 66, 94, 95, 102)

[Dardaillon 14d] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Contrôle d'application flot de données pour les systèmes sur puces : étude de cas sur la plateforme Magali*. In Conférence en Parallélisme, Architecture et Système (ComPAS), Neuchâtel, Switzerland, April 2014. (p. 95, 102)

[Dardaillon 14e] Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin & Henri-Pierre Charles. *Un nouveau flot de compilation pour application flot de données paramétrique*. In Colloque du GDR SoC-SiP, Paris, France, June 2014. (p. 102)

[de Oliveira Castro 10] Pablo de Oliveira Castro, Stéphane Louise & Denis Barthou. *A multidimensional array slicing DSL for Stream Programming*. In International Conference on Complex, Intelligent and Software Intensive Systems, pages 913 – 918, Krakow, Poland, February 2010. (p. 99)

[Derudder 09] V. Derudder, Bruno Bougard, A. Couvreur, A. Dewilde, S. Dupont, L. Folens, L. Hollevoet, F. Naessens, D. Novo, P. Raghavan, T. Schuster, K. Stinkens, J.-W. Weijers & Liesbet Van der Perre. *A 200Mbps+ 2.14 nJ/b digital baseband multi processor system-on-chip for SDRs*. In Symposium on VLSI Circuits, pages 292–293, Kyoto, Japan, June 2009. (p. 13, 15, 16, 17)

[Desnos 13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya & Slaheddine Aridhi. *PiMM : Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration*. In International Conference on Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS), pages 41–48, Samos, Greece, July 2013. (p. 98)

[Dutoit 13] Denis Dutoit, Christian Bernard, Severine Cheramy, Fabien Clermidy, Yvain Thonnart, Pascal Vivet, Christian Freund, Vincent Guerin, Stephane Guilhot, Stephane Lecomte, Gianni Qualizza, Julien Pruvost, Yves Dodo, Nicolas Hotelier & Jean Michailos. *A 0.9 pJ/bit, 12.8 GByte/s WideIO memory interface in a 3D-IC NoC-based MPSoC*. In Symposium on VLSI Technology (VLSIT), pages C22–C23, Kyoto, Japan, June 2013. (p. 83)

[eCos 14] eCos. *embedded Configurable operating system*. `http://ecos.sourceware.org`, 2014. (p. 74)

[Enrici 14] Andrea Enrici, Ludovic Apvrille & Renaud Pacalet. *Communication Patterns : a Novel Modeling Approach for Software Defined Radio Systems*. In International Conference on Advances in Cognitive Radio (COCORA), pages 35–40, Nice, Fr, feb 2014. (p. 100)

[Esmaeilzadeh 11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam & Doug Burger. *Dark silicon and the end of multicore scaling*. In International Symposium on Computer Architecture (ISCA), pages 365–376, San Jose, California, June 2011. (p. 2)

[Feautrier 06] Paul Feautrier. *Scalable and Structured Scheduling*. International Journal of Parallel Programming, vol. 34, no. 5, pages 459–487, May 2006. (p. 30, 99)

[Fradet 12] Pascal Fradet, Alain Girault & Peter Poplavko. *SPDF : A schedulable parametric data-flow MoC*. In International conference on Design, Automation & Test in Europe (DATE), pages 769–774, Dresden, Germany, March 2012. (p. 25, 30, 98)

[Geilen 05] Marc Geilen, Twan Basten & Sander Stuijk. *Minimising buffer requirements of synchronous dataflow graphs with model checking*. In Design Automation Conference (DAC), page 819, San Diego, California, June 2005. (p. 29, 37)

[Glossner 07] John Glossner, Daniel Iancu, Mayan Moudgill, Gary Nacer, Sanjay Jinturkar, Stuart Stanley & Michael J. Schulte. *The Sandbridge SB3011 Platform*. EURASIP Journal on Embedded Systems, vol. 2007, no. 1, pages 1–16, April 2007. (p. 9, 12, 96)

[Gonzalez-Pina 12] Jair Gonzalez-Pina, Rabea Ameur-Boulifa & Renaud Pacalet. *DiplodocusDF, a Domain-Specific Modelling Language for Software Defined Radio Applications*. In Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pages 1–8, Cesme, Izmir, September 2012. (p. 19, 65)

[Gonzalez 09] C.R.A. Gonzalez, C.B. Dietrich, Shereef Sayed, H.I. Volos, J.D. Gaeddert, P.M. Robert, J.H. Reed & F.E. Kragh. *Open-source SCA-based core framework and rapid development tools enable software-defined radio education and research*. IEEE Communications Magazine, vol. 47, no. 10, pages 48–55, October 2009. (p. 17)

[Gorin 10] J Gorin, M Wipliez, F Preteux & Mickaël Raulet. *A portable Video Tool Library for MPEG Reconfigurable Video Coding using LLVM representation*. In Conference on Design and Architectures for Signal and Image Processing (DASIP), pages 183–190, Edinburgh, Scotland, October 2010. (p. 45, 98)

[Goubier 11] Thierry Goubier, Renaud Sirdey, Stéphane Louise & Vincent David. *ΣC A Programming Model and Language for Embedded Manycores*. In International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), pages 385–394, Melbourne, Australia, October 2011. (p. 18, 45)

[Graphviz 14] Graphviz. `http://http://graphviz.org`, 2014. (p. 64)

[Gu 10] Ruirui Gu, Jörn W. Janneck, Mickaël Raulet & Shuvra S. Bhattacharyya. *Exploiting Statically Schedulable Regions in Dataflow Programs*. Journal of Signal Processing Systems, vol. 63, no. 1, pages 129–142, January 2010. (p. 19, 30, 98)

[Gummaraju 08] Jayanth Gummaraju, Joel Coburn, Yoshio Turner & Mendel Rosenblum. *Streamware : Programming General-Purpose Multicore Processors Using Streams*. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), page 297, Seattle, Washington, March 2008. (p. 18)

[Hines 05] Stephen Hines, Joshua Green, Gary Tyson & David Whalley. *Improving program efficiency by packing instructions into registers*. In International Symposium on Computer Architecture (ISCA), volume 33, pages 260–271, Madisson, Wisconsin, June 2005. (p. 71)

[Horrein 11] Pierre-Henri Horrein, Christine Hennebert & Frédéric Pétrot. *Integration of GPU Computing in a Software Radio Environment*. Journal of Signal Processing Systems, vol. 69, no. 1, pages 55–65, December 2011. (p. 11, 18)

[Hsu 10] Chia-Jui Hsu, José Luis Pino & Fei-Jiang Hu. *A mixed-mode vector-based dataflow approach for modeling and simulating LTE physical layer*. In Design Automation Conference (DAC), pages 18–23, Anaheim, California, June 2010. (p. 19)

[Jaaskelainen 10] Pekka O. Jaaskelainen, Carlos S. de La Lama, Pablo Huerta & Jarmo H. Takala. *OpenCL-based design methodology for application-specific processors*. In Conference on Embedded Computer Systems : Architectures, Modeling and Simulation (SAMOS), pages 223–230, Samos, Greece, July 2010. (p. 17)

[Jalier 10] Camille Jalier, Didier Lattard, AA Jerraya, Gilles Sassatelli, Pascal Benoit & Lionel Torres. *Heterogeneous vs homogeneous MPSoC approaches for a mobile LTE modem*. In International conference on Design, Automation & Test in Europe (DATE), pages 184–189, Dresden, Germany, March 2010. (p. 13, 15)

[Johnston 04] Wesley M. Johnston, J. R. Paul Hanna & Richard J. Millar. *Advances in dataflow programming languages*. ACM Computing Surveys, vol. 36, no. 1, pages 1–34, March 2004. (p. 22)

[Kahn 74] Gilles Kahn. *The semantics of a simple language for parallel programming*. In Information Processing : Proceedings of the IFIP Congress, pages 471 – 475, Stockholm, Sweden, August 1974. (p. 22)

[Labonte 04] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis & Mark Horowitz. *The stream virtual machine*. In 13th International Conference on Parallel Architecture and Compilation Techniques (PACT), pages 267–277, Standford, California, September 2004. (p. 18)

[Lin 06] Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid & Krisztian Flautner. *SPEX : A programming language for software defined radio*. In SDR Forum Technical Conference, pages 13 – 17, Orlando, Florida, November 2006. (p. 19)

[Linderman 08] Michael D. Linderman, Jamison D. Collins, Hong Wang & Teresa H. Meng. *Merge : A Programming Model for Heterogeneous Multi-core*

Systems. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), page 287, Seattle, Washington, March 2008. (p. 18)

[Liu 09] Weichen Liu, Zonghua Gu, Jiang Xu, Yu Wang & Mingxuan Yuan. *An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking*. In Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS), pages 61–70, Grenoble, France, October 2009. (p. 29)

[Llopard 11a] Ivan Llopard. Etude et développement d'outils de mapping d'applications sur des architectures réseau sur puce (NoC). Master's thesis, Institut National Polytechnique de Grenoble, Grenoble, France, June 2011. (p. 100)

[Llopard 11b] Ivan Llopard, Jérôme Martin & Frédéric Rousseau. *comC : a NoC Communication Compiler*. In Workshop on Hardware Dependent Software Solutions for SoC Design at the international conference on Design, Automation & Test in Europe (DATE), Grenoble, France, March 2011. (p. 62)

[Llopard 13] Ivan Llopard, Albert Cohen, Christian Fabre, Jérôme Martin, Henri-Pierre Charles & Christian Bernard. *Code generation for an application-specific VLIW processor with clustered, addressable register files*. In Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems (ODES), pages 11–19, Shenzhen, China, February 2013. (p. 71)

[Lodi 06] Andrea Lodi, Andrea Cappelli, Massimo Bocchi, Claudio Mucci, Massimilano Innocenti, C. DeBartolomeis, Luca Ciccarelli, Roberto Giansante, Antonio Deledda, Fabio Campi, M. Toma & R. Guerrieri. *XiSystem : A XiRisc-Based SoC With Reconfigurable IO Module*. IEEE Journal of Solid-State Circuits, vol. 41, no. 1, pages 85–96, January 2006. (p. 14)

[Lomüller 14] Victor Lomüller & Henri-Pierre Charles. *A LLVM Extension for the Generation of Low Overhead Runtime Program Specializer*. In Proceedings of International Workshop on Adaptive Self-tuning Computing Systems at the international conference on High Performance and Embedded Architecture and Compilation (HiPEAC), Vienna, Austria, January 2014. (p. 75)

[Lucarz 08] Christophe Lucarz, Marco Mattavelli, Matthieu Wipliez, Ghislain Roquier, Mickaël Raulet, Jörn W. Janneck, Ian D. Miller & David B. Parlour. *Dataflow/actor-oriented language for the design of complex signal processing systems*. In Conference on Design and Architectures for Signal and Image Processing (DASIP), pages 168–175, Brussels, Belgium, November 2008. (p. 45)

[Marquet 10] Kevin Marquet & Matthieu Moy. *PinaVM : a SystemC front-end based on an executable intermediate representation*. In Proceedings of the tenth ACM international conference on Embedded Software (EmSoft), pages 79–88, Scottsdale, Arizona, October 2010. (p. 50, 54, 56, 94)

[Martin 09] Jérôme Martin, Christian Bernard, Fabien Clermidy & Yves Durand. *A Microprogrammable Memory Controller for high-performance dataflow applications*. In European Solid State Circuit Conference (ESSCIRC), pages 348–351, Athens, Greece, September 2009. (p. 13, 60)

[Mei 02] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man & Rudy Lauwereins. *DRESC : A retargetable compiler for coarse-grained reconfigurable architectures*. In IEEE International Conference on Field-Programmable Technology (FPT), pages 166–173, Hong Kong, Hong Kong, December 2002. (p. 11, 18)

[Minden 07] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, a. M. Wyglinski & A. Agah. *KUAR : A Flexible Software-Defined Radio Development Platform*. In 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN), pages 428–439, Dublin, Ireland, April 2007. (p. 11)

[Mitola III 92] J. Mitola III. *Software Radios Survey, Critical Evaluation and Future Directions*. In National Telesystems Conference, pages 13/15–13/23, Washington, DC, May 1992. (p. 4)

[Moreira 12] Orlando Moreira. *Temporal analysis and scheduling of hard real-time radios running on a multi-processor*. PhD thesis, TU Eindhoven, Eindhoven, Nederlands, January 2012. (p. 4, 24, 30, 96)

[Moy 13] Christophe Moy & Jacques Palicot. *20 ans de radio logicielle, quelles réalités ?* Revue de l'électricité et de l'électronique (REE), no. 1, pages 70–80, 2013. (p. 17)

[Mrabti 09] Amin El Mrabti, Hamed Sheibanyrad, Frédéric Rousseau, Frédéric Petrot, Romain Lemaire & Jérôme Martin. *Abstract Description of System Application and Hardware Architecture for Hardware/Software Code Generation*. In 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pages 567–574, Patras, Greece, August 2009. (p. 4, 62)

[Nishijima 06] Seicchi Nishijima, Miyoshi Saito & Iwao Sugiyama. *Single-Chip Baseband Signal Processor for Software-Defined Radio*. FUJITSU Sci. Tech. J, vol. 42, no. 2, pages 240–247, April 2006. (p. 13)

[Nutaq 14] Nutaq. `http://www.nutaq.com`, 2014. (p. 15, 17)

[Nvidia 13] Nvidia. *NVIDIA SDR (Software Defined Radio) Technology*. White paper, Nvidia, 2013. (p. 96)

[OAI 14] OAI. *Open Air Interface*. `http://www.openairinterface.org`, 2014. (p. 14, 17)

[Panda 01] Preeti Ranjan Panda. *SystemC - A modeling platform supporting multiple design*. In Proceedings of the 14th International Symposium on Systems Synthesis (ISSS), pages 75–80, Montreal, Quebec, September 2001. (p. 46)

[Pentek 14] Pentek. `http://www.pentek.com`, 2014. (p. 15)

[Peterson 77] James L. Peterson. *Petri nets.* ACM Computing Surveys (CSUR), vol. 9, no. 3, pages 223–252, September 1977. (p. 22)

[Pop 13] Antoniu Pop & Albert Cohen. *OpenStream : Expressiveness and data-flow compilation of OpenMP streaming programs.* ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 4, page 53, January 2013. (p. 98)

[Proebsting 96] Todd A. Proebsting & Scott A. Watterson. *Filter fusion.* In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pages 119–130, St. Petersburg Beach, Florida, January 1996. (p. 66)

[Pulley 03] D. Pulley & R. Baines. *Software defined baseband processing for 3G base stations.* In Fourth International Conference on 3G Mobile Communication Technologies, volume 2003, pages 123–127, London, United Kingdom, June 2003. (p. 12)

[Ramacher 07] Ulrich Ramacher. *Software-Defined Radio Prospects for Multistandard Mobile Phones.* Computer, vol. 40, no. 10, pages 62–69, October 2007. (p. 12, 15)

[Ramacher 11] Ulrich Ramacher, Wolfgang Raab, J. A. Ulrich Hachmann, Dominik Langen, Jörg Berthold, R. Kramer, A. Schackow, Cyprian Grassmann, Mirko Sauermann, P. Szreder, F. Capar, G. Obradovic, W. Xu, Nico Brüls, Kang Lee, Eugene Weber, Ray Kuhn & John Harrington. *Architecture and implementation of a Software-Defined Radio baseband processor.* In International Symposium on Circuits and Systems (ISCAS), pages 2193–2196, Rio de Janeiro, Brazil, May 2011. (p. 12, 96)

[Satarkar 09] Sumit Satarkar. Performance analysis of the winc2r platform. Master's thesis, Rutgers, The State University of New Jersey, New Brunswick, New Jersey, October 2009. (p. 15, 16)

[Schmidt-Knorreck 12] C Schmidt-Knorreck, Renaud Pacalet, A. Minwegen, U. Deidersen, Torsten Kempf, R. Knopp & Gerd Ascheid. *Flexible front-end processing for software defined radio applications using application specific instruction-set processors.* In Conference on Design and Architectures for Signal and Image Processing (DASIP), pages 1–8, Karlsruhe, Germany, October 2012. (p. 14, 15)

[Schulte 04] Michael J. Schulte, John Glossner, Suman Mamidi, Mayan Moudgill & Stamatis Vassiliadis. *A low-power multithreaded processor for baseband communication systems.* Computer Systems : Architectures, Modeling, and Simulation, vol. 3133, no. LNCS, pages 393–402, 2004. (p. 12, 15)

[Simeone 08] Osvaldo Simeone, Igor Stanojev, Stefano Savazzi, Yeheskel Bar-Ness, Umberto Spagnolini & R Pickholtz. *Spectrum leasing to cooperating secondary ad hoc networks.* IEEE Journal on Selected Areas in Communications, vol. 26, no. 1, pages 203–213, January 2008. (p. 97)

[Singh 13] Amit Kumar Singh, Muhammad Shafique, Akash Kumar & Jörg Henkel. *Mapping on multi/many-core systems : survey of current and emerging trends.* In Proceedings of the 50th Annual Design Automation Conference (DAC), pages 1–10, Austin, Texas, June 2013. (p. 65)

[Sundance 14] Sundance. `http://www.sundance.com`, 2014. (p. 15)

[Sutton 10] Paul D. Sutton, Jö Lotze, Hicham Lahlou, Suhaib A. Fahmy, Keith E. Nolan, Baris Ozgul, Thomas W. Rondeau, Juanjo Noguera & Linda E. Doyle. *Iris : an architecture for cognitive radio networking testbeds*. IEEE Communications Magazine, vol. 48, no. 9, pages 114–122, September 2010. (p. 19, 65)

[SystemVue 14] Agilent SystemVue. `http://www.agilent.com/find/eesof-systemvue`, 2014. (p. 19)

[Tan 09] Ceryen Tan. A hybrid static/dynamic approach to scheduling stream programs. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2009. (p. 39)

[Tan 11] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang & Geoffrey M. Voelker. *Sora : high-performance software radio using general-purpose multi-core processors*. Communications of the ACM, vol. 54, no. 1, pages 99–107, January 2011. (p. 10)

[Theelen 06] B.D. Theelen, M.C.W. Geilen, Twan Basten, J.P.M. Voeten, S.V. Gheorghita & Sander Stuijk. *A scenario-aware data flow model for combined long-run average and worst-case performance analysis*. In Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pages 185–194, Napa, California, July 2006. (p. 24)

[Thies 02] William Thies, Michal Karczmarek & Saman Amarasinghe. *StreamIt : A language for streaming applications*. Compiler Construction, vol. 2304, no. Lecture Notes in Computer Science, pages 179–196, 2002. (p. 44)

[Thies 09] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 2009. (p. 18, 66)

[Truong 09] Dean N. Truong, Wayne H. Cheng, Tinoosh Mohsenin, Zhiyi Yu, Anthony T. Jacobson, Gouri Landge, Michael J. Meeuwsen, Christine Watnik, Anh T. Tran, Zhibin Xiao, Eric W. Work, Jeremy W. Webb, Paul V. Mejia & Bevan M. Baas. *A 167-Processor Computational Platform in 65 nm CMOS*. IEEE Journal of Solid-State Circuits, vol. 44, no. 4, pages 1130–1144, April 2009. (p. 12, 15)

[Ulversoy 10] Tore Ulversoy. *Software defined radio : Challenges and opportunities*. IEEE Communications Surveys & Tutorials, vol. 12, no. 4, pages 531–550, 2010. (p. 10, 99)

[USRP 14] USRP. *Universal Software Radio Peripheral*. `http://www.ettus.com`, 2014. (p. 10, 15, 17)

[Wang 07] Perry H. Wang, Jamison D. Collins, Gautham N Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y Yang, Guei-yuan Lueh & Hong Wang. *EXOCHI : architecture and programming environment for a heterogeneous multi-core multithreaded system*. In ACM SIGPLAN conference on Programming language design and implementation (PLDI), page 156, San Diego, California, June 2007. (p. 17, 18)

[WARP 14] WARP. `http://warp.rice.edu`, 2014. (p. 14, 15, 16)

[Woh 06] Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti & Krisztian Flautner. *SODA : A Low-power Architecture For Software Radio*. In 33rd International Symposium on Computer Architecture (ISCA), pages 89–101, Boston, Massachusetts, June 2006. (p. 12)

[Woh 07] Mark Woh, Sangwon Seo, Hyunseok Lee, Yuan Lin, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti & Krisztian Flautner. *The next generation challenge for software defined radio*. Embedded Computer Systems : Architecture, Modeling, and Simulation, vol. 4599, no. LNCS, pages 343–354, July 2007. (p. 80)

[Woh 08] Mark Woh, Yuan Lin, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, Richard Bruce, Danny Kershaw, Alastair Reid, Mladen Wilder & Krisztian Flautner. *From SODA to scotch : The evolution of a wireless baseband processor*. In 41st IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 152–163, Como, Italy, November 2008. (p. 12, 15)

[Zhang 09] Q. Zhang, A.B.J. Kokkeler, G.J.M. Smit & K.H.G. Walters. *Cognitive Radio baseband processing on a reconfigurable platform*. Physical Communication, vol. 2, no. 1-2, pages 33–46, March 2009. (p. 14)

[Zyren 07] Jim Zyren & W. McCoy. *Overview of the 3GPP long term evolution physical layer*. Rapport technique, Freescale Semiconductor Inc., 2007. (p. 79)