Spécification et compilation de réseaux de neurones embarqués

Hugo Pompougnac

To cite this version:

Hugo Pompougnac. Spécification et compilation de réseaux de neurones embarqués. Autre [cs.OH]. Sorbonne Université, 2022. Français. NNT : 2022SORUS436 . tel-03997036

HAL Id: tel-03997036
https://theses.hal.science/tel-03997036

Submitted on 20 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

Spécialité Informatique
(École Doctorale Informatique, Télécommunication et Électronique)

Présentée par HUGO POMPOUGNAC

Pour obtenir le grade de
DOCTEUR DE SORBONNE UNIVERSITÉ

SPÉCIFICATION ET COMPILATION DE RÉSEAUX DE NEURONES EMBARQUÉS

Thèse soutenue le 9 décembre 2022
Jury :

Dr.	Corinne Ancourt	ENSMP/CRI	Rapportrice
Dr.	Timothy Bourke	INRIA	Examinateur
Dr.	Albert Cohen	Google France	Examinateur
Dr.	David Monniaux	CNRS	Rapporteur
Prof.	Alix Munier-Kordon	Sorbonne Univ.	Examinatrice
Dr.	Dumitru Potop-Butucaru	INRIA	Directeur de thèse

“La chouette de Minerve prend son envol au crépuscule.”
G.W.F Hegel, *Principes de la philosophie du droit*

Remerciements

Cette entreprise fut pour moi un voyage en terre inconnue. Je me serais perdu cent fois si Dumitru Potop-Butucaru, mon directeur de thèse, n'avait pas été là pour guider mes pas. Je le remercie infiniment pour le temps qu'il y a consacré, pour sa patience, pour la confiance et la bienveillance qu'il n'a cessé de me témoigner.

Albert Cohen fut également un point de repère précieux. Ses observations, ses conseils et ses encouragements ont beaucoup compté.

L'intérêt que Robert de Simone a immédiatement porté à mes travaux m'a donné courage et détermination. Avec lui, c'est également la communauté du synchrone, périodiquement rencontrée lors du *workshop* Synchron, que je remercie.

Patricia Riveill, assistante de l'équipe-projet Kairos de l'INRIA, fut à chaque instant une présence aidante et disponible. Je remercie également Anne Mathurin pour ses "coups de pouce" administratifs dans la dernière ligne droite.

Je remercie Corinne Ancourt et David Monniaux, rapporteuse et rapporteur de ma thèse. Le dialogue avec chacun d'entre eux fut à la fois gratifiant et enrichissant ; il m'a permis de jeter un regard différent sur ma contribution et de mieux la mettre en perspective.

Je remercie Alix Munier-Kordon et Timothy Bourke, qui ont également bien voulu se rendre disponibles pour participer à mon jury de thèse.

Le chemin menant au seuil du doctorat, lui-même, fut escarpé. Je remercie celles et ceux qui m'y ont accompagné. Il s'agit en particulier de Christine Da Silva, Vincent Piumi, Julien Rolland, Yann Régis-Gianas, Jean-Baptiste Yunès, Philippe Plasson.

Je voudrais finalement remercier mes proches. Même confiné, je n'ai jamais été seul durant ces trois années : le monde dans lequel je vis est peuplé de personnes combattives, aimantes et généreuses. Elles ont été et demeurent plus importantes que je ne saurais le dire.

Résumé

Dans cette thèse, nous proposons une approche pour spécifier et compiler conjointement les aspects Calcul Haute-Performance (HPC) et Temps-Réel Embarqué (RTE) d'un même système. Notre approche est fondée sur une intégration formelle, algorithmique et outillée entre deux formalismes sous-tendant une bonne partie des travaux en HPC et en RTE : le formalisme SSA et le langage flot de données synchrone Lustre. Le formalisme SSA est au cœur de bon nombre de compilateurs HPC, dont ceux employés par les *frameworks* d'apprentissage machine tels TensorFlow ou PyTorch. Le langage Lustre est au cœur des processus de mise en œuvre de systèmes embarqués critiques dans l'avionique, ou encore le rail.

Abstract

In this thesis, we propose an approach for the joint specification and compilation of both High-Performance Computing (HPC) and Real-Time Embedded (RTE) aspects of a system. Our approach is based on a formal, algorithmic and tooled integration between two formalisms underlying a large part of works in HPC and RTE fields: the SSA formalism and the synchronous dataflow language Lustre. The SSA formalism is a key component of many HPC compilers, including those used by Machine Learning frameworks such as TensorFlow or PyTorch. The Lustre language is a key component of implementation processes of critical embedded systems in avionics or rail transportation.

Table des matières

1 Introduction **5**
1.1 Embarquer la haute-performance ? 6
1.1.1 Une préoccupation ancienne 7
1.1.2 L'heure des réseaux de neurones profonds 7
1.1.3 L'embarquement des ANNs/DNNs : une question tardive 8
1.2 RTE et DL/HPC : deux domaines distincts 9
1.2.1 Garantie statique de correction vs optimisation incrémentale 9
1.2.2 Parallélisme de tâches vs parallélisme de données . . . 11
1.2.3 Réactif vs transformationnel 11
1.2.4 Une utilisation différente du paradigme flot de données 13
1.3 Contribution 16
1.3.1 Notre approche 16
1.3.2 Démarche détaillée 19

2 État de l'art **22**
2.1 Introduction 22
2.1.1 Compilation SSA 22
2.1.2 Compilation de langages synchrones 23
2.1.3 Spécification et implantation embarquée de réseaux de neurones 24
2.2 Approches supervisées 28
2.2.1 ANNs opérant sur le temps 28
2.2.2 Le point de vue transformationnel 30
2.2.3 Le point de vue réactif 33
2.3 Les applications de *Reinforcement Learning* 38
2.3.1 Les agents RL 39
2.3.2 Interactions entre réseaux de neurones 40

2.3.3 Normalisation lors de l'entraînement 41

3 Au cœur de la compilation HPC avec SSA et MLIR 42
3.1 Le principe SSA 43
3.2 Le formalisme SSA : syntaxe et sémantique intuitive 44
3.2.1 Graphe de flot de contrôle séquentiel 44
3.2.2 L'opérateur ϕ 46
3.2.3 La syntaxe et ses extensions 47
3.2.4 Propriétés de correction 49
3.3 Sémantique formelle 54
3.4 MLIR : Multi Level Intermediate Representation 58
3.4.1 Les régions 59
3.4.2 Les dialectes 61
3.4.3 La compilation d'une spécification MLIR 65
3.5 Conclusion 71

4 Étendre SSA à la concurrence synchrone 73
4.1 Le découpage de l'exécution en cycles disjoints 75
4.1.1 La nécessité d'une barrière de cycle 75
4.1.2 Associer les cycles aux *basic blocks* ? 76
4.1.3 L'opération `tick` 78
4.1.4 Compilation 81
4.2 Les entrées-sorties cycliques 85
4.2.1 De la barrière de cycle aux entrées-sorties cycliques . . 85
4.2.2 Les canaux d'entrée-sortie 86
4.2.3 Propriétés structurelles 87
4.2.4 Compilation 87
4.3 Modularité réactive 88
4.3.1 De la modularité basée sur les fonctions à la modularité réactive 88
4.3.2 L'instanciation des fonctions réactives 90
4.3.3 Contraintes structurelles 91
4.3.4 Compilation 91
4.4 Sémantique formelle 91
4.4.1 Absence et indéfinition 92
4.4.2 Règles de sémantique opérationnelle 95
4.4.3 Correction de la compilation. 96
4.5 Conclusion 97

5 Flot de données synchrone en MLIR **100**
5.1 Le langage Lustre 101
5.1.1 Les nœuds 103
5.1.2 L'opérateur fby 104
5.1.3 Les opérateurs when et merge 105
5.1.4 Les horloges 106
5.2 Le dialecte `lus` 108
5.2.1 Spécification flot de données synchrone en dialecte **lus** 108
5.2.2 Normalisation d'une spécification 111
5.2.3 La compilation d'un nœud normalisé 114
5.3 Conclusion 117

6 Intégration **119**
6.1 Structure d'un *framework* ML 119
6.1.1 *Frameworks* ML traditionnels (transformationnels) . . 119
6.1.2 *Framework* ML réactif 129
6.2 Transformations de code 136
6.2.1 Description de l'exemple : LSTM 136
6.2.2 Génération de code transformationnelle 138
6.2.3 Génération de code réactive 140
6.3 Conclusion 149

7 Résultats **151**
7.1 Réseaux de neurones 151
7.1.1 Présentation de Resnet50 152
7.1.2 Performances (Resnet50 et RNN) 153
7.2 Vocoder 159
7.2.1 Nœud principal 160
7.2.2 Nœuds instanciés par le nœud principal 162
7.3 Conclusion 163

8 Conclusion **169**
8.1 Notre contribution 170
8.2 Travaux ultérieurs 172

Table des acronymes **174**

Chapitre 1

Introduction

Bien que le temps-réel embarqué (RTE, *Real-Time Embedded*) et le calcul haute-performance (HPC, *High-Performance Computing*) soient des domaines différents, ils sont appelés à s'interfacer étroitement à l'avenir. En réalité, ils fonctionnent déjà ensemble dans le présent : qu'on pense, par exemple, à la place croissante qu'occupent les réseaux de neurones artificiels (ANNs, *Artificial Neural Networks*) dans l'autonomie véhiculaire.

Néanmoins, la mise en œuvre de composants HPC dans des systèmes RTE est techniquement difficile. Faute d'interfaces standardisées et de pratiques unifiées, il est souvent nécessaire de recourir à des approches *ad hoc* pour exécuter conjointement des fragments de code provenant de chaînes de conception différentes, qui suivent des modèles d'exécution et des objectifs d'optimisation différents.

Nous proposons d'adresser ce problème dès le niveau de la spécification. La principale contribution de cette thèse est un formalisme qui permet de spécifier et de compiler conjointement les aspects RTE et HPC d'un même système embarqué haute-performance.

Ce chapitre introductif vise à donner une vision d'ensemble du problème et de la solution proposée, avant que les chapitres suivants n'en détaillent les aspects techniques. Ainsi, l'introduction est divisée en trois parties :

- Dans la première, nous expliquons en quoi la jonction HPC/RTE, bien que difficile, est non seulement nécessaire mais est mise au premier plan dans le contexte de l'essor des ANNs.

- Dans la deuxième, nous donnons une vue d'ensemble des difficultés qu'engendrent les différences entre les approches et formalismes utili-

sés en HPC et RTE. Nous montrons que ces difficultés sont d'autant plus nettes dans les applications d'apprentissage profond (DL, *Deep Learning*) lorsque leur spécification fonctionnelle dépend de l'axe du temps.

- Dans la dernière partie, nous résumons les principes de la solution que nous proposons, qui consiste à étendre la programmation HPC avec des primitives réactives tirées de formalismes RTE.

1.1 Embarquer la haute-performance ?

Les formalismes et les méthodes consacrés à l'implémentation de programmes RTE existent depuis longtemps et permettent de contrôler des systèmes divers et souvent critiques dans l'automobile, l'avionique, les installations industrielles, etc. Conçus pour être sûrs et pour éliminer les risques de dégâts humains ou matériels, ces systèmes de contrôle embarqués peuvent nécessiter de renoncer à la pleine exploitation des performances de la plate-forme d'exécution pour atteindre les niveaux requis de prédictibilité. Parmi les éléments auxquels on renonce souvent dans les systèmes critiques, on retrouve :

- des mécanismes matériels comme les caches partagés, la cohérence mémoire, la spéculation ;
- des mécanismes d'exécution, comme l'ordonnancement dynamique ;
- des optimisations qui rendent le code difficilement analysable et traçable, comme les niveaux supérieurs d'optimisation (par exemple, à partir de -O3 en GCC/LLVM).

Par comparaison, dans les systèmes HPC, historiquement, on a surtout visé la performance, généralement mesurée de manière empirique sur des *benchmarks* incluant des cas d'utilisation typiques[103]. Le matériel y est utilisé typiquement sans les restrictions mentionnées plus haut.

Dès lors, l'incorporation de composants HPC dans des systèmes RTE a très tôt représenté un défi : alors qu'ils viennent de domaines dont les approches, les objectifs et les techniques diffèrent significativement, comment combiner, sans annuler leurs propriétés respectives, ces deux sous-ensembles logiciels ?

1.1.1 Une préoccupation ancienne

Ce défi a pris, au fil du temps, différentes formes.

On pense par exemple aux applications traditionnelles de traitement de signal – notamment dans le contexte RTE des radars, ou de la télémétrie des satellites. De telles applications ont à réaliser des calculs importants (e.g. FFT) tout en respectant des exigences strictes (temps de réponse, occupation mémoire, etc). Elles se sont donc appuyées sur des écosystèmes spécialisés, qu'il s'agisse de bibliothèques HPC *target-dependent* (implémentant par exemple le standard BLAS[37]) ou de compilateurs *domain-specific* (à l'image de LGen[62]), générant un code efficace que les concepteurs ont ensuite la responsabilité d'embarquer manuellement. Les aspects HPC et RTE de ces systèmes sont donc traités distinctement, ce qui peut poser des problèmes de robustesse ; par exemple, le fait que la parallélisation et l'optimisation d'un système temps-réel n'aient pas lieu conjointement à l'allocation de ressources complique l'intégration.

Plus récemment, les *jumeaux numériques* – simulant en temps-réel le fonctionnement d'un système physique à des fins de maintenance prédictive ou de contrôle prédictif – ont pris une place considérable. On embarque notamment des jumeaux numériques dans des installations de production énergétique ou dans l'avionique, et la manipulation des données sous des contraintes temps-réel est un défi central[10].

1.1.2 L'heure des réseaux de neurones profonds

L'essor des réseaux de neurones profonds (DNNs, *Deep Neural Networks*) pose les mêmes problèmes, d'abord parce que ces derniers sont utilisés dans les domaines évoqués précédemment. Par exemple, on les retrouve de plus en plus dans les jumeaux numériques[65]. Le rapprochement de ces domaines applicatifs a d'ailleurs lieu dans les deux sens : on recourt notamment aux modèles prédictifs dans les boucles d'apprentissage par renforcement (RL, *Reinforcement Learning*)[57], avec les *modèles du monde*[44] dont on estime qu'ils joueront un rôle central à l'avenir[25].

Plus généralement, le DL, comme branche de l'apprentissage machine (ML, *Machine Learning*), se heurte à des difficultés qui le précèdent, mais qu'il met au premier plan :

- Du fait des résultats spectaculaires de ces technologies et des approches

associées, elles se généralisent et les secteurs industriels qui en dépendent doivent accélérer leur mise en œuvre, malgré les difficultés.

- Du fait de la démocratisation de l'informatique embarquée "intelligente" (smartphones, internet des objets, compagnons domotiques, véhicules autonomes ou semi-autonomes...), le besoin s'exprime également à l'échelle de millions (ou milliards) de particuliers, bien qu'à des niveaux de criticité variables.

Cependant, les techniques sur lesquelles repose le DL n'ont pas été conçues pour ces usages embarqués. Elles adoptent même, par certains aspects, des approches qui divergent nettement avec l'embarquabilité des applications et dont nous discuterons par la suite.

1.1.3 L'embarquement des ANNs/DNNs : une question tardive

Les intuitions fondamentales liées aux ANNs, introduites par analogie avec les neurones biologiques, datent des années 1940[70]. Le perceptron de Rosenblatt[87], en 1958, montra qu'un système arficiel peut apprendre par expérience, et cet apprentissage commença à devenir *profond* en 1965, avec la première définition d'un réseau formé de différentes couches calculatoires[53]. Le domaine prit finalement son essor à partir de la fin des années 1980, lorsque l'algorithme de rétropropagation du gradient fut mis en œuvre avec succès pour entraîner un perceptron multi-couches (MLP, *Multi-Layer Perceptron*) et le réseau récurrent (RNN, *Recurrent Neural Network*) équivalent [88]. L'arrivée des réseaux *convolutifs* (CNNs, *Convolutional Neural Networks*) [28, 26] acheva de donner naissance au DL et aux DNNs.

Bien que l'algorithme de rétropropagation du gradient soit bien plus efficace que les technique d'apprentissage précédentes, il reste coûteux en calcul, en mémoire, et en communications. Il a donc fallu adopter des méthodes résolument tournées vers la performance pour l'implémenter, parfois sous la forme d'heuristiques [1]. Il a aussi fallu que l'état de l'art HPC, à la fois scientifique et technique, autorise le recours à la puissance de calcul, au stockage et

[1] C'est notamment le cas du compromis entre vitesse de convergence et précision des résultats sur lequel repose le choix du *pas d'apprentissage* d'un DNN[54].

aux débits de communication nécessaires [2]. La parallélisation des traitements sur les données, en particulier, a joué un rôle majeur.

Une fois ce point critique atteint et leur utilité démontrée, les ANNs ont pu se développer et se diversifier au-delà de l'apprentissage *supervisé*[3]. En particulier, leur incorporation dans les boucles RL, où l'apprentissage est guidé par un signal de *récompense* fourni par l'environnement de l'application, a pris une importance croissante. Souvent pensées pour l'embarqué, y compris critique[61], ces dernières matérialisent des choix traditionnellement liés au monde RTE.

1.2 RTE et DL/HPC : deux domaines distincts

La difficulté majeure, à l'heure d'embarquer la haute-performance, vient du fait que l'expertise concernant ces systèmes est aujourd'hui distribuée entre deux domaines scientifiques et d'ingénierie bien distincts, chacun avec ses modèles formels, ses algorithmes et ses pratiques industrielles propres.

1.2.1 Garantie statique de correction vs optimisation incrémentale

La conception de systèmes RTE vise à garantir que ceux-ci sont non seulement fonctionnellement corrects, mais qu'en plus ils respectent les exigences non-fonctionnelles auxquelles ils sont soumis (e.g. exigences temps-réel, exigences d'embarquabilité). Il s'agit de systèmes éventuellement critiques et subissant des contraintes physiques importantes : centrales nucléaires, avions, satellites, etc. On raisonne donc à leur sujet en termes d'analyse au pires cas (*Worst Case Analysis*, WCA) pour obtenir des garanties sur les bornes dans lesquelles ils s'exécutent, qu'elles soient temporelles (temps de réaction

[2]La disponibilité de données d'entraînement en quantité suffisante a également représenté un enjeu. Yann Le Cun raconte ainsi que la mise à disposition, par Bell Labs, de 9298 images de chiffres manuscrits collectés par l'USPS a permis un véritable saut qualitatif dans sa recherche[24]. Ce même cap doit encore aujourd'hui être franchi dans plusieurs applications industrielles du DL embarqué, pour lesquelles le manque de jeux de données constitués est un véritable facteur limitant[13].

[3]Où un ANN apprend par rétropropagation à partir de jeux de données étiquetées par des opérateurs humains.

du système à un *stimulus*), énergétiques (énergie consommée par le système durant son exécution) ou de taille mémoire.

L'optimisation se concentre donc sur le niveau système, qui gère l'allocation de ressources aux différentes *tâches*. Typiquement, elle ne vise pas à minimiser ou maximiser un objectif de coût, mais à permettre d'assurer que les exigences sont satisfaites.

De l'autre côté, les travaux HPC se concentrent principalement sur l'optimisation des programmes pour le cas moyen visant à maximiser leur vitesse, minimiser leur taille de code, etc. Cela ouvre la voie à l'utilisation d'optimisations et de mécanismes d'exécution introduisant une forte variabilité des temps d'exécution, mais qui sont très efficaces au cas moyen.

Du fait de cette forte variabilité temporelle, on s'expose à un écart important entre le cas et moyen et le pire cas, et les garanties temporelles au pire cas ne peuvent être que de faible précision.

Il arrive souvent que la spécification précise du matériel et/ou des mécanismes d'exécution HPC mobilisés manque ou qu'elle soit incomplète, rendant l'analyse statique impossible. Dans ce cas, seule la correction fonctionnelle peut être assurée par analyse statique.

Compiler un programme HPC consiste à transformer incrémentalement le code, par une suite de "passes de compilation", préservant sa sémantique fonctionnelle. Chaque passe de compilation applique des optimisations et/ou réécrit le code à un niveau d'abstraction plus bas, plus loin du langage de spécification et plus proche du niveau de l'implémentation. On "descend" ainsi (c'est pourquoi on parle usuellement de *lowering*) jusqu'au code exécutable – lequel doit réaliser les traitements décrits dans le programme source.

Des travaux œuvrant à la prédictibilité de code HPC existent, même en présence de mécanismes à forte variabilité temporelle comme l'exécution spéculative[42], la cohérence des caches[49], l'ordonnancement dynamique[109], l'exécution dans le désordre (*out-of-order execution*)[104] ou les architectures matérielles spécifiques[15]. Cependant, ces recherches visant à réconcilier la performance et la prédictibilité doivent encore être mises en œuvre en pratique – et elles n'ont de toutes manières pas le caractère systématique requis par l'état de l'art des optimisations HPC, certains aspects seulement étant pris en compte.

1.2.2 Parallélisme de tâches vs parallélisme de données

Les approches HPC se concentrent principalement sur l'exploitation du parallélisme des données. On y met à profit les traitements réguliers, comme les nids de boucles[11], par différentes formes de parallélisation : vectorisation/-SIMD, pipelinage matériel et logiciel, parallélisation sur multi-cœurs, réseaux systoliques dans les accélérateurs neuronaux, etc. Le lien avec la machine, sur lequel reposent les gains de performances, est très fort. Dans le contexte du DL, en particulier, le parallélisme est omniprésent :

- **Au niveau spécification.** Les opérations DL les plus courantes et les plus coûteuses sont très parallèles (*embarassingly parallel*), à l'instar des convolutions. De plus, le niveau de parallélisme des spécifications est encore augmenté au moyen du traitement par lots (*batch processing*).

- **Au niveau compilation.** Au-delà de la compilation d'opérations de haut niveau vers des boucles affines, les transformations de code sont étendues au moyen de compilateurs (e.g. XLA) et d'IRs (e.g. HLO) *domain-specific* (e.g. algèbre linéaire) conçus pour maximiser les débits de calcul.

Les approches RTE, quant à elles, visent principalement des applications à parallélisme de tâches. Partant de l'hypothèse que le code de bas-niveau est encapsulé en un nombre réduit de *tâches*, l'objectif est ici d'assurer l'orchestration de ces tâches de manière à assurer leur exécution tout en respectant les exigences de temps d'exécution, de sûreté de fonctionnement, *etc.* Chaque tâche est décrite, souvent de manière très abstraite, au moyen d'un code séquentiel (plus rarement parallèle) avec des caractéristiques non-fonctionnelles comme le pire temps d'exécution (WCET, *Worst Case Execution Time*), la pire consommation mémoire, etc.

La hiérarchie des tâches et leurs interactions sont souvent décrites sous la forme de graphes flot-de-données (DF, *Data Flow*). Si le paradigme DF est utilisé dans les deux mondes, il l'est donc de manière très différente.

1.2.3 Réactif vs transformationnel

HPC et le paradigme transformationnel. Les programmes DL/HPC sont usuellement des fonctions classiques : on leur fournit leurs entrées, ils les utilisent pour accomplir une série de calculs, puis retournent, une fois pour

toutes, leurs résultats. Si le même calcul doit avoir lieu sur d'autres entrées, la fonction peut être appelée à nouveau. Suivant la classification de Harel et Pnueli[48], on parle de systèmes transformationnels.

Le problème de la classification d'images donne l'intuition de ce comportement : l'application reçoit une image en entrée et retourne, en sortie, l'étiquette qu'elle attribue à l'image. Travaillant sur des données qui n'ont pas de dimension temporelle (comme un ensemble de photographies), il est parfaitement naturel de concevoir leur traitement comme l'appel séparé d'une fonction sur chaque donnée/photographie. L'essor des DNNs étant intimement lié au problème de la reconnaissance d'images[4], les réseaux de neurones ont, très tôt, été implémentés de cette manière.

Dans un registre voisin, le principe même de l'entraînement supervisé est transformationnel : il consiste en effet à alimenter un modèle DL avec un ensemble fini d'entrées annotées, à partir desquelles le calcul produit l'ensemble des poids entraînés.

Aujourd'hui encore, les formalismes flot de données (e.g. Keras[60] [5], PyTorch[84], JAX[56]) auxquels on recourt pour décrire leur comportement portent l'empreinte de cette genèse. Pour autant, dès lors qu'il s'agit de traiter des données qui ont une dimension temporelle (e.g. vidéo, séquences sonores), ce modèle transformationnel devient contre-intuitif et, sous certaines conditions, contre-productif.

RTE et le paradigme réactif. À l'inverse, les systèmes RTE sont réactifs. Leur fonction est de *réagir* continuellement à des *stimuli* externes. Un système RTE typique va cycliquement acquérir des entrées, réaliser des calculs, et produire des sorties. Le calcul de ces réactions est souvent soumis à des *exigences non-fonctionnelles*, par exemple *temps-réel* (e.g. le calcul d'une réaction doit être borné dans le temps).

Prenons l'exemple du satellite PLATO[1], qui est notamment développé au LESIA de l'Observatoire de Paris et que l'ESA prévoit de lancer en 2026. Conçu pour détecter des exoplanètes dans la zone habitable de certaines étoiles, la spécification de l'instrument exige qu'il dispose d'unités de traitement des données (DPU) réalisant en parallèle les opérations suivantes :

- Les données acquises par les deux caméras reliées à chaque DPU sont

[4]Le réseau LeNet[27], le premier "grand" réseau que Y. Le Cun construit à Bell Labs[24], est ainsi conçu pour classer des chiffres manuscrits.

[5]Keras est l'API haut-niveau placée au sommet du *framework* TensorFlow[98].

lues toutes les 25 secondes, et des données partielles sont acquises toutes les 6.25 secondes ;

- Les images sont traitées pour en extraire courbure de la lumière, centroïdes et "imagettes"[6] ;
- Ces données raffinées sont communiquées au composant qui doit les compresser et les envoyer au sol (en les écrivant dans la télémétrie de l'instrument) ;
- Un deuxième algorithme, relié à des caméras plus rapides, fait des lectures toutes les 2.5 secondes.

L'ensemble est une application *réactive* multi-périodique.

Le parallélisme de tâches des systèmes réactifs est souvent mis en évidence au travers de langages et de formalismes dédiés, comme les formalismes synchrones[7] et flot-de-données.

1.2.4 Une utilisation différente du paradigme flot de données

Malgré la relative étanchéité entre les communautés HPC et RTE, elles utilisent toutes les deux, au niveau spécification, le paradigme flot-de-données. Il est utilisé dans les deux domaines de manière très différente :

- Les formalismes flot de données synchrones comme Scade/Lustre[8, 45] et les formalismes graphiques type Simulink sont utilisés pour décrire les comportements réactifs de systèmes de tâches complexes (e.g. systèmes de tâches dépendantes).
- Les formalismes flots de données utilisés pour la spécification des modèles DL comme Keras, décrivent des fonctions incorporant souvent des comportements fortement cycliques (e.g. itérations) mais fondamentalement transformationnelles.

Cette dichotomie existe en fait dans le paradigme flot de données dès sa formalisation par J. Dennis en 1974/1975, entre graphes flot de données décrivant des architectures matérielles réactives[32] et graphes flot de données

[6] Plus petites images, rognées pour être centrées sur les étoiles ciblées.

décrivant des composants logiciels transformationnels incorporant des comportements cycliques[31]. La première approche a été adoptée par le monde RTE, la seconde par le monde DL/HPC.

Le paradoxe du DL embarqué. Or, un modèle DL embarqué est un système réactif, qu'il est nécessaire de pouvoir implémenter de manière réactive à partir d'une spécification réactive (exécution infinie, entrées/sorties cycliques, etc.). Dès lors, le caractère transformationnel des *frameworks* DL traditionnels pose problème. Bien que le flot de données de ces derniers admette des comportements cycliques au sens limité vu ci-dessus, il décrit une exécution finie où toutes les entrées/sorties sont acquises et produites de manière monolithique. Les spécifications y font donc l'objet de passes de compilation incrémentales basées sur des IRs transformationnelles, dérivant de SSA ou proches de SSA[86, 95], au terme desquelles le réseau est implémenté sous la forme de fonctions compilées. Le recours à cette chaîne d'outils transformationnelle pour spécifier/implémenter un ANN réactif est donc *un paradoxe*.

En effet, le concepteur du modèle doit ainsi écrire une spécification transformationnelle *compatible* avec l'implémentation réactive ultérieure et la compiler de manière transformationnelle. Les fonctions compilées sont ensuite incorporées manuellement dans un système réactif, qui peut être un *runtime* exécutant simplement le code d'inférence/prédiction de manière cyclique sur un flot d'entrées, ou un agent RL plus complexe, mélangeant phases d'entraînement et d'inférence aux mêmes pas de temps.

Ce détour par le transformationnel est illustré en Fig. 1.1, où les flèches rouges désignent les ajustements manuels *ad hoc* requis par l'usage des *frameworks* traditionnels. Il a des conséquences au niveau de l'implémentation comme de la spécification.

Niveau implémentation. Un tel environnement technique, d'abord, ne permet pas de recourir pleinement à l'état de l'art de l'implémentation RTE. En effet, même l'exécution d'un simple MLP séquentiel implique des méthodes d'allocation des ressources complexes, par exemple basées sur le *pipelining* logiciel[16], où le traitement de plusieurs échantillons a lieu de manière concurrente et synchronisée. Or, de tels composants sont naturellement modélisés au moyen de formalismes réactifs. De la même manière, le recours à ces formalismes permettrait de faciliter l'interfaçage, au niveau système, de

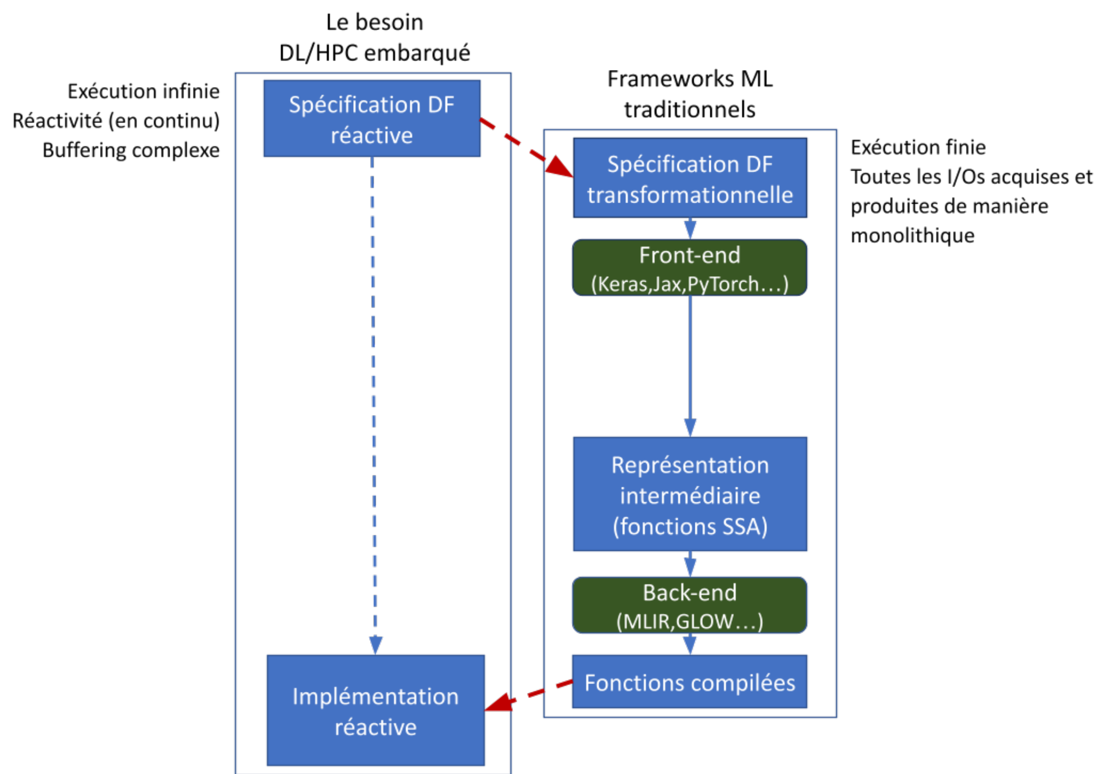


FIGURE 1.1 : Le détour contre-intuitif par le transformationnel exigé par l'implémentation réactive d'un modèle DL.

l'application avec son environnement, et donc aussi :

- de faciliter l'utilisation des techniques issues du monde RTE (e.g. WCAs) ;
- de contribuer à l'unification des implémentations dont la variété actuelle complique notamment la reproductibilité des résultats RL[78].

Niveau spécification. Au-delà de l'implémentation, l'arrière-plan transformationnel des formalismes ML a aussi un impact dès le niveau de la spécification : les comportements réactifs des ANNs, qu'ils soient embarqués ou non, y sont décrits au moyen de structures de contrôle qui n'ont pas le caractère systématique (e.g. primitives générales, sémantique formelle) que l'on retrouve dans un formalisme réactif synchrone comme Lustre. Ils sont donc plus difficiles à programmer et à vérifier. Un formalisme comme Keras permet notamment de spécifier, au-delà des DAGs d'opérations quelconques, de l'exécution conditionnelle (dépendante ou non des données) et même de la *récurrence*. Nombreux sont ainsi les ANNs dont la correction dépend du temps, non seulement d'un point de vue non-fonctionnel (e.g. WCET), mais également d'un point de vue fonctionnel. C'est le cas dès lors que le traitement des données dépend de leur dimension temporelle – c'est-à-dire que des *dépendances temporelles* relient les calculs entre eux.

1.3 Contribution

Le paradoxe que nous avons présenté complique et limite le recours, dans les applications ML, aux méthodes du monde RTE. Notre objectif est au contraire d'œuvrer à *intégrer* ces différentes approches au plan formel et outillé, afin de permettre la spécification et la compilation **conjointes** des aspects HPC et RTE d'un même système.

1.3.1 Notre approche

La solution que nous présentons dans cette thèse repose sur trois constats :

1. Les *frameworks* ML (e.g. Keras/TensorFlow, PyTorch) concentrent le savoir-faire en compilation HPC, ainsi que la communauté faisant vivre ces outils de compilation. Ils sont donc incontournables dans toute approche pratique d'implémentation DL, embarquée ou non.

2. Ces *frameworks*, dont les représentations intermédiaires sont soit fondées sur SSA, soit proches de SSA, se heurtent à des limites importantes pour représenter et compiler les aspects réactifs d'un système.

3. Les aspects réactifs d'un ANN peuvent être programmés – avec profit – au moyen de formalismes flot de données synchrones issus du monde RTE.

Nous choisissons donc d'étendre l'IR SSA, qui sert de *back-end* à plusieurs *frameworks* ML majeurs, vers deux formalismes synchrones :

- les primitives flot de contrôle synchrones qui offrent une représentation bas niveau des comportements synchrones ;

- les primitives flot de données synchrones de Lustre, bâties sur le socle des précédentes, qui permettent de spécifier les aspects réactifs d'une application à haut niveau.

Nous étendons SSA au niveau spécification, en y incorporant les opérateurs réactifs synchrones permettant de décrire les comportements réactifs, mais également au niveau compilation, en proposant des méthodes de compilation implémentant ces opérateurs synchrones sous la forme de constructions SSA traditionnelles. Ainsi, cette approche donne accès, dans le contexte de la programmation HPC/ML, aux techniques RTE basées sur les formalismes réactifs synchrones et à la généralité de leurs primitives temporelles. Elle n'implique pas pour autant de renoncer aux opérations et transformations de code HPC coagulées dans les principaux *frameworks* ML. Elle permet donc de construire un *framework* réactif entièrement outillé, comme illustré en Fig. 1.2, duquel le détour manuel par le transformationnel exposé en Fig. 1.1 est éliminé. Dans un tel *framework*, l'exécution finie induite par l'approche transformationnelle est vue comme un cas particulier de l'exécution *a priori* infinie induite par l'approche réactive.

Notre contribution se situe donc au niveau langage et compilation. Cependant, deux ensembles de travaux auxquels elle donne accès sont en-dehors du champ de cette thèse :

1. **Différentiation automatique des opérations réactives.** La synthèse de code d'apprentissage n'est pas couverte par ce manuscrit, étant l'objet de travaux encore en cours.

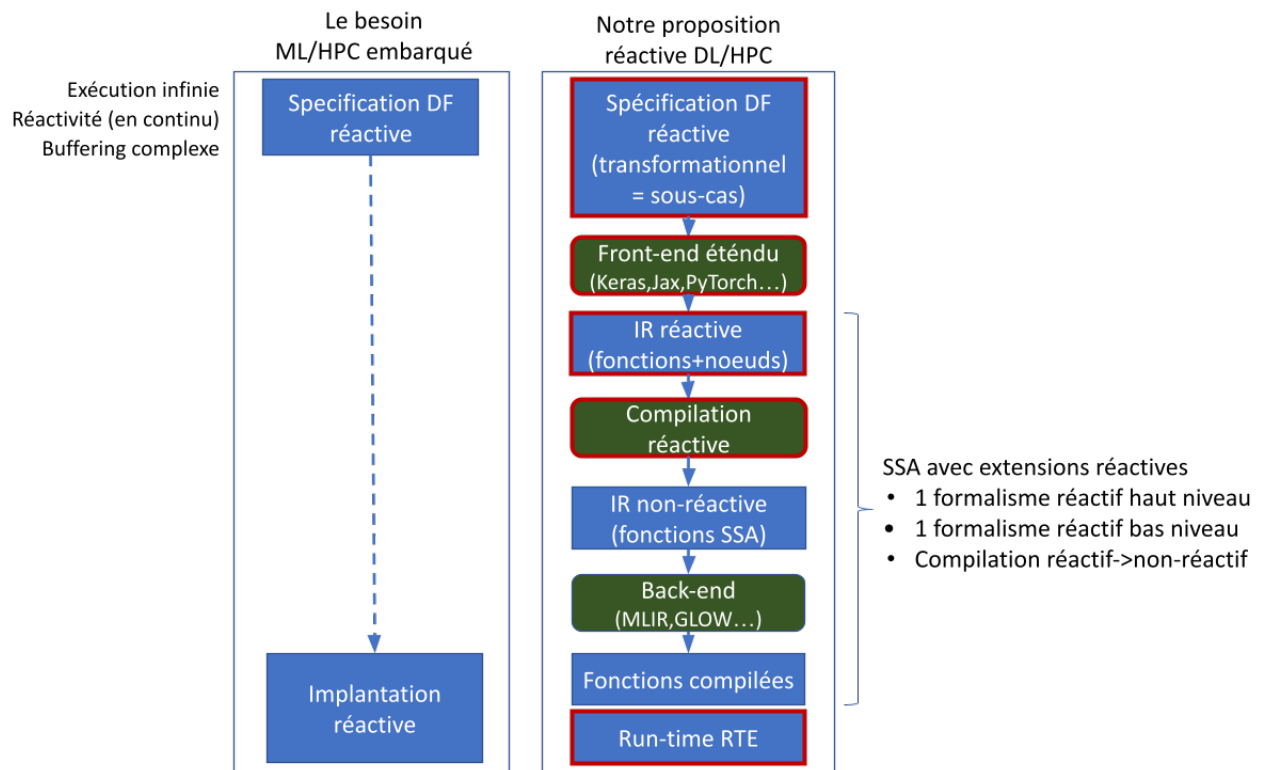


FIGURE 1.2 : Le *framework* réactif rendu possible par l'extension de SSA à deux formalismes réactifs synchrones.

2. **Compilation et *runtime* efficaces et prédictibles.** Notre contribution permet d'ores et déjà de générer du code fonctionnellement correct, optimisé par les *back-ends* des *frameworks* ML et interfaçable avec le code système. Cependant, une solution complète demanderait davantage de travaux d'interfaçage entre *runtimes* RTE et ML, et le recours à des techniques de génération de code orientées prédictibilité (e.g. allocation statique de la mémoire vs allocation dynamique).

1.3.2 Démarche détaillée

État de l'art. Au Chapitre 2, nous présentons les travaux reliés à notre contribution dans l'état de l'art en compilation à base de forme normale SSA et en compilation de langages synchrones. Nous proposons ensuite une vue d'ensemble des travaux visant la spécification et la compilation de réseaux de neurones d'un point de vue orienté vers une mise en œuvre embarquée.

Étude sémantique de SSA. Nous réalisons d'abord au Chapitre 3 une étude sémantique de SSA pour comprendre quels mécanismes manquent pour représenter la réactivité synchrone. Comme base de nos travaux, nous choisissons le dialecte SSA de MLIR[71]. Les avantages de ce choix sont multiples : en particulier, MLIR est une IR conçue pour être extensible et, distribuée avec les compilateurs LLVM, elle a une base de développeurs conséquente. On la retrouve ainsi en en *backend* de nombreux *frameworks* ML.

Extension réactive flot de contrôle de SSA. Nous proposons au Chapitre 4 une extension de la syntaxe et de la sémantique de SSA de manière à pouvoir représenter les primitives permettant aux composants concurrents d'un programme d'interagir dans le temps, entre eux et avec leur environnement. Ces primitives permettent de définir l'affectation de chaque opération à son cycle d'exécution synchrone, mais également la lecture et l'écriture d'entrées-sorties cycliques, la modularité synchrone (implémentée par des nœuds/processus communiquants, et non par des fonctions classiques) et l'absence d'un signal lors d'un cycle. Ce dernier point présente une difficulté particulière pour s'insérer dans une spécification SSA correcte, puisque le formalisme SSA interdit[29], au moyen de contraintes structurelles sur lesquelles nous reviendrons en Chapitre 3, qu'une donnée non-initialisée puisse exister dans une spécification. Nous relions donc la notion d'absence à un

usage généralisé des valeurs *indéfinies* dans la sémantique SSA[30] et dans les représentations bas niveau comme LLVM IR[64].

Flot de données synchrone en MLIR. Alors que notre extension réactive de SSA fournit un niveau naturel pour la description d'implémentations réactives, il nous faut aussi permettre une spécficication naturelle de haut niveau pour ces systèmes. Comme dans beaucoup de domaines (contrôle embarqué, traitement de signal, multi-média, ML) ce niveau suit une forme flot de données, nous proposons au Chapitre 5 l'incorporation du noyau sémantique du langage Lustre dans MLIR.

Nous introduisons dans MLIR les primitives flot de données Lustre `when` et `merge`, ainsi que l'analyse statique (*clock calculus*) par laquelle Lustre garantit qu'une valeur absente (non-produite et donc non-initialisée) n'est jamais utilisée dans aucun calcul. Nous relions ces mécanismes de transmission conditionnelle des données aux travaux sur la prédication des opérations SSA[17, 74].

La représentation flot de données de l'*état* d'un nœud Lustre pose davantage de problèmes. En effet, la primitive Lustre `fby`, par laquelle Lustre spécifie la transmission de valeurs d'un cycle à l'autre (autorisant une forme de mémoire entre les cycles) recourt à une syntaxe incompatible avec les contraintes SSA. Nous montrons donc qu'une phase de normalisation permet de les rendre ce comportement compatible avec SSA.

Cas d'études. Aux Chapitres 6 et 7, nous mettons ces extensions à l'épreuve dans une chaîne de compilation entièrement outillée allant de la spécification flot de données à l'exécution de code réactif sur plate-formes CPU ou GPU. Les cas d'étude applicatifs sont choisis pour évaluer deux critères :

- **L'expressivité.** Nous spécifions et compilons une application de traitement de signal RTE traditionnelle (un *vocoder* manipulant des données sonores) qui embarque un contrôle réactif complexe (e.g. fenêtres glissantes) et des filtres classiques (e.g. FFTs).

  Nous compilons également un RNN basé sur LSTM[51], lesquels sont massivement utilisés mais plus complexes que les DAGs convolutifs qui ont présidé à l'essor du DL.

- **Performances.** Nous montrons que notre extension n'engendre pas de pessimisation des performances par rapport à l'état de l'art HPC : nous

exhibons à cet égard un ResNet50 (réseau résiduel[50] de 50 couches), très profond, qui, s'il n'a rien d'original au plan du contrôle, est particulièrement intensif en calcul.

Chapitre 2

État de l'art

2.1 Introduction

Cette thèse avance l'état de l'art dans trois domaines :

1. Compilation (généraliste et HPC) à base de forme normale SSA ;
2. Programmation réactive et synchrone ;
3. Formalisation et compilation du ML.

Des états de l'art compréhensifs existent pour les domaines 1-2 (la compilation fondée sur SSA[95] et la conception et la compilation de langages synchrones [47, 7, 81, 9, 93]). Nous ne citerons à leur sujet que quelques références dont on peut identifier une relation particulière avec nos travaux. Le gros de ce chapitre sera dédié à une tentative de systématisation des travaux visant la spécification et la compilation de réseaux de neurones d'un point de vue orienté vers une mise en œuvre embarquée.

2.1.1 Compilation SSA

Par rapport à l'état de l'art SSA[95], nos travaux fournissent les constructions syntaxiques, les exigences de correction structurelles et les extensions sémantiques permettant la représentation en forme SSA de comportements réactifs synchrones. Cette contribution est complètement originale. Cependant, des aspects particuliers nécessaires à notre extension ont déjà été abordés dans les travaux précédents concernant l'exécution conditionnée par des prédicats

(*predicated execution*)[17, 74] et dans ceux définissant une valeur sémantique absente ou non-définie à utiliser lors de la compilation[30, 64].

Dans les travaux précédents, les prédicats conditionnant l'exécution doivent être représentés explicitement à chaque point du programme dont l'exécution dépend du prédicat. Dans nos travaux la prédication est implicite, comme dans les approches flot de données où l'absence d'entrées détermine l'absence d'exécution[22]. Cela demande une définition claire de la correction de ces programmes, absente dans [17, 74].

De même, notre traitement de l'absence est différent de celui proposé dans le monde SSA, notamment par LLVM[64]. Notre objectif n'est pas d'assurer la préservation de la sémantique (de l'ensemble de traces) d'un programme lors de sa compilation lorsque ces traces peuvent impliquer des valeurs absentes/non-définies. Dans notre cas, la compilation n'est envisagée que pour les programmes où les valeurs absentes ne sont jamais utilisées dans les calculs ou dans les décisions. Certaines variables peuvent demeurer absentes lors d'une exécution, mais dans ce cas elles ne doivent pas être utilisées. Lier ces deux visions de la compilation est réalisé par le théorème d'abstraction en Chapitre 4.

2.1.2 Compilation de langages synchrones

Dans l'état de l'art des langages et formalismes synchrones, les compilateurs restructurent le code profondément pour atteindre sa forme classique comprenant une fonction `step` réalisant les pas de calcul synchrones et une fonction `reset` visant à initialiser l'état de calcul avant l'exécution [1]. L'exécution de la fonction `step` peut ensuite être parallélisée[33] ou pipelinée[16] et quelques approches existent[34, 75] permettant directement la synthèse d'un code multi-périodes, mais globalement il reste très difficile de sortir de ces quelques structures d'exécution. Dans cette thèse, nous proposons une approche bien plus souple permettant l'exécution d'une spécification synchrone comme un réseau d'automates communiquants, chacun cachant son état. Il s'agit donc d'une approche plus modulaire. Dans cette approche, chaque composant (automate/nœud réactif) peut être soumis à toute transformation légale de code SSA (e.g. déroulement de boucle). Les générations de code traditionnelles du monde synchrone sont toujours possibles, mais pas

[1]Cette restructuration sera abordée *in extenso* en Section 4.1.4. La même section propose aussi des générations de code alternatives non-couvertes par l'approche classique.

obligatoires.

Nous montrons aussi que lors de la compilation d'un langage synchrone comme Lustre, on peut réutiliser sans changement un grand nombre de techniques fondamentales de la compilation traditionnelle à base SSA : analyse de types, analyse de causalité, ordonnancement, allocation mémoire etc. Seules quelques étapes de compilation spécifiques au monde synchrone s'avèrent incontournables, notamment l'analyse d'horloges et la synthèse du contrôle impératif explicite. Ainsi, la compilation de langages synchrones comme Lustre peut être vue comme une extension de la compilation traditionnelle, ce qui permet des avancées en formalisation et en allocation prédictible de ressources pour systèmes réactifs haute-performance.

2.1.3 Spécification et implantation embarquée de réseaux de neurones

L'état de l'art en spécification et implantation embarquée de réseaux de neurones artificiels (ANNs) présente, quant à lui, un caractère bien plus disparate. Nous expliquons ici cet état de fait, et définissons le périmètre couvert par notre état de l'art.

L'expressivité croissante de la programmation ML. La programmation ML, en partant de réseaux très simples, a progressivement (et rapidement) gagné en expressivité. Son évolution peut être mise en relation avec celle des langages généralistes à partir des années 50 :

1. Elle a étendu le perceptron originel vers les réseaux séquentiels, lesquels consistent à appliquer une simple séquence de couches calculatoires aux entrées pour produire les sorties. Les MLPs ou les CNNs les plus élémentaires en sont des exemples.

2. La diffusion (*broadcasting*) est venue ensuite, donnant la possibilité de représenter des graphes orientés acycliques (DAGs, *Direct Acyclic Graphs*) généraux. Les ANNs appartenant à la famille des *réseaux résiduels*, que nous présentons en Section 7.1.1, exploitent largement cette propriété.

3. L'itération avec état s'est également superposée aux mécanismes antérieurs, au travers des RNNs (*Recurrent Neural Networks*, dont font

partie les réseaux basés sur LSTM que nous étudions en Section 6.2), ou des agents de RL (*Reinforcement Learning*) dotés d'une mémoire de reprise (*replay memory*).

4. L'exécution conditionnelle est venue sous des formes diverses, qu'il s'agisse de la réinitialisation implicite des réseaux récurrents ou de comportements plus généraux comme le `select` flot de données et même l'activation conditionnelle.

5. Les réseaux récurrents bidirectionnels ont étendu la récurrence en permettant de parcourir l'axe du temps dans les deux sens (vers l'avant et vers l'arrière).

6. Les réseaux récursifs ont permis d'incorporer la récursion générale aux ANNs.

Notre analyse couvre les réseaux dont une implantation embarquée (à ressources bornées) peut être donnée, ce qui correspond principalement aux niveaux 1 à 4 ci-dessus. En effet, le niveau 5 implique de mémoriser complètement l'intervalle de temps le long duquel la récurrence est appliquée ; quand cet intervalle de temps n'est pas statiquement borné, borner l'occupation mémoire et le temps d'exécution n'est pas possible. Le niveau 6 se heurte également à ces difficultés de prédictibilité de la consommation de ressources, ici engendrées par le recours à la récursion générale.

L'extension *ad-hoc* des *frameworks* ML. Les gains en expressivité de la programmation ML ont souvent été réalisés par l'extension *ad-hoc* de *front-ends* programmés en Python au sein de *frameworks* ML comme TensorFlow ou PyTorch. On peut retracer certaines de ces couches sédimentaires, par exemple, en considérant keras-rl (surcouche de Keras pour le RL) ou tf-df (surcouche de TensorFlow couvrant les modèles récursifs à base d'arbres de décision). Cette approche d'extension des langages par (finalement) leurs utilisateurs a permis une agilité certaine, mais pose désormais de nouveaux problèmes.

D'une part, les constructions incorporées aux formalismes ML de cette manière sont souvent peu générales, étant adaptées à un cas d'utilisation unique. Ainsi, l'itération à état peut être décrite par la spécification de l'état d'un RNN ou par la spécification de la *replay memory* d'un agent RL, mais

il n'existe pas de notion générale subsumant ces deux cas d'utilisation, qui permettrait par ailleurs aussi de représenter le traitement cyclique réactif d'entrées-sorties dans le temps. L'état de l'art que nous proposons vise ainsi à identifier les problèmes généraux sous-jacents à ce qui se présente en l'état comme une collection de problèmes disjoints.

D'autre part, les gains d'expressivité des formalismes de spécification en *front-end* n'ont été que partiellement reportés dans les représentations intermédiaires des *back-ends* de compilation. S'agissant de l'itération à état le long du temps, nous verrons par exemple qu'elle est mise en œuvre, dans les compilateurs ML, au moyen d'une itération de boucle sur les dimensions d'un tableau déjà stocké en mémoire.

Les mécanismes d'expression généraux que nous proposons visent à adresser les deux aspects de ce problème :

- Fournir un langage plus homogène, simple et formel pour la définition d'algorithmes ML embarquables. Cela revient aussi à proposer aux concepteurs d'algorithmes ML une vision plus claire de la complexité des algorithmes qu'ils développent, dans un contexte où la frugalité énergétique devrait être un des critères centraux dans la conception de nouveaux algorithmes.

- Contribuer à combler le *fossé* entre *front-end* et *back-end* en rapprochant la sémantique des représentations intermédiaires de la sémantique des spécifications.

Apprentissage et inférence. Le principal critère de classification des approches ML porte sur la manière dont le processus d'apprentissage est mis en place. On identifie généralement 3 grandes classes d'approches :

1. L'approche dite *supervisée* consiste à guider l'apprentissage par une fonction d'erreur calculant l'écart entre le résultat du calcul et le résultat attendu (matérialisé par les annotations que le *data scientist* a préalablement associé aux données d'entrées). L'entraînement du réseau y est entièrement accompli *avant* l'inférence (traitement de données nouvelles/inconnues), cette dernière étant typiquement la seule pour laquelle une mise en œuvre embarquée est envisagée.

2. L'approche dite *par renforcement* consiste à guider l'apprentissage par un signal de récompense, lequel permet au réseau d'évaluer ses résultats

à chaque pas de temps ; les phases d'entraînement et d'inférence y ont donc lieu en même temps. Très populaire, notamment, dans le domaine de la voiture autonome, cette approche est plus générale que celle basée sur l'apprentissage supervisé (un problème d'apprentissage supervisé peut toujours être reformulé comme un problème d'apprentissage par renforcement, mais l'inverse n'est pas vrai[92]).

3. L'approche dite *non-supervisée* consiste à traiter des données sans étiquetage préalable ni signal de récompense, de manière à en extraire les motifs et régularités sous-jacents. Dans la mesure où les systèmes embarqués doivent typiquement fournir des garanties de correction, ces approches non-supervisées ont été moins étudiées dans ce contexte et nous ne les couvrons pas ici.

L'état de l'art ML des Sections 2.2 et 2.3 suit cette classification et correspond aux approches 1 et 2.

***Focus* sur le contrôle de haut niveau.** La définition, la spécification et la compilation des opérations tensorielles centrales dans l'algorithmique HPC/ML [2], ainsi que les méthodes mathématiques sur lesquelles reposent l'entraînement des ANNs (e.g. différentiation automatique, algorithme du gradient), sont hors des objectifs de cette thèse. De ce point de vue, bien qu'ils soient certainement perfectibles, nous considérons que les formalismes et compilateurs ML existants fournissent les meilleures solutions. Nous n'évoquons donc ces aspects que ponctuellement et superficiellement dans notre état de l'art.

Notre objectif a été de permettre une meilleure spécification de l'intégration de ces primitives tensorielles dans un système réactif, ainsi que la compilation conjointe des aspects HPC et RTE de ce même système.

À ce titre, bien que ce point soit détaillé dans les Chapitres 6 et 7, notons dès à présent que :

- Les **langages réactifs synchrones**, à l'image de Lustre, suffisent à représenter le contrôle réactif des ANNs appartenant aux classes 1 à 4 définies ci-avant ;

- La spécification réactive de ces ANNs nécessite toutes les primitives du noyau Lustre présenté au Chapitre 5.

[2] Comme les convolutions, les fonctions d'activation, etc.

C'est la raison pour laquelle nous sélectionnons cette famille de formalismes (et notamment Lustre) pour mener le travail d'unification formelle exposé en Section 1.3.

L'utilisation de mises en œuvre réactives ou de formalismes réactifs pour l'implantation d'algorithmes ML spécifiés et compilés par ailleurs a déjà été proposée auparavant[83, 107]. Cependant, ces travaux suivent une approche qui reflète le caractère *ad-hoc* des travaux en spécification et implémentation ML. En effet, la programmation ML et la programmation réactive y sont radicalement découplées, alors que nous cherchons au contraire à identifier et à exploiter leur unité formelle.

2.2 Approches supervisées

Les approches supervisées distinguent la phase d'entraînement d'un réseau, où ses poids internes sont itérativement mis à jour à partir de données d'entraînement annotées, et sa phase d'inférence, où il traite des données inconnues. Dans cette section, nous commençons par présenter les classes d'ANNs usuellement utilisées dans ces approches pour opérer le long de l'axe du temps et pour traiter les dépendances temporelles qui relient entre elles les données d'entrée. Nous présentons ensuite les problèmes liés à la mise en œuvre transformationnelle de ces réseaux, et les défis importants relatifs à leur mise en œuvre réactive.

2.2.1 ANNs opérant sur le temps

En faisant abstraction des difficultés liées à l'implémentation (sur lesquelles nous revenons dans la suite de l'exposé), les ANNs ont souvent à traiter des *séquences temporelles* formées de données réparties sur plusieurs pas de temps et liées entre elles par des dépendances temporelles. C'est par exemple le cas de l'identification de mots-clés dans un flot sonore : un échantillon sonore unique, qui dure une fraction de seconde, ne suffit pas à repérer un mot-clé. Il faut consommer plusieurs entrées successives pour pouvoir idenfitier les dépendances temporelles qui les relient et pouvoir produire un résultat positif ou négatif [3]. Nous verrons que ces comportements sont souvent décrits et

[3] Ce problème, dans le cas particulier où les entrées sont de simples scalaires, est connu sous le nom de *classification de séries temporelles* (*TimeSeries Classification*[36]).

implémentés au moyen de techniques de bufferization *ad-hoc*, faute de disposer des primitives générales fournies par des formalismes réactifs synchrones comme Lustre. Dans un premier temps, nous présentons les classes d'ANNs les plus courantes pour opérer sur l'axe du temps.

MLPs et CNNs.

Le problème de l'identification de corrélations temporelles dans des séquences temporelles a d'abord été vu comme une variante du problème consistant à identifier des corrélations spatiales (e.g. dans une image). On lui a donc appliqué les techniques pré-existantes, comme les MLPs ou les CNNs.

Les filtres de convolution permettant de *factoriser* les poids du réseau[36], les CNNs ont pris une importance considérable. Si les convolutions sont particulièrement bien adaptées à l'extraction de dépendances locales (i.e. *de court terme*), elles peuvent également être composés dans des réseaux plus *profonds* pour capturer différentes échelles de temps[23], de la même manière qu'appliqués à des problèmes spatiaux, leur profondeur permet de repérer des corrélations à différentes échelles [4]. Bien sûr, le nombre des calculs nécessaires peut vite devenir un problème ; c'est pourquoi des travaux cherchent à le limiter sans renoncer à la précision des résultats. C'est le cas des deux variantes de TCNs (*Temporal Convolutional Networks*) [63], qui consistent toutes les deux à réduire la quantité d'information nécessaire :

- L'architecture encodeur/décodeur des *Encoder-Decoder TCNs* permet de limiter le nombre de couches calculatoires nécessaires au moyen d'opérations de sur-échantillonnage et de sous-échantillonnage des entrées.

- Les convolutions dilatées des *Dilated TCNs*, sans limiter le nombre des couches calculatoires, limitent le nombre de calculs à chaque couche en insèrant des "trous" dans les filtres de convolution.

Ces approches portent une intuition fortement transformationnelle, en cela qu'elles supposent la disponibilité immédiate de données réparties le long du temps (et non leur traitement séquentiel).

[4]Par exemple, sur une photographie, repérer successivement une silhouette, un visage et la couleur des yeux.

RNNs

Les RNNs sont d'abord apparus comme une alternative aux MLPs pour traiter les dépendances temporelles[88]. Ils portent donc une intuition fortement réactive (leur implémentation transformationnelle, que nous présentons plus loin, est d'autant plus paradoxale).

Dans un RNN, une couche récurrente traite ses données d'entrée les unes après les autres. Elle applique à chaque échantillon le même calcul, et des valeurs peuvent être transmises d'un cycle de calcul au suivant. On appelle ces valeurs, qui décrivent les dépendances temporelles, *l'état* d'un RNN. Les RNNs sont strictement plus expressifs que les CNNs/MLPs : un RNN peut décrire le même calcul qu'un réseau convolutif à plusieurs niveaux (par *pipelinage* des couches calculatoires), mais les chaînes de dépendances temporelles qu'il représente peuvent être longues, portentiellement non-bornées dans le temps, à la différence de celles que peuvent représenter les réseaux convolutifs. Cela facilite, par exemple, la conception de systèmes d'identification de mots-clefs, mais demande une grande attention à la définition des mécanismes de transmission de l'état pour éviter le problème de disparition du gradient (*gradient vanishing problem*). Ces travaux ont notamment abouti à la définition de RNNs particuliers comme LSTM[51].

Dans les formalismes ML de haut niveau (e.g. Keras), la spécification de la récurrence d'un RNN est réalisée par des constructions hiérarchiques dédiées. Le réseau global est acyclique, mais certaines de ses *couches* peuvent être récurrentes. Chacune de ces couches récurrentes englobe un autre graphe flot de données acyclique décrivant le calcul d'un cycle. La structure de l'état est décrite dans la définition même d'une couche récurrente. Ainsi, la définition d'un RNN n'est jamais un simple DAG, mais un DAG hiérarchique. Couches récurrentes, convolutives et autres peuvent être librement mélangées pour tirer profit de leurs bonnes propriétés respectives[90].

2.2.2 Le point de vue transformationnel

Alors que la spécification de haut niveau de réseaux de neurones se fait par des formalismes flot-de données (dont l'intuition est naturellement réactive), les *frameworks* ML les plus utilisés (Keras, PyTorch) donnent à ces spécifications une sémantique et une mise en œuvre transformationnelle. Cette approche paradoxale a d'importantes conséquences pratiques.

La conversion temps-espace totale

La conversion temps-espace totale consiste à considérer que l'axe du temps est converti en une dimension spatiale supplémentaire des données d'entrée (et de sortie) de l'application. Une telle conversion n'est possible que si le calcul se fait sur une durée de temps finie, pour permettre sa représentation sous la forme d'une matrice ou d'un tenseur. La totalité des données d'entrée correspondant à la durée de calcul est reçue en même temps.

Lorsque l'ANN concerné s'exécute hors-ligne, consommant des données qui sont *effectivement* disponibles d'emblée en totalité, cette approche est naturelle. Mais lorsqu'il s'exécute dans un système RTE dont les entrées arrivent sous la forme d'un flot séquentiel et éventuellement illimité dans le temps (e.g. les images provenant de la caméra d'une voiture autonome), la conversion temps-espace totale implique de convertir le temps dans l'espace *à la volée*, par exemple au moyen d'une fenêtre glissante sur l'axe du temps empilant les entrées jusqu'à ce que le vecteur résultant atteigne une certaine longueur. Les segments de données créés par ce procédé sont ensuite traités chacun sous l'hypothèse de conversion temps-espace totale, de manière transformationnelle.

La conversion temps-espace totale est la norme dans les *frameworks* ML les plus courants (Keras, PyTorch...). Ainsi, en Keras, une couche récurrente comme LSTM prendra en entrée un tenseur tridimensionnel dont les dimensions sont : la taille du *batch* (nombre de calculs indépendants à mener en parallèle), la longueur de la dimension temporelle, et finalement la taille spatiale des échantillons correspondant à chaque pas temporel. De plus, le comportement par défaut[5] de la couche LSTM en Keras est de ne produire sa sortie qu'au dernier pas temporel. Ainsi, la logique transformationnelle apparaît nettement.

La longueur des segments temporels. Dans cette approche, la longueur des segments traités est un élément central du calcul [6] ; par exemple, dans un ANN entraîné pour identifier des mots-clés dans un flot de son, si on considère qu'un mot-clé n'excède jamais une seconde de son, les segments traités par le réseau doivent agréger autant d'échantillons que nécessaire pour arriver à une

[5]Des annotations permettent d'exiger la production de la sortie pour chaque pas temporel. Cependant, cette sortie adviendra sous la forme d'un seul tenseur contenant tous les résultats.

[6]L'influence de ce paramètre est discutée au-delà du domaine du ML[91].

seconde de données sonores. Lorsque les segments pertinents sont de tailles variables (certains mots-clés peuvent être très longs, d'autres plus courts), il faut les ramener à la même échelle, au moyen de calculs[97] qui ajoutent complexité et fragilité au système.

Compilation. Lors des premières étapes de compilation, toute intuition réactive est perdue quand la forme flot de données est remplacée par la forme impérative/SSA des formats intermédiaires de compilation comme MLIR. Ainsi, le calcul cyclique et à état d'un RNN sur une dimension temporelle finie est représentée par une itération à taille fixe sur une dimension de tableau.

Par exemple, dans le *framework* TensorFlow, une couche LSTM sera transformée en une fonction représentant le corps de la boucle, itérée à l'aide d'une construction `while`.

Implantation réactive

Le code transformationnel issu de la compilation décrite plus haut peut être utilisé de plusieurs manières dans des implantations réactives/embarquées.

Dans le cas où un réseau récurrent comprend une seule couche, la solution la plus simple est de faire la compilation pour une dimension temporelle de longueur 1, et ensuite d'itérer cette fonction dans le temps, en prenant soin de transmettre l'état d'itération d'un cycle à l'autre. Le code réalisant cette itération, ainsi que l'interaction avec les entrées et les sorties, doit être manuellement écrit. Dans le cas général où le réseau comprend plusieurs couches, cette approche n'est plus appliquable.

La seconde approche consiste à créer une fenêtre glissante sur les entrées, et à appliquer la fonction obtenue par compilation à chaque pas d'avancement de la fenêtre. Dans ce cas, les problèmes sont liés à l'occupation mémoire et surtout au nombre de calculs requis à chaque appel. Ainsi, si la fenêtre glissante est de taille N et qu'elle avance de 1 pas temporel à chaque fois, l'entrée correspondant à un pas de temps sera traitée indépendamment par N étapes de calcul (alors que sous l'intuition réactive chaque entrée est traitée exactement une fois). Cette augmentation de la complexité de calcul doit être prise en compte lors d'une mise en œuvre temps réel.

Un autre problème lié à ce type d'implantation est fonctionnel : les dépendances temporelles prises en compte par le code généré ne peuvent aller au delà de la taille de la fenêtre glissante. Pour retrouver des dépendances de plus longue durée, il est donc nécessaire, par exemple, d'appliquer un

post-traitement aux résultats produits à chaque déplacement de la fenêtre glissante sur les entrées[19].

Un standard *de facto*

Sans que les questions précédentes ne soient résolues, ces approches forment un standard *de facto* pour l'apprentissage supervisé et sont massivement déployées, au point qu'elles sont parfois vues comme naturelles. C'est aussi vrai dans l'embarqué, dès l'origine[69] et au travers des chaînes d'outils pour l'embarqué fournies par les principaux *frameworks* ML (à l'image de TensorFlow Lite), lesquels se concentrent souvent sur d'autres aspects – comme l'empreinte mémoire et le nombre de calculs [19], ou encore la compression des modèles[73], notamment au moyen de techniques de *quantization*[39]. C'est pourquoi même des travaux venant du monde RTE conçoivent les ANNs de cette manière, à l'image de la boîte à outils DL intégrée au formalisme réactif Simulink[94], où une couche LSTM est insérée dans le flot de données en supposant que toutes ses entrées sont disponibles en même temps.

2.2.3 Le point de vue réactif

Si les principaux frameworks ML mettent en œuvre une approche transformationnelle, le besoin d'implantation embarquée et temps réel a mené à l'émergence de méthodes de mise en œuvre réactive et permettant de limiter la consommation de ressources.

Ces méthodes visent a minima l'exécution du code d'inférence. Cependant, le choix réactif a une incidence, aussi, sur la phase d'apprentissage, que la méthode peut prendre en compte.

Inférence sur flots de données

Le traitement réactif a été particulièrement mis en avant dans le domaine de recherche des applications traitant des flots de données en temps-réel ou proche du temps-réel (*streaming applications*). Un exemple classique, souvent utilisé comme *benchmark*, est celui de l'identification à la volée de mots-clefs (KWS, *Key Word Spotting*) sur la base de données audio brutes produites par un microphone.

En plus de la couche neuronale, les applications KWS doivent aussi réaliser du traitement de signal et du *buffering* non-triviaux. Ces applications

n'ont pas seulement de hautes exigences de précision (détecter les mots clefs prononcés de diverses manières et dans des contextes bruités, et sans faux positifs), mais aussi de latence (détecter rapidement) et de ressources (utiliser peu de mémoire pour permettre d'embarquer le code dans des téléphones portables, assistants domotiques, etc.).

Ces particularités, ainsi que l'omniprésence des applications de type *streaming* dans notre environnement quotidien, expliquent pourquoi leur conception (avec souvent KWS comme cas d'étude) est un domaine de recherche très productif[68]. Pour présenter les principales caractéristiques de ces approches à l'aide d'un exemple, nous choisissons la bibliothèque Python kws-streaming[89, 2], issue de l'écosystème Keras/TensorFlow.

Pour permettre la réutilisation d'une grande partie de la chaîne de génération de code Keras/TensorFlow, kws-streaming se présente comme une extension du *front-end* Keras, tant au niveau spécification que génération de code. Fig. 2.1 illustre cette phase de conversion vers le réactif.

Les couches primitives de Keras ne changent pas dans `kws-streaming` (seule leur implantation change). Les couches n'ayant pas de comportement temporel (e.g. Dense) préservent ainsi leur sémantique et la génération de code.

Les principaux changements apparaissent au niveau des couches récurrentes (LSTM, GRU) et des couches convolutives dans le temps (comme DepthwiseConv2d), dont le code Keras est remplacé. Pour les couches récurrentes, la transformation est naturelle : la récurrence est réalisée dans le temps, sans conversion temps-espace. A chaque pas temporel, une couche récurrente de `kws-streaming` va recevoir une donnée d'entrée et va produire une sortie. L'exécution est *a priori infinie*, avec une seule initialisation de l'état, réalisée au premier pas temporel.

La conversion réactive affecte aussi les convolutions dans le temps. Celles-ci n'ayant plus à disposition l'ensemble des données d'entrée pour tous les pas temporels, il faut mettre en place des mécanismes de *buffering*. Par exemple, si le noyau de convolution a une dimension de 3 sur l'axe temporel, il faut maintenir un tampon (fenêtre glissante [7]) contenant les 3 dernières entrées de la couche convolutive.

Ces changements n'affectent pas le *back-end* de compilation, Ils affectent seulement le *front-end* et le *run-time*. Ainsi, au lieu d'une fonction traitant des données après conversion temps-espace, `kws-streaming` va produire une

[7] Aussi appelée *ring buffer*.

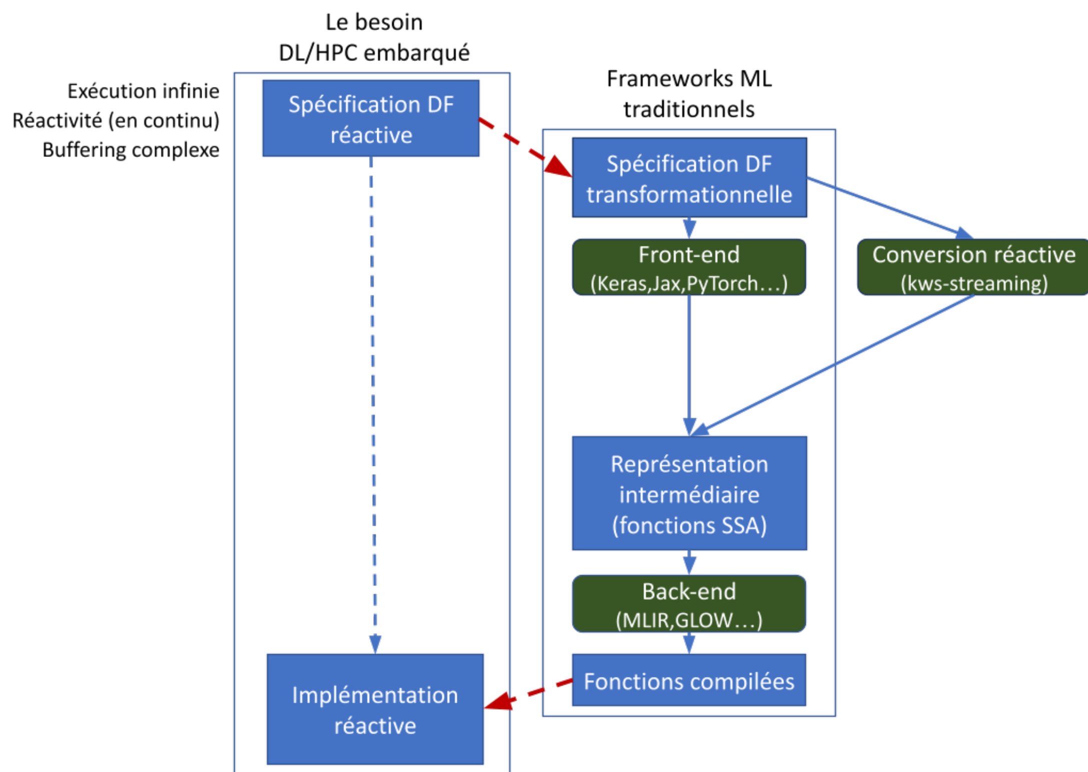


FIGURE 2.1 : La conversation réactive d'une spécification ML au moyen de `kws-streaming`.

fonction à appeler à chaque pas temporel, ainsi que le *runtime* réalisant cet appel cyclique à chaque réception d'un échantillon en entrée et maintenant l'état d'exécution entre appels. La fonction du *back-end* reste celle de compiler de la manière la plus efficace possible des fonctions.

Cette approche reste cependant limitée :

- En raison de l'utilisation des formalismes ML existants, cette approche ne dispose pas de primitives flot de données générales, ce qui limite son expressivité. Ainsi, les fenêtres glissantes ne peuvent pas être spécifiées (mais seulement être le résultat d'une compilation), la récurrence et l'exécution conditionnelle ont une expressivité limitée.

- Similairement, la mise en œuvre (compilation) de ces mécanismes réactifs suit des approches *ad-hoc* adaptées à des cas d'utilisation particuliers (comme KWS) et non pas des schémas de compilation généraux couvrant les aspects réactifs (en plus de ceux HPC).

- La mise en œuvre bas-niveau comprend souvent des mécanismes d'exécution dont la prédictibilité temporelle est difficile à assurer (comme l'allocation dynamique de la mémoire).

Implatation prédictible (temps-réel) de réseaux de neurones

Les approches présentées ici proposent des solutions au dernier problème identifié auparavant (la prédictibilité temporelle), sans pour autant apporter de solution systématique aux problèmes de spécification réactive.

Les approches visant a éliminer les sources d'imprédictibilité temporelle peuvent directement viser la conception de matériel prédictible. Les accélérateurs neuronaux utilisent souvent des mises en œuvres connues pour leur prédictibilité, tels les réseaux systoliques utilisés dans les *Tensor Processing Units* (TPU) de Google[35].

Une autre famille d'approches vise à améliorer la prédictibilité d'implantations logicielles tournant sur des architectures multi-cœurs. Ces approches peuvent viser :

- Les couches neuronales elles-mêmes, en fournissant des implantations à la fois efficaces et prédictibles[20].

- L'assemblage de ces couches en réseaux neuronaux, et leur ordonnancement prédictible sur plate-forme d'exécution[107, 76]. Ces approches

peuvent viser un remplacement de l'allocation dynamique de la mémoire par des mécanismes d'allocation statique, la simplification de la synchronisation (par des mécanismes de synchronisation globale), ou même l'augmentation du débit de calcul par l'utilisation de mécanismes de pipelinage logiciel.

Le gros inconvénient de ces approches est qu'elles perdent à la fois :

- le lien avec les *back-ends* des *frameworks* de compilation ML existants, qui concentrent le savoir-faire en compilation efficace.
- l'expressivité des *front-ends* classiques, car seule une partie des couches traditionnelles est implantée.

Apprentissage en présence de dépendances temporelles

Considérons un réseau de neurones comprenant des couches récurrentes. Dans une implantation traditionnelle, comme celle produite par les frameworks ML classiques (Keras, PyTorch), l'apprentissage est réalisé typiquement sur des sequences temporelles de longueur fixe, et avec une annotation correspondant uniquement au dernier pas temporel. L'initialisation de l'état de récurrence est donc réalisée régulièrement, à pas fixe, lors du calcul.

Par comparaison, lors d'une exécution réactive sans réduction au cas transformationnel (par fenêtre glissante, tel qu'expliqué en Section 2.2.2) les chaînes de récurrence sont non-bornées dans le temps, l'initialisation de la récurrence peut être arbitrairement éloignée dans le temps, et une sortie est attendue à chaque pas temporel (cycle de calcul).

Ces différences sont importantes, et on peut déjà avoir l'intuition de changements qualitatifs :

- L'initialisation de la récurrence devrait avoir moins d'influence sur le comportement d'un tel réseau, car la distance par rapport à celle-ci peut être arbitrairement longue.
- Des mécanismes d'oubli devraient être mis en place pour permettre (si nécessaire) de donner plus de poids au passé récent. Par exemple, dans un application de type KWS la détection dans un passé lointain d'un certain mot-clef ne devrait pas influer sur la détection de nouveaux mots-clefs dans le présent.

L'entraînement des réseaux réactifs impose par ailleurs des contraintes. Ainsi, les algorithmes à base de rétropropagation du gradient doivent mémoriser lors du parcours vers l'avant les valeurs permettant de calculer les gradients utilisés lors du parcours vers l'arrière, sur toute la longueur temporelle du cas d'entraînement. Lorsque ces longueurs temporelles sont grandes le stockage nécessaire l'est aussi, potentiellement de manière rhédibitoire. Des stratégies dédiées sont nécessaires pour réduire la quantité de données stockées [105, 106].

Finalement, d'autres solutions visent à réduire la complexité de l'entraînement en changeant la nature de l'algorithme ML. Parmi celles-ci, le *Reservoir Computing*[101]. Un réseau appartenant à cette famille, à l'image des ESN (*Echo State Networks*)[55], compose :

- Un *réservoir* récurrent dont l'état récurrent est calculé en appliquant une couche de poids non-entraînables (aléatoires) sur les entrées puis une fonction d'activation non-linéaire, dont le résultat est sommé à l'état antérieur. Cette architecture n'est pas coûteuse à différencier, mais du fait du critère aléatoire de l'activation, elle permet généralement à l'état produit lors d'un pas de temps donné de survivre le long du temps (sous la forme d'un *écho*). Ce dernier décrit alors les dépendances temporelles des entrées.

- À chaque pas de temps, l'état du réservoir est transmis à un classifieur classique (e.g. perceptron) entraîné pour décoder les dépendances temporelles qu'il encode.

2.3 Les applications de *Reinforcement Learning*

Les boucles de renforcement[57] incorporent d'emblée les modèles ML dans des composants explicitement réactifs. Les primitives générales d'un formalisme flot de données synchrones comme Lustre peuvent donc faciliter considérablement la spécification et l'implémentation de ces comportements réactifs, et notamment aider à l'unification des techniques du domaine. En effet, aujourd'hui, une grande variété d'implémentations coexistent et, comme nous l'avons évoqué précédemment, affectent même la reproductibilité des résultats de la recherche[78]. La conception de l'environnement OpenAI Gym[14]

se veut une réponse à ce besoin de standardisation des implémentations et des interfaces pour les environnements de simulation, mais à notre connaissance, rien de tel n'existe pour les environnements réels.

Dans cette section, nous présentons donc les *agents* RL, qui sont les composant réactifs fondamentaux des boucles RL. Nous présentons ensuite les comportements complexes de ces agents qui peuvent motiver le recours au flot de données synchone.

2.3.1 Les agents RL

Une application RL embarque un ou plusieurs[108] *agents*. Chaque agent peut être vu comme un processus consommant cycliquement deux signaux fournis par son environnement, un signal d'observation et un signal de récompense (qui guide l'apprentissage), et fournissant cycliquement à son environnement un signal d'action En effet, si l'on observe le système à haut niveau, à chaque pas de temps[96] :

- L'agent acquiert une *observation* décrivant l'état de son environnement externe.
- Le ou les réseaux de neurones incorporés dans l'agent traitent cette observation pour choisir une *action* à accomplir (on parle alors d'*exploitation*), ou bien l'agent sélectionne une action arbitraire (on parle alors d'*exploration* [8]) de manière à *découvrir* de nouveaux comportements. L'algorithme produisant une action à partir de l'état de l'environnement est appelé la *politique* de l'agent.
- L'agent applique l'action qu'il a choisie à son environnement externe.
- L'environnement externe produit la *récompense* attribuée à l'action de l'agent.
- Connaissant sa récompense, l'agent modifie les poids internes du ou des réseaux de neurones qu'il embarque de manière à produire, à l'avenir, des actions maximisant les récompenses qu'il reçoit aux pas de temps suivants. La politique souhaitée consiste souvent à maximiser la *récompense cumulative* (i.e. somme des récompenses le long du temps).

[8] L'exploration peut également être guidée par un composant ML préalablement entraîné de manière supervisée[77].

Le concepteur de l'agent RL :

- Choisit un type d'agent dans la littérature (ou dans le *framework* utilisé), ou en développe un nouveau ;
- Décrit précisément l'architecture du ou des réseaux de neurones qui calculent l'étape d'exploitation ;
- Fournit les autres paramètres de l'agent, comme l'agorithme d'exploration.

Les RNNs dans les agents RL. Dans la définition initiale du RL, l'observation au pas de temps courant doit suffire à choisir une action (*propriété de Markov*). En particulier, l'agent ne doit dépendre ni du passé, ni du futur pour décider. Néanmoins, le RL a été étendu pour permettre la résolution de tâches *non-markoviennes*, où prendre une décision nécessite par exemple de traiter les dépendances temporelles entre des observations arrivées successivement dans le temps. Le comportement de tels agents est naturellement décrit au moyen de RNNs dont l'état est persistant d'un pas de temps au suivant, à l'image de LSTM[5] .

2.3.2 Interactions entre réseaux de neurones

La phase d'exploitation d'un agent peut être implémentée au moyen d'un unique réseau de neurones. C'est le cas dans le QLearning[102, 72], où l'agent sélectionne une action au moyen d'une *fonction valeur* (ou fonction Q) qui, étant donné l'état courant de l'agent, attribue une valeur à chaque action possible ; l'agent sélectionne finalement l'action qui maximise la fonction valeur.

Cependant, d'autres approches combinent plusieurs réseaux de neurones. Un agent peut ainsi embarquer un *modèle de l'environnement* permettant de simuler les comportements de l'environnement dans le temps et de planifier les décisions. Les modèles du monde évoqués précédemment entrent dans cette catégorie, et les RNNs en constituent là aussi des composants naturels[43].

L'interaction entre différents composants de l'agent peut exister même sans modèle de l'environnement : dans les algorithmes *Action-Critic*[41], une fonction (l'*acteur*) sélectionne une action, qui est ensuite évaluée par une seconde fonction (le *critique*) chargée de diriger l'entraînement. L'acteur et

le critique peuvent être des réseaux de neurones distincts, ou des couches de sortie distinctes d'un même réseau. Dans un contexte temps-réel, l'interaction synchronisée de ces composants[85] s'appuie sur des hypothèses et des exigences proches de celles que l'on retrouve dans les formalismes réactifs synchrones.

2.3.3 Normalisation lors de l'entraînement

Dans le QLearning, il est courant de maintenir un historique des tuples observation/action/récompense et d'alimenter l'apprentissage, non avec le tuple produit au pas de temps courant, mais avec des tuples tirés aléatoirement de l'historique. Ce mécanisme permet notamment de ne pas entraîner l'agent à partir d'échantillons consécutifs (lesquels peuvent être fortement corrélés), et donc de limiter la variance des mises à jour des poids. Le stockage de ces tuples a lieu dans une mémoire nommée *replay buffer* dont la taille est typiquement bornée, permettant sa représentation sous un paradigme synchrone.

Chapitre 3

Au cœur de la compilation HPC avec SSA et MLIR

Nous nous intéressons ici à la forme Static Single Assignment (SSA)[95]. Elle fut initialement conçue comme un ensemble de principes et de propriétés de bien-formé[86] facilitant les optimisations de bas niveau pratiquées dans les représentations intermédiaires (IR) des compilateurs (e.g. élimination de code mort). Progressivement, elle a acquis une syntaxe et une sémantique propre [79] et est donc devenue un formalisme à part entière, qui sert de base aux représentations intermédiaires de plusieurs chaînes de référence, comme GCC, LLVM ou Swift[1]. Plus récemment encore, ses bonnes propriétés sémantiques, au premier rang desquelles la concurrence déterministe, sont finalement apparues comme un excellent support pour la compilation HPC (et donc ML), au travers d'IRs comme MLIR (*Multi-Level Intermediate Representation*)[71].

Dans ce chapitre, nous détaillons ces trois aspects :

- L'intuition derrière le "principe SSA" dont le formalisme SSA n'est qu'une mise en œuvre.
- La syntaxe et la sémantique du formalisme SSA utilisé en compilation traditionnelle au travers des IRs des compilateurs.
- Les extensions apportées par MLIR au formalisme SSA.

[1] Il existe néanmoins des compilateurs, tels CompCert[66], qui (contrairement à CompCertSSA[6]) n'y recourent pas.

```
1  x = 0;              1  x1 = 0;
2  x++;                2  x2 = x1 + 1;
3  x = f(x);           3  x3 = f(x2);
4  x = 42;             4  x4 = 42;
5  x = f(x);           5  x5 = f(x4);
```

FIGURE 3.1 : Un petit fragment de code C (gauche) et un même calcul obéissant au principe SSA (droite).

Ces éléments nous permettront, dans le chapitre suivant, d'étendre le formalisme SSA et MLIR avec des primitives de spécification réactive, indispensables dans la conception de systèmes embarqués.

3.1 Le principe SSA

Dans sa forme la plus générale, le principe SSA exige deux propriétés, l'une statique et l'autre dynamique :

1. Dans le programme, chaque variable est affectée à exactement un endroit (par une seule instruction).

2. Dans chaque *lifetime*, une variable ne peut pas être utilisée avant d'avoir été affectée.

Cette propriété est traditionnellement exigée au niveau des formats de compilation intermédiaires, mais elle peut aussi être mise en œuvre au niveau de fragments de code de plus haut niveau, comme en Fig. 3.1. Dans cet exemple, le code à gauche manipule une variable x, la réaffectant 5 fois, ce qui est interdit en SSA. Le code de droite calcule les mêmes valeurs, mais en respectant le principe SSA. La réaffection étant interdite, les 5 affectations de x donnent naissance à 5 variables distinctes x1, x2, x3, x4 et x5.

Le respect des exigences SSA permet d'éliminer les comportements indéfinis (y compris en présence de concurrence) et fonde le *déterminisme* des spécifications en forme SSA, que nous introduisons formellement en Section 3.2. Du fait de la règle de l'affectation unique, les dépendances de données suffisent à déterminer l'ordre d'exécution des opérations dans une spécification respectant le principe SSA. C'est le cas en Fig. 3.1, où la réplication des variables (à droite) permet de calculer en même temps `x1` et `x4`. Ainsi, convertir

un programme en lui faisant respecter les contraintes SSA permet en général de découvrir de la concurrence exploitable lors de la compilation et de l'exécution.

L'incarnation la plus courante du principe SSA est le formalisme SSA introduit en Section 3.2. Cependant, d'autres formalismes satisfont le principe SSA, notamment des langages flot de données utilisés en conception de systèmes RTE, tels Lustre/Scade ou Simulink. Cela facilitera notre travail de rapprochement HPC-RTE en Chapitres 4 et 5.

3.2 Le formalisme SSA : syntaxe et sémantique intuitive

Une spécification en formalisme SSA est un graphe de flot de contrôle séquentiel (SCFG, *Sequential Control-Flow Graph*) dont les nœuds sont des fragments de code appelés *basic blocks* (BB). Chaque BB contient lui-même une séquence d'*opérations*.

Conçu pour spécifier des graphes, le formalisme SSA a naturellement été introduit sous la forme d'une représentation graphique, mais sa manipulation en tant qu'IR a nécessité qu'on lui donne une traduction textuelle. Dans cette section, nous employons tout à la fois la syntaxe graphique traditionnelle et une syntaxe textuelle inspirée de celle de MLIR, définie formellement en Fig. 3.3. Notre syntaxe simplifie celle de MLIR pour faciliter la définition de la sémantique formelle en Section 3.3[2].

Nous illustrons ces deux représentations possibles d'un programme SSA en Fig. 3.2 et les expliquons ci-dessous : la fonction factorielle qui y est définie nous permettra de donner l'intuition des définitions tout au long de cette section.

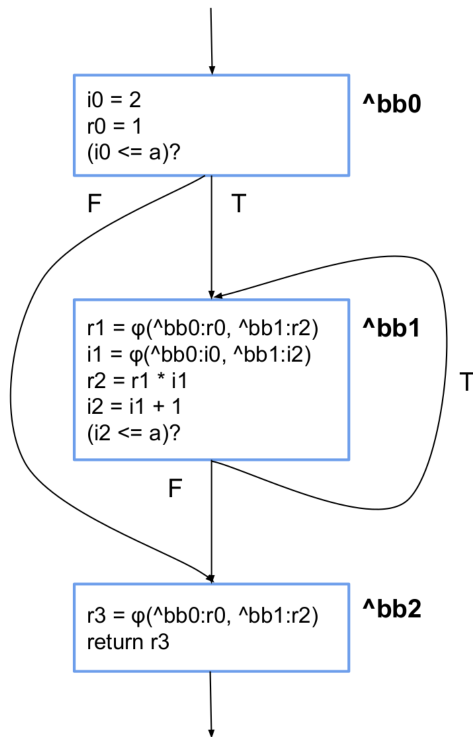
3.2.1 Graphe de flot de contrôle séquentiel

Formellement, une spécification en formalisme SSA est un SCFG G = (B, T, b_0), où :

- B est l'ensemble fini des BBs (*basic blocks*) du graphe. Un BB contient une séquence d'opérations de calcul qui définissent et/ou utilisent des

[2] Au-delà de ces exigences de définition formelle, à partir de Section 3.4, nous recourrons directement à la syntaxe de MLIR sans variations.

```
1  int factorial(int a) {
2    int r = 1;
3    for (int i = 2; i <= a; i++) {
4      r *= i;
5    }
6    return r;
7  }
```




```
1  func @factorial(i32) -> (i32)
2  {
3  ^bb0(a: i32)
4    i0 = constant 2
5    r0 = constant 1
6    c0 = (i0 <= a)
7    cond_br c0, ^bb1(r0, i0),
8                ^bb2(r0)
9
10 ^bb1(r1, i1):
11   r2 = r1 * i1
12   i2 = i1 + 1
13   c1 = (i2 <= a)
14   cond_br c1, ^bb1(r2, i2),
15               ^bb2(r2)
16
17 ^bb2(r3):
18   return r3
19 }
```

FIGURE 3.2 : Une fonction factorielle écrite en langage C (en haut à gauche), sa représentation SSA graphique (en bas à gauche) où on suppose définie la variable a, et sa représentation textuelle (en bas à droite).

variables.

L'opération terminant un BB doit être un *terminateur*, c'est-à-dire :

- Soit une opération de branchement qui connecte le BB à un autre BB. Dans Fig. 3.2, bb0 et bb1 se terminent par des opérations de branchement conditionnel, choisissant – en fonction de la valeur d'une variable booléenne – le BB qui recevra le contrôle.
- Soit une opération de fin d'exécution, rendant le contrôle à l'environnement du SCFG. Dans Fig. 3.2, bb2 se termine ainsi par une opération `return` qui, dans le contexte d'une fonction, rend le contrôle au code appelant.

- T est l'ensemble fini des transitions du graphe, qui définissent les passages de contrôle possibles entre BBs. Ces transitions sont déterminées par les opérations de branchement conditionnel ou non-conditionnel servant de terminateurs aux *basic blocks*. Pour chaque transition t $\in$ T, on peut identifier son BB source src(t) $\in$ B et son BB destination dest(t) $\in$ B, mais la source et la destination n'identifient pas de manière unique une transition [3].
- $b_0 \in$ B est le BB par où le contrôle entre dans le graphe.

3.2.2 L'opérateur ϕ

Si les BBs peuvent contenir des opérations de calcul et des terminateurs, on voit en Fig. 3.2 qu'ils peuvent aussi contenir des opérateurs ϕ. Spécifiques au formalisme SSA, ces derniers permettent à un BB destinataire de transitions multiples de construire une valeur unique à partir des différentes sources.

Dès lors, dans tout BB, chaque opérateur ϕ demande autant de paramètres que le BB n'a de transitions entrantes. C'est notamment le cas, en Fig. 3.2, de bb1 : l'opérateur ϕ construisant la variable r1 et celui construisant la variable i1 exigent chacun deux paramètres, correspondant aux arcs entrants depuis bb0 et bb1. Ces paramètres sont identifiés de manière unique par une paire : le nom de la variable fournie en paramètre, et le BB source dans lequel elle est définie.

[3] On peut avoir deux transitions avec la même source et la même destination, quand les deux branches d'une opération de branchement conditionnel ont la même destination.

Le contrôle arrive toujours dans un BB par (exactement) un arc ; du fait de cette *séquentialité* du SCFG, la valeur de retour de tout opérateur ϕ est définie de manière unique. Ce mécanisme garantit le déterminisme des fonctions SSA.

Dans la représentation SSA textuelle de Fig. 3.2, les opérateurs ϕ sont remplacés par le plus intuitif *Continuation Passing Style* (CPS) [4], dont la syntaxe est proche de l'appel de fonctions. Ainsi, on déclare les valeurs construites par les opérateurs ϕ d'un BB directement dans l'en-tête de ce dernier. En Fig. 3.2, le BB bb1 sous forme textuelle (à droite) déclare ainsi deux arguments, correspondant aux deux opérateurs ϕ de bb1 sous forme graphique (à gauche). Les valeurs correspondant aux transitions d'entrée sont représentées dans les opérations de branchement des BBs source. Ainsi, quand l'opération `cond_br` en lignes 7 et 8 donne le contrôle à bb1, r1 prend la valeur de r0, et i1 prend la valeur de i0.

Le BB d'entrée du graphe est un cas particulier de ce mécanisme car il doit déclarer les arguments qu'il reçoit avec le contrôle depuis l'extérieur quand le contrôle est transmis par appel de fonction. En forme textuelle, on fait l'hypothèse que b_0 est toujours le premier BB de la fonction. Les arguments de ce BB doivent correspondre en nombre et en type aux arguments d'entrée spécifiés en signature de la fonction.

3.2.3 La syntaxe et ses extensions

La syntaxe SSA que nous utilisons dans ce chapitre pour les développements formels est présentée en Fig. 3.3. Nous y distinguons 3 couches : le noyau SSA (en noir), la modularité (en rouge) et la mémoire persistante (en vert). Dans cette section, nous présentons les deux extensions – la modularité fondée sur les appels de fonction et l'accès à la mémoire persistante –.

La définition et l'appel de fonctions

La définition et l'appel de fonctions introduisent modularité et hiérarchie dans les SCFGs. Une définition de fonction SSA consiste en :

1. Son interface, déclarant son nom et sa structure (paramètres d'entrée et types de sortie). En Fig. 3.2 (à droite), l'interface de la fonction est en ligne 1.

[4] La correspondance entre SSA et CPS est bien connue[59].

```
<ssa_spec> ::= <function>+
<function> ::= func <fun_name> <fun_iface> { <fun_body> }
<fun_iface> ::= (<type>*)->(<type>*)
<fun_body> ::= <block>+
<block> ::= <blk_arg>:<blk_body>
<blk_arg> ::= <block_name>(<tvar>*)
<tvar> ::= <var>:<type>
<blk_body> ::= <op>* <term_op>
<op> ::= (<var>*)=<op_id>(<tvar>*):<type>*
       | <var> = load(<tvar>):<type>
       | store(<tvar>,<tvar>)
<term_op> ::= cond_br <var> <blk_arg> <blk_arg>
       | br <blk_arg> | return(<tvar>*)
<op_id> ::= <arith_op> | <bool_op> | call <fun_name>
```

FIGURE 3.3 : Syntaxe SSA

2. Son corps (lignes 2 à 20 en Fig. 3.2), qui décrit le SCFG. Les conventions liées à la définition du BB servant de point d'entrée à la fonction ont été discutées plus haut. Les arguments des opérations `return` dans ses BBs de sortie (ligne 18 en Fig. 3.2) donnent les types de retour de la fonction (ligne 1 en Fig. 3.2).

L'opération `call` permet d'appeler une fonction depuis le corps d'une autre fonction.

Les accès à la mémoire persistante

Les accès à la mémoire persistante supposent l'existence d'une mémoire adressable avec des pointeurs (des valeurs de type entier). Les opérations arithmétiques sur entiers permettent donc de faire l'arithmétique des pointeurs, et nous avons besoin de deux nouvelles opérations pour stocker et pour charger les valeurs en mémoire :

- L'opération `store` écrit la valeur d'une variable (son premier argument) à l'adresse indiquée par une autre variable (son second argument, le pointeur).

- L'opération `load` produit une nouvelle variable, qui prend la valeur stockée dans la zone mémoire référencée par le pointeur en argument.

Cette extension est illustrée dans l'exemple en Fig. 3.4, qui réimplémente la fonction factorielle présentée en Fig. 3.2. À chaque itération, le pointeur r est déréférencé (ligne 13), multiplié avec le terme suivant de la factorielle (ligne 14), puis le résultat est réécrit sur la zone mémoire pointée (ligne 15).

Signalons que l'accès à la mémoire persistante, par construction, contourne le principe SSA, lequel s'applique seulement aux variables (et non aux zones mémoire éventuellement pointées par ces dernières). On peut tout à la fois, au moyens de pointeurs :

- Lire une zone mémoire qui n'a pas été initialisée.
- Réécrire plusieurs fois une même zone mémoire.

Assurer un ordonnancement consistant de ces opérations, et donc le déterminisme, demande un traitement particulier des accès mémoire, que l'analyse de dominance spécifique en SSA (et définie en Section 3.2.4) ne couvre pas.

3.2.4 Propriétés de correction

Comme dans la plupart des langages de programmation, la seule correction syntaxique ne suffit pas à garantir qu'une spécification SSA est bien formée. Nous illustrons cela en Fig. 3.5, où nous reprenons la fonction factorielle de Fig. 3.2 en y introduisant une malformation : la variable r2 retournée à la fin de bb2 n'est pas définie si le contrôle arrive de bb0, car elle n'est jamais initialisée (r2 n'est plus la sortie d'un opérateur ϕ de bb2).

Pour assurer qu'une spécification est bien formée, nous devons aussi exiger le respect de propriétés de correction sémantiques. En formalisme SSA, ces dernières ont la particularité de pouvoir être vérifiées structurellement par une algorithmique de graphe de faible complexité. Un SCFG G = (B, T, b_0) doit ainsi vérifier deux propriétés suivantes :

1. **L'affectation unique.** Chaque variable doit être soit définie dans l'entête d'un BB, soit définie par une seule opération d'un seul BB.
2. **La dominance.** Cette propriété, vérifiée par *l'analyse de dominance*, exige que, pour chaque opération o2 du *basic block* b2 utilisant la variable v1 produite dans le *basic block* b1 :

```
1  func @factorial(arg: i32) -> (i32) {
2
3  ^bb0(a: i32)
4    i0 = constant 2
5    r = constant 0x8000
6    one = constant 1
7    store(r, one)
8    c0 = (i0<=a)?
9    cond_br c0, ^bb1(i0),
10               ^bb2
11
12 ^bb1(i1):
13   r_tmp = load(r)
14   r_new = r_tmp * i1
15   store(r, r_new)
16   i2 = i1 + 1
17   c1 = (i2 <= a)
18   cond_br c1, ^bb1(r2),
19               ^bb2
20
21 ^bb2:
22   r_end = load(r)
23   return r_end
24
25 }
```

FIGURE 3.4 : La spécification SSA d'une factorielle accumulant le résultat dans la zone mémoire référencée par le pointeur r. Les opérations manipulant la mémoire persistante sont surlignées en vert.

```
1  func @factorial(arg: i32) -> (int)
2
3  ^bb0(a: i32)
4    i0 = constant 2
5    r0 = constant 1
6    c0 = (i0 <= a)
7    cond_br c0, ^bb1(r0, i0),
8                ^bb2
9
10 ^bb1(r1, i1):
11   r2 = r1 * i1
12   i2 = i1 + 1
13   c1 = (i2 <= a)
14   cond_br c1, ^bb1(r2, i2),
15               ^bb2
16
17 ^bb2:
18   return r2
19
20 }
```

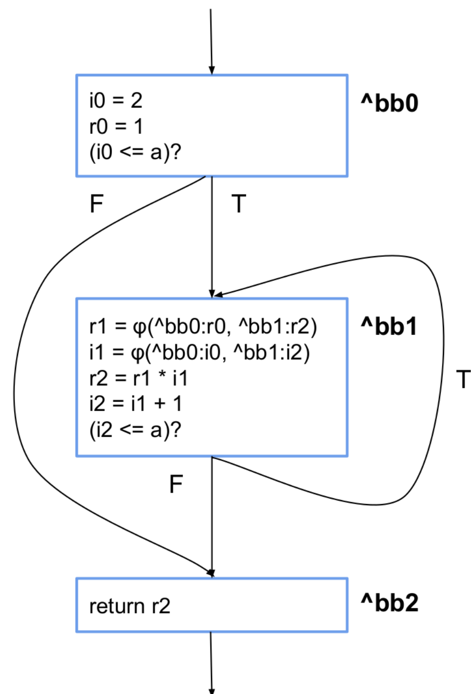


FIGURE 3.5 : La spécification SSA d'une factorielle (forme textuelle à gauche et forme graphique à droite), syntaxiquement correcte mais ne respectant pas le principe SSA : si le contrôle arrive à bb2 depuis bb0, la variable r2 n'est pas définie. La vérification de la *dominance* rejette ce programme.

- Si b1 = b2, alors la définition de v1 doit précéder o2 dans b2. On exige donc des opérations internes aux *basic blocks* qu'elles soient triées suivant l'ordre topologique induit par la production et l'utilisation des variables.
- Si b1 $\neq$ b2, alors b1 doit *dominer* b2.

  Pour caractériser la relation de dominance entre deux *basic blocks*, nous devons définir la notion de *chemin* entre deux BBs. Un chemin p allant de $b_1 \in B$ à $b_2 \in B$ est une suite finie de transitions $p = t_0, t_1, ..., t_k \subset T$ telle que $src(t_0) = b_1, dest(t_k) = b_2$ et pour $0 \leq i \leq k$, $dest(t_i) = src(t_{i+1})$. On dit qu'un chemin p passe par $b_3 \in B$ si et seulement si il existe $t_3 \in p$ tel que $b_3 = dest(t_3)$.

  Dès lors, dans un SCFG G = (B, T, b_0), si $b_1, b_2 \in B$, on dit que b_1 domine b_2 si tout chemin entre b_0 et b_2 contient b_1.

Le critère de dominance est souvent reformulé au moyen de la *frontière de dominance*, qui caractérise la *lifetime* des variables.

Considérons les notions de :

- *Prédécesseur immédiat.* Le BB $b_1 \in B$ est un prédécesseur immédat de $b_2 \in B$ si et seulement si il existe une transition $t \in T$ telle que $src(t) = b_1$ et $dest(t) = b_2$.
- *Successeur.* Un BB b_3 est un successeur de b_2 s'il existe un chemin de b_2 à b_3.

Dans un SCFG G = (B, T, b_0), la frontière de dominance de $b \in B$ est l'ensemble des BBs b_i tels que b domine un prédécesseur immédiat de b_i mais ne domine pas b_i. La Fig. 3.6 illustre cette notion : b4 et b5 y forment la frontière de dominance de b1, car, contrairement à leurs prédécesseurs immédiats, ils ne sont pas dominés par b1 (on peut atteindre b4 directement par b0, et b5 par b4).

Dès lors, le respect du critère de dominance revient à exiger qu'une variable v définie dans $b \in B$ ne soit utilisée ni dans les BBs constituant la frontière de dominance de b, ni dans aucun de leurs successeurs.

Accessibilité. Les premières définitions de SSA exigeaient l'absence de code mort. Formellement, elles exigeaient d'un SCFG G = (B, T, b_0) que, pour tout $b \in B$, il existe $p \subset T$ tel que p est un chemin allant de b_0 à b.

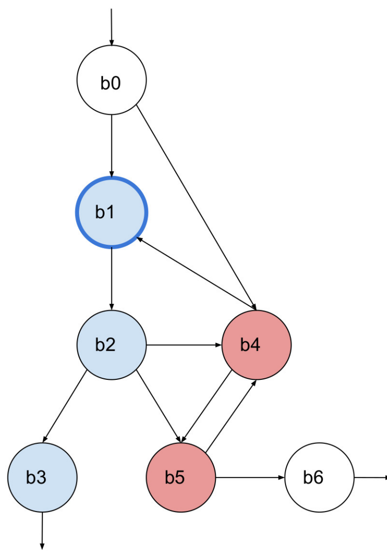


Figure 3.6 : Les relations de dominance d'un BB b1 au sein d'un graphe SSA. Les BBs dominés par b1 sont en bleu. La frontière de dominance de b1 est en rouge.

Cette propriété, n'étant pas vérifiable exactement, est en réalité approximée : par exemple, MLIR vérifie que pour tout $b \in B \setminus \{b_0\}$, il existe $t \in T$ tel que $dest(t) = b$, même si un tel chemin est infaisable du fait des branchements conditionnels impliqués.

Concurrence. SSA est un formalisme globalement séquentiel, du fait de son organisation en graphe flot de contrôle séquentiel. SSA est en même temps un formalisme localement concurrent : à l'intérieur de chaque *basic block*, les opérations peuvent être réordonnées du moment que l'ordre partiel entre opérations déterminé par la dominance est respecté. Cette forme de concurrence sert ainsi à déterminer quelles opérations peuvent être exécutées en même temps sur une architecture parallèle, ou quelles opérations peuvent commuter lors des transformations de code (e.g. optimisations). Ceci est d'ailleurs le point de vue que nous avons adopté pour définir la sémantique formelle en Section 3.3.

Lorsque la mémoire est utilisée, le déterminisme d'une spécification SSA nécessite de limiter encore plus la concurrence pour éviter les *data races* de type *read-write* ou *write-write*, et de compléter l'analyse de dominance par d'autres analyses assurant qu'une zone mémoire n'est jamais lue sans avoir été écrite (i.e. qu'aucune opération ne manipule des valeurs non-définies).

3.3 Sémantique formelle

Par souci de concision et sans affecter la généralité des résultats, nous supposons que toutes les variables (y compris les booléens) sont représentées par des entiers. Nous supposons également que chaque opération du programme est étiquetée de manière unique, par exemple au moyen de la ligne où elle apparaît (en supposant que deux opérations ne partagent jamais la même ligne).

Notations Le cardinal d'un ensemble $\mathcal{S}$ est dénoté $|\mathcal{S}|$. Nous utilisons la notation OCaml pour les listes : [] est la liste vide, $h :: t$ la liste dont le premier élément est h et la queue t.

Int est le domaine des entiers. Pour représenter la valeur d'une zone mémoire ou d'une variable non initialisée, nous utilisons la valeur spéciale "indéfini" dénotée $\perp$. Par conséquent, la valeur de toute variable ou zone mémoire, à n'importe quel point de l'exécution, est un élément de $\overline{Int} = Int \cup \{\perp\}$.

Par abus de langage, nous dénotons aussi $\bot$ toute fonction qui retourne toujours $\bot$, quel que soit son domaine. Étant donnée une fonction mathématique $f : A \to B$ et $x \in A$, $y \in B$, nous définissons une nouvelle fonction $f[x \leftarrow y] : A \to B$ définie par :

$$f[x \leftarrow y](z) = \begin{cases} y & \text{si } z = x \\ f(z) & \text{sinon.} \end{cases}$$

Par définition, $\mathcal{L}^{\mathtt{f}}$ et $\mathcal{V}^{\mathtt{f}}$ sont respectivement les ensembles d'étiquettes et de variables d'une fonction SSA `f`, et $b_0^{\mathtt{f}}$ est son *basic block* d'entrée. Nous supposons que les BBs d'une spécification sont identifiés de manière unique, et nous notons avec $fun(b)$ la fonction SSA contenant le BB b. Suivant la syntaxe type MLIR, chaque BB b a une liste d'arguments, que nous modélisons par un ensemble ordonné $in(b) \subseteq \mathcal{V}^{fun(\mathtt{b})}$, où $in(b)_i$ est la i^{e} entrée de b. L'ensemble ordonné des variables affectées par les opérations d'un BB b est dénoté $loc(b)$. L'étiquette de la première opération dans le BB b est notée $fst(b)$. L'opération associée à l'étiquette $l \in \mathcal{L}^{\mathtt{f}}$ est dénotée $op(l)$. Si $op(l)$ n'est pas une opération terminale, alors $nxt(l)$ est le label de l'opération suivante dans le BB auquel elle appartient. Le nombre d'arguments d'une opération `return` dans une fonction `f` est dénoté O^f.

État d'exécution Dans notre sémantique, l'état d'exécution d'une *fonction* SSA `f` est soit :

- Un état initial $Start^{\mathtt{f}}(v_1, \ldots, v_{|in(b_0^{\mathtt{f}})|})$, où les valeurs $v_i \in \overline{Int}$ sont les paramètres de la fonction.
- Un état final $End^{\mathtt{f}}(w_1, \ldots, w_{O^f})$, où les valeurs $w_i \in \overline{Int}$ sont les sorties de `f` (et les entrées de `return`).
- Un état intermédiaire $Run^{\mathtt{f}}(pc, val)$ formé de l'étiquette pc de l'opération à exécuter (le compteur de programme), et d'une valuation partielle $val : \mathcal{V}^{\mathtt{f}} \to \overline{Int}$ de toutes les variables de `f`.

L'état d'exécution d'une *spécification* SSA comprenant une ou plusieurs fonctions pouvant s'appeller les unes les autres est un triplet (s, cs, m) formé de l'état s de la fonction en cours d'exécution, d'une liste cs d'états intermédiaires représentant la pile d'appel, et de l'état courant $m : Int \to \overline{Int}$ de la mémoire.

Un *état initial* d'une spécification est a la forme $(Start^{\mathtt{f}}(\ldots), [], m)$, où m est l'état initial de la mémoire et `f` la fonction qui sert de point d'entrée à l'exécution. L'*état final* d'une spécification a la forme $(End^{\mathtt{f}}(\ldots), [], m)$.

Exécution d'un programme Les règles de transition sont fournies en Fig. 3.7. Une *trace d'exécution* d'une spécification SSA est une séquence quelconque de transitions partant d'un état initial. Notons que si $t = s_0 \to s_1 \to \ldots \to s_n \to \ldots$ est une trace, alors tout préfixe $t_n = s_0 \to s_1 \to \ldots \to s_n$ est aussi une trace. Nous dénotons avec $\leq$ l'ordre préfixe sur les traces, ce qui signifie que pour t et t_n définis pus haut, nous pouvons écrire $t_n \leq t$.

Notons également que les règles **(ifthen)** et **(ifelse)** définissent le comportement d'un branchement conditionnel seulement quand la variable testée est définie. Lorsque sa valeur est $\perp$, l'exécution ne peut pas avancer – elle bloque. L'exécution bloque également quand une opération `load` ou `store` accède à une adresse qui est soit $\perp$, soit invalide. D'après l'interprétation de l'indéfinition par LLVM[64], cela revient à supposer que l'utilisation de $\perp$ dans les décisions et dans les calculs constitue un comportement indéfini immédiat (*immediate undefined behavior*). Pour rejoindre cette interprétation, nous supposons également que la sémantique $[\![op_id]\!]$ de chaque opération est une fonction partielle, indéfinie lorsqu'elle a des entrées valant $\perp$.

Les traces maximales d'une spécification SSA (au sens de $\leq$) sont donc soit infinies, soit finissent dans un état final, soit finissent parce que l'exécution bloque.

Hypothèses de séparation Notre sémantique sépare, dans la spécification de l'état et dans les règles de transition, le fragment correspondant au noyau SSA des extensions nécessaires pour représenter les appels de fonction et la mémoire persistante. Ainsi, pour ne considérer que la sémantique du noyau SSA, il suffit de supprimer de la spécification de l'état la pile d'appels et les termes décrivant la mémoire, et de considérer seulement les règles en noir dans Fig. 3.7 (qui n'accèdent ni à la mémoire ni à la pile d'appels). De même, les règles pour l'appel de fonction n'accèdent pas à la mémoire, et les règles pour la mémoire n'accèdent pas à la pile d'appels.

Déterminisme et correction Si nous supposons que toutes les opérations sont déterministes (ce qui, dans notre dispositif formel, revient à supposer que toute $[\![op_id]\!]$ est une fonction partielle), la séquentialité de l'exécution

$$(\mathit{Start}^{\mathtt{f}}(v_1,\ldots,v_{|in(b_0^{\mathtt{f}})|}), cs, m) \to (\mathit{Run}^{\mathtt{f}}(\mathit{fst}(b_0^{\mathtt{f}}), \bot[in(b_0^{\mathtt{f}})_k \leftarrow v_k \mid 1 \le k \le |in(b_0^{\mathtt{f}})|]), cs, mem) \ \textbf{(start)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = op_id(w_1,\ldots,w_m)\text{“} \qquad (o_1,\ldots,o_n) = [\![op_id]\!](s(w_1)\ldots,s(w_m))}{(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to (\mathit{Run}^{\mathtt{f}}(nxt(l), s[v_i \leftarrow o_i \mid 1 \le i \le n]), cs, m)} \ \textbf{(opcall)}$$

$$\frac{op(l) = \text{“}\mathtt{br}\ bb(v_1,\ldots,v_{|in(bb)|})\text{“}}{\begin{array}{l}(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to \\ \quad (\mathit{Run}^{\mathtt{f}}(\mathit{fst}(bb), s[in(bb)_i \leftarrow s(v_i) \mid 1 \le i \le |in(bb)|][loc(bb)_i \leftarrow \bot \mid 1 \le i \le |loc(bb)|]), cs, m)\end{array}} \ \textbf{(goto)}$$

$$\frac{op(l) = \text{“}\mathtt{cond_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v) \notin \{0,\bot\}}{\begin{array}{l}(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to \\ \quad (\mathit{Run}^{\mathtt{f}}(\mathit{fst}(bb1), s[in(bb1)_i \leftarrow s(v_i) \mid 1 \le i \le |in(bb1)|][loc(bb1)_i \leftarrow \bot \mid 1 \le i \le |loc(bb1)|]), cs, m)\end{array}} \ \textbf{(ifthen)}$$

$$\frac{op(l) = \text{“}\mathtt{cond_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v) = 0}{\begin{array}{l}(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to \\ \quad (\mathit{Run}^{\mathtt{f}}(\mathit{fst}(bb2), s[in(bb2)_i \leftarrow s(w_i) \mid 1 \le i \le |in(bb2)|][loc(bb2)_i \leftarrow \bot \mid 1 \le i \le |loc(bb2)|]), cs, m)\end{array}} \ \textbf{(ifelse)}$$

$$\frac{op(l) = \text{“}\mathtt{return}(v_1,\ldots,v_{O^f})\text{“}}{(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to (\mathit{End}^{\mathtt{f}}(s(v_1),\ldots,s(v_{O^f})), cs, m)} \ \textbf{(end)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to (\mathit{Start}^{\mathtt{g}}(s(w_1),\ldots,s(w_k)), \mathit{Run}^{\mathtt{f}}(l,s)::cs, m)} \ \textbf{(call)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(\mathit{End}^{\mathtt{g}}(x_1,\ldots,x_n), \mathit{Run}^{\mathtt{f}}(l,s)::cs, m) \to (\mathit{Run}^{\mathtt{f}}(nxt(l), s[v_i \leftarrow x_i \mid 1 \le i \le n]), cs, m)} \ \textbf{(ret)}$$

$$\frac{op(l) = \text{“}w = \mathtt{load}(addr)\text{“} \qquad s(addr) \ne \bot \qquad valid_addr(s(addr))}{(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to (\mathit{Run}^{\mathtt{f}}(nxt(l), s[w \leftarrow m(s(addr))]), cs, m)} \ \textbf{(load)}$$

$$\frac{op(l) = \text{“}\mathtt{store}(addr, v)\text{“} \qquad s(addr) \ne \bot \qquad valid_addr(s(addr))}{(\mathit{Run}^{\mathtt{f}}(l,s), cs, m) \to (\mathit{Run}^{\mathtt{f}}(nxt(l), s), cs, m[s(addr) \leftarrow s(v)])} \ \textbf{(store)}$$

FIGURE 3.7 : Règles de transition de la sémantique SSA. Règles du noyau SSA en noir. Règles liées aux appels de fonction en rouge. Règles liées à la mémoire persistante en vert.

SSA implique son déterminisme : pour tout état initial s, il existe une trace unique partant de s qui est maximale au sens de $\leq$.

Mais le déterminisme ne suffit pas. La correction nécessite aussi que l'exécution ne bloque jamais. Cette propriété est garantie au moyen d'une propriété plus contraignante, exigeant que toutes les variables soient différentes de $\perp$ lorsqu'elles sont utilisées dans les décisions, les accès mémoire et les calculs. Supposons :

- Que la variable v soit définie par le BB b (soit par une opération de b, soit par son en-tête) ;
- Que v sert d'entrée à une opération o' de b'.

Alors, pour garantir que v n'a pas la valeur $\perp$ quand o' est exécutée, on doit garantir que toute exécution atteignant o' traverse la définition de v. Cela est le cas *si et seulement si* l'une des conditions suivantes est vraie :

D1 $b = b'$, et la défintion de v précède o' dans b, soit comme argument du BB, soit comme sortie d'une opération.

D2 $b \neq b'$, et toute exécution possible atteingant b' traverse nécessairement b.

Si la vérification de **D1** est triviale, montrer que **D2** est vraie pour toute variable v et toute opération o utilisant v n'est pas possible dans le cas général (le problème de la satisfaisabilité booléenne peut être réduit à la vérification de **D2**). Pour cette raison, la compilation basée sur SSA garantit **D2** au moyen d'une propriété suffisante, le critère de dominance que nous avons présenté précédemment.

Considérés ensemble, le critère de dominance, la correction syntaxique et les propriétés structurelles définies en Section 3.2 suffisent à garantir la correction des spécifications SSA qui n'accèdent pas à la mémoire. Lorsque la mémoire est utilisée, ces propriétés doivent être complétées d'une preuve montrant que chaque zone mémoire est initialisée avant d'être lue.

3.4 MLIR : Multi Level Intermediate Representation

Nous avons vu précédemment comment le formalisme SSA s'est enrichi, au fil de ses évolutions, avec des accès à la mémoire persistante et avec une

modularité fondée sur les appels de fonction.

MLIR raffine ce formalisme de base au moyen de constructions facilitant l'écriture et la lecture du code :

- La représentation en *Continuation Passing Style* (CPS) des opérateurs ϕ.
- Un système de types adapté à la compilation ML, représentant des données agrégées sous la forme de tenseurs ou de tableaux alloués en mémoire (nommés en jargon MLIR *memrefs*), autorisant une forme limitée de polymorphisme sur ces types agrégés.
- Un système d'*attributs* représentant des informations statiques (sous la forme de paires clé/valeur) qui paramètrent les opérations. Un attribut peut être attaché à la définition d'une opération, auquel cas il est généralement obligatoire, ou bien il peut être générique, auquel cas il est généralement optionnel. Dans la première catégorie, on peut citer le prédicat associé à une opération de comparaison (à choisir parmi $>$, $\geq$, $<$ ou $\leq$). Dans la deuxième catégorie, on peut citer l'attribut permettant de spécifier la cible matérielle sur laquelle une opération doit être exécutée.

Mais au-delà de ces extensions classiques dans le monde des représentations intermédiaires, MLIR apporte deux extensions spécifiques : les régions et les dialectes.

3.4.1 Les régions

La plupart des mises en œuvre de SSA introduisent un premier mécanisme permettant une spécification modulaire – les fonctions et les appels de fonction, au moyen desquels un graphe appelé peut recevoir le contrôle d'un graphe appelant et le lui rendre. MLIR généralise ce principe avec les régions.

Une forme très générale de hiérarchie

Bien plus qu'une forme particulière de modularité, les régions permettent une forme très générale et extensible de spécification hiérarchique.

Une région est un SCFG associé à une opération particulière. Par exemple, la définition d'une fonction est une opération `func` ayant un attribut définissant son nom, et une région qui définit son corps. De façon similaire, une opération de flot de contrôle structuré `if` a deux régions, une correspondant à la branche `then`, l'autre à la branche `else`.

Les régions peuvent être très profondément imbriquées les unes dans les autres : en effet, l'opération à laquelle est attachée une région peut elle-même appartenir à un BB, lequel appartient à une région, laquelle est attachée à une opération, etc. Cette imbrication des régions les unes dans les autres est illustrée en Fig. 3.8. On y a transformé la fonction factorielle présentée en Fig. 3.2 pour illustrer l'insertion d'une boucle `for` (et de sa région associée) dans le corps d'une fonction (i.e. la région associée à la définition d'une fonction). Par définition, la seule opération qui peut être placée à *top-level* d'un programme, hors de toute région (et de tout BB) est l'opération `module`. Un module contient une unique région, laquelle contient un unique BB qui peut à son tour contenir toute opération (en général, des fonctions et des variables globales) et n'a pas besoin d'opération terminale. Un programme MLIR est un ensemble de modules, bien que par défaut, l'opérateur `module` encapsulant toutes les opérations d'un fichier soit implicite. On peut néanmoins l'écrire explicitement, comme en Fig. 3.9.

Une fonction sans région (en ligne 4) est la déclaration d'une fonction externe.

MLIR autorise à ne pas nommer le BB d'entrée d'un graphe. Les premières opérations d'une région donnée y appartiennent implicitement.

Suivant l'opération à laquelle une région est attachée, l'opération qui la termine n'est pas nécessairement `return`. S'agissant de `for`, son opération terminale est `yield`.

Certaines opérations ne déclarent pas de région (c'est le cas par exemple d'une somme entière), et d'autres en déclarent plusieurs (c'est le cas de l'opération `if` qui déclare deux régions, une pour chacune de ses branches *then* et *else*).

Relaxation de la dominance

Les régions sont aussi le support d'un mécanisme permettant de relaxer les règles de correction de SSA. Les régions où ce mécanisme est activé sont soumises à toutes les vérifications de correction SSA (e.g. affectation unique) sauf l'analyse de dominance. Cela permet notamment la représentation de

```
func @factorial(%a:i32) -> (i32) {
  %lb = arith.constant 2: i32
  %step = arith.constant 1: i32
  %r0 = arith.constant 1: i32

  %r2 = scf.for %i = %lb to %a step %step
    iter_args(%r = %r0)->(i32) {

    %r1 = arith.muli %r,%i: i32

    scf.yield %r1: i32
  }

  return %r2: i32
}
```

FIGURE 3.8 : La région interne à une boucle for imbriquée dans la région interne à la définition d'une fonction (factorielle).

systèmes de dépendances cycliques ou non-ordonnées.

Les régions ainsi spécifiées permettent donc l'importation de spécifications qui ne sont pas encore en forme SSA. Ces régions ne peuvent pas faire l'objet de l'algorithmique SSA existante (analyses de correction, optimisations, étapes de *lowering*). Elles sont donc conçues pour être utilisées au début d'un processus de compilation, mais doivent rapidement être normalisées pour retrouver la compatibilité avec SSA. Nous illustrons ce mécanisme en Chapitre 5, où nous représentons (puis normalisons) des spécifications Lustre dans MLIR.

MLIR fournit à cet égard une fonction de normalisation limitée, permettant de recalculer automatiquement l'ordre topologique des opérations au sein d'un *basic block* dès lors que le système des dépendances n'est pas cyclique, tel qu'illustré Fig. 3.10.

3.4.2 Les dialectes

Les compilateurs HPC sont, la plupart du temps, conçus pour procéder par passes de compilation incrémentales (*lowering*) préservant la sémantique fonctionnelle du programme source : une spécification de haut niveau est incrémentalement réécrite sous la forme de représentations de plus bas niveau, jusqu'à atteindre le niveau de l'implémentation (code objet, bytecode de machine virtuelle, etc.).

```
module {

  func @print(i32) -> ()

  func @factorial(%a: i32) -> (i32) {
    %lb = arith.constant 2: i32
    %step = arith.constant 1: i32
    %r0 = arith.constant 1: i32

    %r2 = scf.for %i = %lb to %a step %step
      iter_args(%r = %r0)->(i32) {

      %r1 = arith.muli %r,%i: i32

      scf.yield %r1: i32
    }

    return %r2: i32
  }

  func @main() -> (i32)
    %zero = arith.constant 0: i32
    %ten = arith.constant 10: i32
    %f = call @factorial(%ten): (i32) -> (i32)
    call @print(%f): (i32) -> ()
    return
  }

}
```

FIGURE 3.9 : Un module MLIR embarquant la déclaration d'une fonction (externe) d'affichage, la définition d'une fonction "factorielle", et une fonction *main* appelant l'une et l'autre.

```
1 func @f(%i: i32) -> (i32) {
2
3   %r = arith.addi %i,%j:i32
4   %j = arith.constant 1:i32
5
6   return %r: i32
7 }
```

```
1 func @f(%i: i32) -> (i32) {
2
3   %j = arith.constant 1: i32
4   %r = arith.addi %i,%j: i32
5
6   return %r:i32
7 }
```

```
1 func @g(%k: i32) -> (i32) {
2
3   %i = op1(%j):i32
4   %j = op2(%i):i32
5   %r = addi %j,%k:i32
6
7   return %r:i32
8 }
```

FIGURE 3.10 : Une spécification où la dominance est relaxée (haut) peut être normalisée en recalculant l'ordre topologique des opérations (milieu), mais c'est impossible quand il y a des dépendances cycliques sur les opérations (bas). .

```
func @f() -> () {                              // dialecte std
  affine.for %i = 0 to 100 {                   // dialecte affine
    "mydialect.foo"() : () -> ()               // dialecte mydialect
    %v = scf.execute_region -> i64 {           // dialecte scf
      cond_br %cond, ^bb1, ^bb2                // dialecte std

    ^bb1:
      %c1 = arith.constant 1 : i64             // dialecte arith
      br ^bb3(%c1 : i64)                       // dialecte std

    ^bb2:
      %c2 = arith.constant 2 : i64             // dialecte arith
      br ^bb3(%c2 : i64)                       // dialecte std

    ^bb3(%x : i64):
      scf.yield %x : i64                       // dialecte scf
    }
    "mydialect.bar"(%v) : (i64) -> ()          // dialecte mydialect
  }
  return                                       // dialecte std
}
```

FIGURE 3.11 : Différents niveaux d'abstraction coexistant dans une même fonction MLIR. Le dialecte de chaque opération est indiqué en commentaire.

MLIR est entièrement structuré autour de ce principe : les opérations, attributs et types de données correspondant à un niveau d'abstraction particulier sont agrégés dans un dialecte, qui est un sous-ensemble de l'IR spécifique à un domaine (*domain-specific*). Ainsi, les opérations sur les nids de boucles affines sont agrégées dans un dialecte **affine**, les opérations et types de données utilisés lors de la vectorisation des calculs sont agrégés dans un dialecte **vector**, etc. Le dialecte auquel appartient une opération ou un type est indiqué par son préfixe. Par exemple, l'opération `affine.for` est un niveau de boucle affine.

Différents dialectes peuvent cœxister dans une même spécification MLIR, comme en Fig. 3.11, où des opérations issues du cœur SSA (groupées dans le dialecte **std**, dont le préfixe est implicite) coexistent avec des opérations de flot de contrôle structuré (dialecte **scf**), avec des opérations sur les boucles affines (dialecte **affine**), avec des opérations arithmétiques (dialecte **arith**) et avec les opérations d'un dialecte **mydialect** étendant les dialectes standard de MLIR.

La sémantique des opérations composant un dialecte est partiellement définie de manière interne au dialecte lui-même :

- Au moyen des vérifications statiques associées à chaque opération (par exemple, un programme MLIR contenant une boucle for dont la région associée est vide est rejeté).

- Au moyen des transformations de code internes au dialecte, qui doivent préserver la sémantique fonctionnelle de la spécification initiale. Il peut s'agir de l'optimiser, comme avec la parallélisation automatique de boucles dans le dialecte **affine**, ou de produire une spécification équivalente en vue de transformations ultérieures, par exemple en réécrivant une boucle `for` sous la forme d'une boucle `while`, comme en Fig. 3.12.

- Au-delà des optimisations spécifiques à chaque dialecte, nous avons déjà indiqué auparavant que tout programme SSA peut faire l'objet d'optimisations et de transformations génériques reposant sur les propriétés SSA du code et ne modifiant pas le niveau d'abstraction de la spécification : élimination de code redondant, propagation de constantes, élimination de code mort, etc.

Le reste de la sémantique est définie au travers des étapes de *lowering* décrites dans la section suivante.

3.4.3 La compilation d'une spécification MLIR

Les passes de *lowering* d'un dialecte à l'autre

Construire un compilateur au-dessus de MLIR consiste à déterminer les dialectes nécessaires au cours de la compilation, ainsi que les transformations de code permettant de passer d'un niveau d'abstraction supérieur à un autre plus bas, en approchant le code exécutable. On appelle ces transformations des passes de *lowering*. Chacune d'entre elles doit spécifier les dialectes d'entrée et de sortie, et chaque transformation de code (réécriture au même niveau d'abstraction ou *lowering*) peut imposer des contraintes sur la spécification d'entrée et de sortie (e.g. quels dialectes/opérations/types sont acceptés ou non).

La sémantique des opérations d'un dialecte n'est pas seulement caractérisée par ses vérifications et transformations internes : dans la mesure où le

```
func @for_with_yield(%arg0: memref<1024xf32>) -> f32 {
  %cst = arith.constant 0.000000e+00 : f32
  %c0 = arith.constant 0 : index
  %c10 = arith.constant 10 : index
  %c2 = arith.constant 2 : index
  %0 = scf.for %arg1 = %c0 to %c10 step %c2
       iter_args(%arg2 = %cst) -> (f32) {
    %1 = memref.load %arg0%arg1 : memref<1024xf32>
    %2 = arith.addf %arg2, %1 : f32
    scf.yield %2 : f32
  }
  return %0 : f32
}
```

```

func @for_with_yield(%arg0: memref<1024xf32>) -> f32 {
  %cst = arith.constant 0.000000e+00 : f32
  %c0 = arith.constant 0 : index
  %c10 = arith.constant 10 : index
  %c2 = arith.constant 2 : index
  %0:2 = scf.while (%arg1 = %c0, %arg2 = %cst) : (index, f32) -> (index, f32) {
    %1 = arith.cmpi slt, %arg1, %c10 : index
    scf.condition(%1) %arg1, %arg2 : index, f32
  } do {
  ^bb0(%arg1: index, %arg2: f32):  // no predecessors
    %1 = arith.addi %arg1, %c2 : index
    %2 = memref.load %arg0%arg1 : memref<1024xf32>
    %3 = arith.addf %arg2, %2 : f32
    scf.yield %1, %3 : index, f32
  }
  return %0#1 : f32
}
```

FIGURE 3.12 : La réécriture, interne au dialecte scf, d'une boucle for en une boucle while.

```
func @for_with_yield(%buffer: memref<1024xf32>) -> (f32) {
  %sum_0 = arith.constant 0.0 : f32
  %sum = affine.for %i = 0 to 10 step 2
         iter_args(%sum_iter = %sum_0) -> (f32) {
    %t = affine.load %buffer%i : memref<1024xf32>
    %sum_next = arith.addf %sum_iter, %t : f32
    affine.yield %sum_next : f32
  }
  return %sum : f32
}
```

```
func @for_with_yield(%arg0: memref<1024xf32>) -> f32 {
  %cst = arith.constant 0.000000e+00 : f32
  %c0 = arith.constant 0 : index
  %c10 = arith.constant 10 : index
  %c2 = arith.constant 2 : index
  %0 = scf.for %arg1 = %c0 to %c10 step %c2
       iter_args(%arg2 = %cst) -> (f32) {
    %1 = memref.load %arg0%arg1 : memref<1024xf32>
    %2 = arith.addf %arg2, %1 : f32
    scf.yield %2 : f32
  }
  return %0 : f32
}
```

```
func @for_with_yield(%arg0: memref<1024xf32>) -> f32 {
  %cst = arith.constant 0.000000e+00 : f32
  %c0 = arith.constant 0 : index
  %c10 = arith.constant 10 : index
  %c2 = arith.constant 2 : index
  br ^bb1(%c0, %cst : index, f32)
^bb1(%0: index, %1: f32):  // 2 preds: ^bb0, ^bb2
  %2 = arith.cmpi slt, %0, %c10 : index
  cond_br %2, ^bb2, ^bb3
^bb2:  // pred: ^bb1
  %3 = memref.load %arg0%0 : memref<1024xf32>
  %4 = arith.addf %1, %3 : f32
  %5 = arith.addi %0, %c2 : index
  br ^bb1(%5, %4 : index, f32)
^bb3:  // pred: ^bb1
  return %1 : f32
}
```

FIGURE 3.13 : Le dialecte affine (haut) compilé vers le dialecte scf (milieu) compilé vers les dialectes SSA de base de MLIR (bas).

lowering d'une spécification doit préserver sa sémantique, la conversion d'un dialecte en un dialecte de plus bas niveau tient lieu de définition sémantique.

Ainsi, par exemple, le dialecte **linalg**, qui regroupe les abstractions relatives à l'algèbre linéaire, est compilé vers le dialecte **affine** qui représente les boucles affines, et lui-même est compilé vers le dialecte **scf** qui représente le flot de contrôle structuré (e.g. `if`, `for`) habituel dans des langages comme C. À partir du dialecte **scf**, on descend sans difficulté vers les dialectes de base, avec les opérations de branchement explicites, etc. Fig. 3.13 illustre ce mécanisme (le fragment du milieu est le même que le code source en Fig. 3.12).

Les traits Certaines opérations, qu'elles appartiennent au même dialecte ou non, ont des propriétés communes qui impliquent des contraintes (ou des transformations) communes lors de la compilation. Ces propriétés sont décrites sous la forme de *traits* déclarés lors de la définition d'une opération. Le trait *IsTerminator*, par exemple, est partie prenante de la définition des opérations terminales et informe le compilateur qu'aucune transformation ne peut positionner cette opération ailleurs qu'à la fin d'un *basic block*. Nous exploitons en Chapitre 4 le trait *HasNoSideEffects*, dont l'absence dans la définition d'une opération indique que cette dernière accède à la mémoire persistante (et ne peut donc commuter, durant la compilation, avec une autre opération du même type). Parmi les opérations présentées en Section 3.2.3, les accès à la mémoire sont naturellement dans cette situation, mais les appels de fonction aussi.

Les chaînes de compilation

Les différentes transformations de code (internes aux dialectes et entre les dialectes) sont combinées au sein de chaînes composant les réécritures, depuis le formalisme source (nécessitant éventuellement une phase d'importation/-normalisation, i.e. de mise en conformité avec les contraintes SSA) jusqu'au formalisme cible, préservant tout au long du processus la sémantique fonctionnelle de la spécification et son respect des contraintes SSA. Fig. 3.14 donne l'exemple d'une telle chaîne, sous une forme légèrement simplifiée [5]. En particulier :

[5] Certains dialectes comme **shape** sont négligés, et les boîtes HLO et Base SSA sont en fait des agrégats de dialectes.

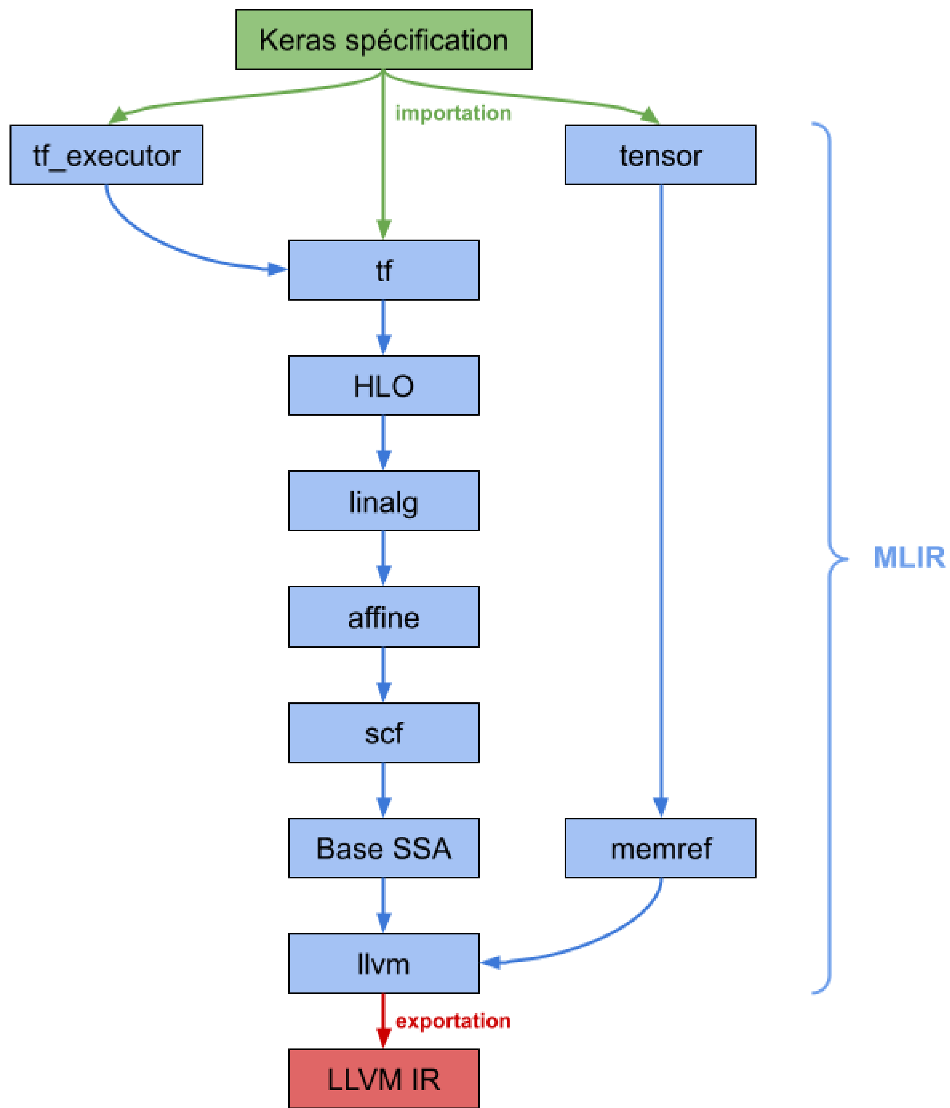


FIGURE 3.14 : Une chaîne MLIR (bleu) compilant incrémentalement une spécification Keras importée (vert) en un code LLVM IR exporté sous un format compatible avec LLC ou les autres outils LLVM (rouge).

- Les dialectes **tf** et **tf_executor** désignent la représentation MLIR des opérations TensorFlow.
- Les dialectes de la famille HLO (**lhlo**, **mhlo**...) désignent les opérations du compilateur d'algèbre linéaire XLA.
- Le dialecte **linalg** désigne la représentation MLIR standard des opérations d'algèbre linéaire.
- Le dialecte **affine** désigne la représentation MLIR des nids de boucles affines.
- Le dialecte **scf** désigne la représentation MLIR des opérations de flot de contrôle structuré (boucles for...).
- Les dialectes agrégés dans Base SSA désignent les opérations SSA élémentaires (arithmétique de base, branchement entre basic blocks...).
- Le dialecte **llvm** désigne la représentation MLIR de LLVM IR, l'IR standard de l'écosystème LLVM.
- Le dialecte **tensor** désigne la représentation des données agrégées sous la forme de tenseurs abstraits, et le dialecte **memref** leur représentation sous la forme d'une mémoire persistante explicite.

Les dialectes évoqués jusqu'ici ne représentent qu'un petit sous-ensemble de ceux qui sont embarqués dans les chaînes de compilation standard, ce qui signifie que de nombreuses chaînes différentes sont possibles, à partir de plusieurs formalismes d'entrée et ciblant plusieurs plate-formes d'exécution (code LLVM IR, bytecode IREE, noyaux GPU, etc.) [6]. À ce titre, les dialectes sont le point d'extension principal de MLIR : l'incorporation de formalismes synchrones que nous présentons en Chapitres 4 et 5 consiste en fait en la définition de dialectes synchrones.

Du fait de ce principe d'extensibilité, MLIR prend place dans la chaîne de compilation de nombreux frameworks ML, lesquels implémentent l'importation d'un formalisme de spécification particulier vers des dialectes MLIR de haut niveau – par exemple, une spécification JAX est directement importée

[6]Grâce à la modularité induite par les dialectes, plusieurs implémentations peuvent être générées à partir d'une même spécification ; ainsi, le dialecte **tf-lite** décrit la mise en œuvre embarquée TensorFlow Lite[99] d'une même spécification Keras/TensorFlow.

vers les dialectes de la famille HLO et fait l'économie des dialectes spécifiques à TensorFlow qui sont illustrés au sommet de Fig. 3.14.

3.5 Conclusion

Dans ce chapitre, nous avons d'abord identifié les propriétés constitutives du *principe SSA*, qui est mis en œuvre, non seulement par le *formalisme SSA*, mais également par les langages synchrones, tel Lustre.

Nous avons ensuite montré que le *formalisme* SSA garantit ces propriétés en imposant le respect d'une propriété *suffisante* (le critère de dominance) par une analyse de graphe de faible complexité (l'analyse de dominance). Par comparaison, nous verrons au Chapitre 5 que Lustre garantit le principe SSA au moyen d'une analyse de nature très différente (la conjonction du calcul d'horloges et de l'analyse de causalité).

Nous avons ensuite formellement défini la syntaxe et la sémantique des primitives du formalisme SSA. Notre définition inclut les deux extensions SSA traditionnelles : la modularité à base de fonctions et la manipulation explicite de la mémoire persistante[7].

Notons dès à présent, bien que nous traitions essentiellement ce point dans le chapitre suivant, que le formalisme SSA admet des comportements cycliques (e.g. branchement arrière d'un *basic block* sur lui-même), mais qu'aucune des primitives SSA exposées ne permet de spécifier le traitement réactif d'entrées-sorties dans le temps, ni la synchronisation entre l'exécution d'une spécification SSA et l'exécution de son environnement.

Finalement, ce chapitre présente MLIR, à la fois représentation intermédiaire (IR) de compilation et boîte à outils pour la compilation d'applications HPC et ML. L'IR MLIR est fondée sur SSA, qu'elle étend par :

- Les régions, au moyen desquelles un SCFG local peut être attaché à une opération MLIR, permettent la spécification d'opérations structurées (défintion de fonctions, mais aussi boucles *for* ou opérations *if-then-else*).

[7]Nous soulignons que cette dernière n'est pas couverte par l'analyse de dominance. Pour qu'une spécification SSA manipulant la mémoire persistante respecte le principe SSA, il est donc nécessaire que des vérifications supplémentaires garantissent qu'aucune zone mémoire allouée mais non initialisée ne soit utilisée dans les calculs ou dans les décisions du programme.

Pour facilier la spécification dans les phases initiales de compilation, le respect du critère de dominance SSA peut être temporairement *relâché* à l'échelle d'une région. Nous exploitons cette propriété en Section 5.2 pour étendre MLIR/SSA aux primitives flot-de-données synchrones de Lustre.

- Les dialectes permettent de grouper un ensemble d'opérations, de types et d'attributs correspondant à un domaine d'utilisation particulier et faisant l'objet de transformations de code particulières. Ils constituent le point d'extensibilité principal de MLIR, dont nous tirons profit pour définir les dialectes **sync** au Chapitre 4 et **lus** en Section 5.2.

Chapitre 4

Étendre SSA à la concurrence synchrone

Dans ce chapitre, nous étendons la syntaxe et la sémantique SSA en leur ajoutant les mécanismes opérationnels nécessaires pour représenter la concurrence réactive synchrone. Nous mettons ces extensions en œuvre sous la forme d'un dialecte MLIR.

L'objectif est de pouvoir décrire le comportement de systèmes embarqués (donc réactifs) ayant des composants HPC (dont DL). Dans ces systèmes, les composants interagissent cycliquement entre eux et avec leur environnement.

Comme nous l'avons vu en Fig. 3.2 (page 45), le formalisme SSA permet déjà de décrire des comportements cycliques, avec le branchement d'un *basic block* sur lui-même. De telles boucles peuvent également être illimitées dans le temps : il suffit que le branchement arrière ne soit pas conditionnel ou que la condition d'arrêt ne soit jamais vraie. Ce modèle d'exécution présente néanmoins deux limites fondamentales, sur lesquelles porte notre extension :

1. Si on peut effectivement découper l'exécution en cycles, (e.g. changement de cycle lors des sauts en arrière), un tel découpage n'est pas strict : il n'est pas préservé par les transformations de code SSA, à commencer par les optimisations.

2. Les entrées-sorties d'une fonction SSA ne sont pas cycliques, et donc définir l'interaction entre composants et avec l'environnement ne peut reposer que sur l'utilisation d'effets de bord, y compris pour la synchronisation (au travers de protocoles complexes et fragiles).

Solution proposée Pour lever ces limites, nous donnons une représentation de premier ordre à la modularité d'automates communicants, qui maintiennent eux-même un état interne et dont l'exécution est découpée en cycles discrets. Cette modularité suit le paradigme flot de contrôle des approches SSA existantes. Cela explique pourquoi notre représentation des automates communicants est conçue comme une extension de la modularité par fonctions impératives (similairement aux *process* et aux *threads*) et n'est pas fondée sur la modularité des nœuds flots de données. Pour identifier nos *fonctions réactives*, nous utilisons d'ailleurs le mot-clef `func`. Pour permettre de distinguer les deux types de fonctions en vue de certaines étapes de compilation, les fonctions réactives sont définies comme opération dans un nouveau dialecte MLIR **sync**.

Les différences entre fonctions SSA traditionnelles et fonctions réactives sont marquées au niveau de nouvelles primitives nécessaires pour spécifier rigoureusement :

- les mécanismes permettant à une fonction réactive de communiquer cycliquement (entrées/sorties) avec son environnement ;
- les barrières de cycles d'exécution, au moyen desquelles on signale que tous les calculs d'un cycle sont terminés dans une fonction réactive, celle-ci pouvant se mettre en attente du cycle suivant.

Dans ce chapitre nous introduisons les constructions nécessaires : fonctions réactives, signaux d'entrée-sortie et les opérations d'envoi et de réception associées, barrière de cycle. Nous détaillons à chaque fois le problème particulier auquel chacune d'entre elles apporte une solution, et expliquons pourquoi nous l'avons choisie plutôt qu'une autre.

Nos extensions sont entièrement compatibles avec la syntaxe et la sémantique traditionnelle de SSA, assurant le fait que les transformations de code correctes du point de vue SSA classique préservent la sémantique synchrone des systèmes formés de fonctions réactives.

Nous présentons dans un premier temps la sémantique intuitive et la compilation de chacune de ces constructions, et détaillons finalement leur sémantique formelle en Section 4.4. Ce travail sémantique s'appuie en particulier sur la liaison que nous établissons entre la notion d' *absence* du paradigme synchrone et les notions d*indéfinition* couramment utilisées dans les chaînes de compilations basées sur SSA (notamment LLVM).

L'extension SSA que nous proposons est implémentée dans le nouveau dialecte MLIR appelé **sync** que nous avons évoqué précédemment.

4.1 Le découpage de l'exécution en cycles disjoints

Dans cette section, nous exposons d'abord en Section 4.1.1 la nécessité d'introduire en SSA une barrière de cycle stricte, garantissant que l'affectation d'une opération à son cycle ne varie pas durant la compilation. Nous explorons ensuite plusieurs solutions pour atteindre cet objectif :

- En Section 4.1.2, nous en étudions deux qui associent les cycles aux éléments structurels de SSA (opérations de branchement, basic blocks), mais les rejetons.
- Nous retenons finalement en Section 4.1.3 une troisième solution, consistant à introduire une nouvelle opération primitive `tick` pour identifier les barrières de cycle d'exécution tout en assurant, au moyen des propriétés du formalisme SSA (e.g. dominance) et de sa mise en œuvre MLIR (e.g. protection des effets de bord) que `tick` ne puisse commuter avec aucune autre opération.

4.1.1 La nécessité d'une barrière de cycle

SSA donne la possibilité de spécifier des exécutions cycliques (à l'aide de branchements) mais ne définit pas de mécanisme standard permettant d'identifier les débuts de cycle. Cela peut être fait de plusieurs manières. Par exemple, on peut décider que chaque branchement en arrière termine un cycle et en démarre un autre. Cependant, une telle représentation est susceptible de varier durant le processus de compilation, notamment du fait d'optimisations communes (comme le déplacement d'invariants de boucle, ou le déroulement de boucle). Or, la concurrence synchrone exige que les cycles d'exécution ne se superposent pas et qu'ils forment une *base de temps logique*, où toute opération est associée à exactement un cycle, lequel peut être identifié au moyen de son index. Dès lors, une barrière de cycle que les opérations ne peuvent pas traverser lors des optimisations est nécessaire.

Ce temps logique est une notion descriptive utilisée dans l'analyse, mais il est également indispensable pour permettre la synchronisation de l'exécution cyclique avec des bases de temps externes. Par exemple, l'exécution périodique temps-réel repose typiquement sur la synchronisation du déclenchement d'un cycle (la base de temps logique) avec un *timer* matériel.

4.1.2 Associer les cycles aux *basic blocks* ?

Il est tentant de lier (et même d'identifier) :

- Le découpage du temps en cycles tel que le décrit le paradigme synchrone ;
- Le découpage d'un programme SSA en *basic blocks* reliés par des opérations de branchement, tel que le décrit le formalisme SSA.

Ainsi ramènerions-nous la séquentialité globale entre les cycles d'un programme synchrone à celle entre *basic blocks* d'une spécification SSA. La difficulté est que, pour pouvoir représenter des cycles de calcul contenant un contrôle non-trivial, il faut pouvoir distinguer deux types d'opérations de branchement : celles qui terminent un instant logique synchrone, passant au suivant, et celles qui ne changent pas d'instant logique. Dans cet esprit, deux méthodes peuvent être envisagées :

- L'annotation de l'opération de branchement SSA au moyen d'un attribut MLIR dédié. Cette approche est illustrée en Fig. 4.1. Un branchement t ainsi annoté signale à l'environnement que tous les calculs assignés au cycle courant ont été exécutés. Le contrôle est rendu à l'environnement et ne reviendra pas avant que ce dernier ne soit achevé à l'échelle du système (i.e. synchronisation avec le temps externe). À ce moment, le contrôle sera rendu au *basic block* destinataire de t.

  Cette approche permet de tirer profit de l'analyse de dominance pour représenter naturellement la non-persistance des variables d'un cycle à l'autre.

  Mais cette préservation sémantique implique une limitation fondamentale : les branchements annotés restent des branchements standard. Ainsi, ils font l'objet des transformations de code (e.g. optimisations) SSA pratiquées par le compilateur, et celles-ci ne préservent pas les opérations de branchement. Considérons ainsi l'application d'un simple

```
1 func @periodic()->() {
2 ^reset:
3   %x0 = call @init():()->tensor<64xi8>
4   br ^step(%x0:tensor<64xi8>)
5 ^step(%x1:tensor<64xi8>):
6   %x2 = addi %x1, %x1: tensor<64xi8>
7   br {cycle} ^step(%x2:tensor<64xi8>)
8 }
```

FIGURE 4.1 : La synchronisation des calculs au moyens d'annotations des opérations de branchement (en bleu). L'annotation de changement de cycle est ici `{cycle}`.

```
1 func @periodic()->() {
2 ^reset:
3   %x0 = call @init():()->tensor<64xi8>
4   br ^step(%x0:tensor<64xi8>)
5 ^step(%x1:tensor<64xi8>):
6   %x2 = addi %x1, %x1: tensor<64xi8>
7   %x3 = addi %x2, %x2: tensor<64xi8>
8   %x4 = addi %x3, %x3: tensor<64xi8>
9   br {cycle} ^step(%x4:tensor<64xi8>)
10 }
```

FIGURE 4.2 : La synchronisation des calculs au moyens du paramétrage des opérations de branchement, au terme d'une transformation de déroulement de boucle.

déroulement de boucle de longueur 3 à l'exemple de Fig. 4.1. Dans le code résultant (en Fig. 4.2), l'opération produisant %x4 (ligne 8), a lieu au cycle 0 au lieu du cycle 2. La sémantique synchrone n'est donc pas préservée par cette optimisation.

- L'extension syntaxique et sémantique de SSA au moyen d'une nouvelle opération de branchement, dont la sémantique est entièrement définie par nos soins. Cette approche est illustrée en Fig. 4.3 ; comme précédemment, elle permet de changer de cycle en même temps qu'on change de *basic block*. Ne pré-existant pas dans MLIR/SSA, elle ne fait l'objet d'aucune transformation de code standard : son comportement ne

```
1 func @periodic()->() {
2 ^reset:
3   %x0 = call @init():()->tensor<64xi8>
4   br ^step(%x0:tensor<64xi8>)
5 ^step(%x1:tensor<64xi8>):
6   %x2 = addi %x1, %x1: tensor<64xi8>
7   br_cycle ^step(%x2:tensor<64xi8>)
8 }
```

FIGURE 4.3 : La synchronisation des calculs au moyens d'une nouvelle opération de branchement dédiée (en bleu).

> risque donc pas d'être perturbé par des optimisations usuelles. Mais, revers de la médaille, elle nous prive de l'algorithmique existante sur les SCFGs. Par conséquent, la spécification résultante ne peut plus faire l'objet des transformations SSA usuelles, ce qui contredit directement notre objectif initial (i.e. la compilation conjointe et efficace des aspects réactifs et des aspects HPC d'une même spécification).

Ces deux approches nous amènent à conclure que le changement de cycle et le changement de *basic block* ne peuvent être identifiés à ce niveau sans soit changer en profondeur les algorithmes de compilation, soit renoncer à certaines optimisations, et donc à l'efficacité du code généré.

4.1.3 L'opération `tick`

La solution que nous choisissons est l'introduction d'une nouvelle opération, nommée `tick`, pour explicitement identifier les barrières de cycle. Fig. 4.5 illustre l'incorporation d'une telle opération dans une spécification SSA.

Pour assurer que les transformations de code SSA ne changent pas l'affectation des opérations à leurs cycles de calcul synchrones (en ne permettant pas à ces opérations de commuter avec les opérations tick), il est impératif d'utiliser seulement les mécanismes fournis par SSA et par MLIR. Ainsi, l'algorithmique existante du compilateur reste inchangée.

L'ordre basé sur la dominance. Comme nous l'avons vu précédemment et comme le montre la syntaxe de `tick` fournie en Fig. 4.4, `tick` peut recevoir un nombre arbitraire de variables en entrée, ce qui permet de spécifier le fait

```
  <type> += in(<type>) | out(<type>)
    <op> += (<var>?) = tick(<tvar>*)
          | <var> = sync(<tvar> <tvar>+)
          | <var> = input(<var>):<type>
          | (<var>?) = output(<var>:<var>):<type>
          | <var> = undef:<type>
 <op_id> += inst <fun_name> <inst_id>
```

FIGURE 4.4 : Extensions de la syntaxe SSA pour permettre la programmation réactive synchrone. Les symboles grisés sont repris de la figure Fig. 3.3. La syntaxe “+=” étend la définition d’un non-terminal.

que les opérations produisant ces variables sont exécutées avant la barrière de cycle.

Pour permettre de spécifier qu’une opération se passe *après* une barrière de cycle, l’opération `tick` produit une valeur. Celle-ci a le type `none`, qui est le type unitaire de MLIR. Au même titre que le type unitaire de la programmation fonctionnelle ou que les *signaux purs* de la programmation synchrone [81], une variable de type `none` représente une synchronisation pure. Elle ne transporte pas d’informations mais force, du fait du critère de dominance, les opérations qui l’utilisent à venir après l’opération qui la produit. Pour faciliter la spécification de ces contraintes d’ordre, nous introduisons également l’opération `sync` qui permet la transmission synchronisée de deux ou plusieurs variables. L’opération prend ainsi deux variables ou plus en entrée, et copie la valeur de sa première entrée vers sa sortie dès lors que toutes ses entrées sont arrivées.

Comme le montre Fig. 4.5 (gauche), cette manière de contraindre l’ordre est particulièrement utile au début de la compilation, lorsque les données agrégées sont représentées sous la forme de types abstraits (comme les *tenseurs* de MLIR) manipulés par des opérations sans effets de bord.

L’ordre basé sur les effets de bord. Nous définissons également `tick` comme ayant des effets de bord, ce qui signifie qu’elle ne peut commuter, durant les transformations de code SSA, avec les opérations qui lisent ou écrivent dans la mémoire persistante. Cette manière de contraindre l’ordre

```
1 func @periodic1()->() {
2 ^reset:
3
4   %x0 = call @init():
5     ()->tensor<64xi8>
6   br ^step(%x0:tensor<64xi8>)
7 ^step(%x1:tensor<64xi8>):
8   %x2 = addi %x1, %x1:
9     tensor<64xi8>
10  %s = tick(%x2:tensor<64xi8>)
11  %x3 = sync(%x2:tensor<64xi8>,
12             %s:none)
13  br ^step(%x3:tensor<64xi8>)
14 }
```

```
1 func @periodic2() -> () {
2 ^reset:
3   %x0 = alloc(): memref<64xi8>
4   call @init(%x0):
5     (memref<64xi8>)->()
6   br ^step
7 ^step:
8   call @do_addi(%x0)
9     :(memref<64xi8>)->()
10  tick()
11
12
13  br ^step
14 }
```

Figure 4.5 : Synchronisation des calculs par rapport aux barrières de cycle (`tick`). À gauche, l'ordre est assuré en exploitant la dominance. À droite, les effets de bord.

est particulièrement utile tard dans la chaîne de compilation, une fois que l'allocation mémoire des données tensorielles a été accomplie. C'est le cas en Fig. 4.5 (droite), qui pourrait être une implémentation du programme en Fig. 4.5(gauche).

Propriétés structurelles. Pour garantir que l'exécution d'une *fonction réactive SSA* soit une séquence infinie de cycles d'exécution bornés dans le temps, nous exigeons :

1. Qu'une telle fonction ne contienne pas d'opérations `return`. Cette contrainte pose un problème syntaxique dans les spécifications de haut niveau du type de celle que nous exhibons en Fig. 4.6, car MLIR exige que tout *basic block* (et donc en l'occurence l'unique *basic block* de la région associée à la fonction `periodic1`) s'achève par une opération terminale, ce que n'est pas `scf.while`. Nous résolvons ce problème en définissant simplement une nouvelle opération terminale `halt`, qui bloque l'exécution mais devrait ne pas être accessible.

2. Que tout cycle d'exécution potentiellement non-borné d'une spécification SSA contienne au moins une opération `tick`.

```
1 func @periodic1()->() {
2   %x0 = call @init(): ()->tensor<64xi8>
3   %true = arith.constant 1: i1
4   scf.while(%x1=%x0): (tensor<64xi8>)->() {
5     scf.condition(%true)
6     do {
7   ^step(%x1:tensor<64xi8>):
8     %x2 = arith.addi %x1, %x1: tensor<64xi8>
9     %s = tick(%x2:tensor<64xi8>)
10    %x3 = sync(%x2:tensor<64xi8>,
11               %s:none)
12    scf.yield %x3:tensor<64xi8>
13  }
14  halt
15 }
```

FIGURE 4.6 : La barrière de cycle à l'intérieur d'une boucle `while` définie dans le dialecte **scf** (*Structured Control Flow*) de MLIR.

La première exigence ci-dessus est synonyme d'abence de création/destruction dynamiques de composants réactifs. Cette règle pourrait être relâchée, mais cela compliquerait largement le traitement syntaxique et sémantique, et dépasse nos objectifs d'expressivité.

4.1.4 Compilation

La transformation d'une spécification SSA standard en un code séquentiel exécutable est un procédé bien connu. Cependant, l'introduction de `tick` change fondamentalement la sémantique SSA, en exigeant une interaction cyclique avec l'environnement/l'ordonnanceur.

Inversion du contrôle. La manière dont cette interaction est traditionnellement implémentée dans la compilation de langages synchrones[81, 9, 8] est illustrée en Fig. 4.7. Le flot de contrôle du code source (fonction `@n` dans notre cas) est complètement restructuré (inverti) pour produire un programme sémantiquement équivalent avec une seule opération `tick` placée dans une boucle infinie. Ce programme est typiquement structuré en trois fonctions :

- une fonction *step* (pas de calcul) contenant tout le code itéré par la

boucle infinie, sauf l'opération `tick` (`@n_step` en Fig. 4.7).

- une fonction *reset* (initialisation) contenant le code d'initialisation de la boucle infinie (`@n_reset` en Fig. 4.7).
- une fonction *driver* contenant l'appel à `@n_reset`, suivi de la la boucle infinie, qui elle-même contient l'appel à `@n_step` et l'opération `tick` (`@n_drv` en Fig. 4.7).

Cette inversion du contrôle demande une représentation explicite de l'état de la fonction originale *entre cycles de calcul*. Cet état est initialisé par la fonction *reset*. Ensuite, à chaque itération de la boucle infinie (qui implante un cycle de calcul synchrone) l'état courant est pris en entrée par la fonction *step*, qui le décode avant de potentiellement le modifier et le produire en sortie pour être transmis au cycle suivant.

Dans notre exemple, cet état est transmis par les arguments de `^step` et consiste en :

- Une valeur booléenne (de type `i1`) utilisée pour déterminer quelle fonction, de `@f` ou de `@g`, doit être exécutée au prochain cycle (état du contrôle).
- Une valeur de type `i32` autorisant la transmission de la sortie de `@f` à `@g` (état des données).

Au début de chaque cycle, l'état est décodé, les calculs requis sont déclenchés, puis un nouvel état est encodé et transmis au cycle suivant. Ce procédé est usuellement représenté par une *fonction d'étape* distincte, dans notre cas `@n_step`.

Les compilateurs synchrones classiques se contentent habituellement de générer les fonctions *step* et *reset* et la structure de données décrivant l'état de l'application. Utilisée en conjonction avec la modularité synchrone décrite en Section 4.3, cette approche permet une génération de code modulaire[1] [9]. Cependant, la fonction *driver* (dans notre cas `@n_drv`), et en particulier l'implémentation de `tick`, sont généralement considérées comme trop dépendantes de l'architecture cible pour être générées.

[1] Une fonction *step* et une fonction *reset* par module synchrone, la représentation de l'état d'un module incluant celle des sous-modules qu'il inclut hiérarchiquement, et les fonctions *step* et *reset* appelant les fonctions *step* et *reset* de ses sous-modules.

```
func @n()->() {     func @n_step(%s:i1,%x:i32)  func @n_reset()->(i1,i32){
^start:                       ->(i1,i32){          %s0 = constant 0 : i1
  %x = call @f():     cond_br %s, ^true,^false     %x0 = constant 0 : i32
    ()->(i32)       ^true:                         return %s0,%x0:i1,i32
  tick()              %x1 = call @f()            }
  call @g(%x):          :()->(i32)
    (i32)->()         %false = constant 0:i1     func @n_drv()->() {
  tick()              return %false,%x1            %s0,%x0 = call @n_reset()
  br ^start             :i1,i32                      :()->(i1,i32)
}                   ^false:                        br ^step(%s0,%x0:i1,i32)
                      call @g(%x):(i32)->()      ^step(%s:i1,%x:i32):
                      %true = constant 1:i1        %s1,%x1 = call @n_step(%s,%x)
                      return %true,%x:i1,i32           :(i1,i32)->(i1,i32)
                    }                              tick()
                                                   br ^step(%s1,%x1:i1,i32)
                                                 }
```

FIGURE 4.7 : L'approche de génération de code des langages synchrones (par inversion du contrôle). Le code source est à gauche, l'implémentation au milieu et à droite.

Cette approche de compilation a été longuement testée en pratique [7, 45], et a montré ses forces (notamment en termes de modularité), mais également ses limites. Ces dernières sont essentiellement dues à la forme très particulière et fixe du code généré, avec une seule fonction *step* et une représentation de l'état qui doivent couvrir les besoins de toutes les transitions d'un cycle à l'autre. Dans notre exemple, la variable d'état de type `i32` est affectée et passée au cycle suivant à chaque pas d'itération alors même qu'elle n'est sémantiquement produite (par `@f`) que dans les cycles impairs.

Une première évaluation de l'inefficacité des représentations inversées d'état des programmes synchrones est fournie dans les travaux précédents sur l'optimisation de ces représentations dans la compilation du langage Esterel[81] [2]. Ces optimisations exploitent des propriétés de redondance similaires à celle que nous avons mis en évidence dans notre exemple, où la position du contrôle (transmise par `s`) rend inutile la transmission de `x` dans les cycles pairs. Il est important de souligner que, une fois l'inversion du contrôle

[2] L'allocation mémoire pour les langages synchrones pose également d'autres difficultés, comme le fait que la génération de code modulaire nécessite, dans le cas général, un encodage inefficace pour les constructions mémoire, qui utilisent non pas une mais deux variables pour chaque élément de l'état. Différents articles adressent cette difficulté dans les travaux antérieurs[38, 34].

réalisée, les optimisations de compilation classiques sont confinées au corps de la fonction *step*, les optimisations impliquant différents cycles d'exécution devant être spécialement conçues pour chaque formalisme particulier et chaque manière d'encoder l'état (ce qui réduit largement leur portée).

Compilation sans inversion du contrôle. Alors qu'elle est employée par tous les compilateurs de langages synchrones, l'inversion totale du contrôle n'est pas une obligation. D'ailleurs, non seulement l'inversion du contrôle peut limiter les optimisations, mais dans de nombreux cas *l'implémentation doit avoir une structure différente.* Par exemple, dans les applications avioniques MIF/MAF[34], l'implémentation *doit* avoir une structure similaire à celle de Fig. 4.7 (gauche), où un motif périodique global (la boucle principale, nommée *major frame* ou MAF) est segmentée par les opérations `tick` en intervalles de temps de même longueur (MIFs, *minor frames*) dont chacun contient un code différent.

Pour permettre l'implémentation sans restructuration de tout graphe SSA réactif satisfaisant les propriétés définies précédemment, nous proposons de revenir aux fondamentaux de la conception de systèmes réactifs, en rendant explicite l'intéraction avec l'ordonnanceur du système. Dans cette approche, une spécification réactive SSA comme la fonction `@n` de Fig. 4.7 (gauche) est vue comme un processus séquentiel s'exécutant sous un ordonnanceur multi-tâches coopératif.

A chaque fois que l'exécution de @n atteint une opération `tick`, le contexte d'exécution de @n[3] est sauvegardé et le contrôle est rendu à l'ordonnanceur. Lorsque ce dernier détermine qu'un nouveau cycle d'exécution peut être déclenché, il restaure l'état de @n et lui rend le contrôle en laissant l'opération `tick` (la même qui avait rendu le contrôle au cycle précédent) se terminer et rendre le contrôle en séquence.

Ce mécanisme opérationnel, dont la sémantique formelle est définie en Section 4.4, peut aisément être implémenté sur une large variété de plateformes d'exécution :

- *timers* bas niveau sans OS (*bare metal*)[34] ;
- services système POSIX comme `longjmp` (c'est la méthode que nous mettons en œuvre dans les expérimentations décrites en Chapitres 6 et

[3]C'est à dire son état. Mais cet état n'a pas besoin d'un codage explicite sous la forme d'une structure de données comme dans le cas de l'inversion du contrôle.

7) ;

- mécanismes de coroutines disponibles dans de nombreux langages ;
- services RTOS comme `PERIODIC_WAIT` dans le standard avionique IMA/ARINC 653[3].

Naturellement, cette proposition n'exclut pas l'approche classique de la compilation des langages synchrones.

4.2 Les entrées-sorties cycliques

Dans cette section, nous prenons appui sur la barrière de cycle définie précédemment pour introduire les primitives décrivant les entrées-sorties cycliques.

4.2.1 De la barrière de cycle aux entrées-sorties cycliques

Sans barrière de cycles synchrone, la manipulation d'entrées-sorties cycliques est difficile en SSA. Par exemple, des transformations classiques de code SSA[4] vont naturellement regrouper un nombre arbitraire d'acquisitions de données, les plaçant avant le traitement de ces données et de la production des sorties correspondantes. Cela est incompatible avec une exécution réactive, où l'environnement a potentiellement besoin de la sortie du cycle n avant de permettre la lecture de l'entrée du cycle $n + 1$.

Les barrières de cycles (opération `tick`) introduites précédemment permettent de synchroniser les entrées et les sorties avec les cycles synchrones.

Dans des implantations embarquées traditionnelles, ces entrées-sorties sont typiquement implantées par des lectures et des écritures d'adresses mémoire spécifiques. En remontant les niveaux d'abstraction, ces accès mémoire peuvent être représentés par des accès à des variables C volatiles, ou tout simplement par des appels de fonctions I/O dédiées. Finalement, les langages réactifs fournissent des abstractions de ces mécanismes, sous la forme de *variables* ou *signaux* d'entrée-sortie qui peuvent être lus et écrits à chaque cycle, à condition de respecter les deux contraintes liées au modèle synchrone :

[4]Déroulage de boucles suivi d'une étape d'ordonnancement plaçant les opérations sans dépendances au début du corps de boucle.

- Une variable de sortie ne peut être écrite plus d'une fois par cycle.
- Toutes les lectures d'une variable d'entrée durant un cycle d'exécution doivent retourner le même résultat.

En comparaison, si l'on fait abstraction des effets de bord, les fonctions MLIR interagissent seulement de deux manières avec leur environnement :

- Au début de leur exécution, elles lisent les valeurs de leurs arguments d'entrée, qui ne changent pas durant l'exécution de la fonction.
- Lorsqu'elles atteignent une opération `return`, la fonction rend définitivement le contrôle et la valeur de ses sorties à l'environnement.

Les variables *volatiles* n'existent que dans certains dialectes très bas niveau, et sont très dépendantes de l'implémentation. Elles ne peuvent donc pas servir de base pour la définition d'un mécanisme général de communication réactive.

4.2.2 Les canaux d'entrée-sortie

La solution que nous choisissons pour représenter les I/O cycliques est basée sur la représentation de *canaux* d'entrée et de sortie d'une fonction réactive. Ces canaux sont définis sous la forme d'arguments d'entrée de la fonction ayant les types spéciaux `in(t)` ou `out(t)`, où `t` est le type de la donnée transmise par le canal. L'accès à ces canaux passe exclusivement par deux opérations nommées `input` et `output`, dont la syntaxe est fournie en Fig. 4.4. L'opération `input` a un unique argument de type canal d'entrée. Chaque fois qu'elle est exécutée, `input` échantillonne le canal et produit une valeur du type correspondant, qui est placée dans une variable de sortie. Une opération `output` a deux entrées : un canal de sortie et une seconde variable (du type de donnée correspondant) dont la valeur est écrite sur la sortie.

Pour spécifier la relation d'ordre entre les opérations d'entrée-sortie et les autres opérations, nous utilisons les mécanismes que nous avons déjà discuté pour `tick` dans Section 4.1.3. Nous définissons `input` et `output` comme ayant des effets de bord, ce qui empêche la permutation avec les opérations accédant à la mémoire et avec les appels de fonction. De plus, pour permettre la définition d'un ordre fondé sur la dominance, l'opération `output` a une sortie optionnelle de type `none` (à l'instar de `tick`).

4.2.3 Propriétés structurelles

Notre implémentation courante exige que seules les fonctions représentant des comportements réactifs puissent communiquer au moyen de canaux (i.e. manipuler des variables de types `in(t)` ou `out(t)` et contenir les opérations `input` et `output`). Nous exigeons également que les fonctions réactives aient uniquement des arguments de type canal, et qu'aucune opération ne produise de variable de type canal.

Pour respecter l'exigence synchrone d'après laquelle chaque variable a au plus une valeur durant chaque cycle (nous développerons cet aspect dans Chapitre 5), il est également requis que chaque variable canal soit lue par au plus une opération `input` ou `output` entre deux instances de `tick`. Cette propriété sémantique peut être garantie au moyen de propriétés structurelles, e.g. en exigeant que tout chemin du CFG entre deux opérations sur le même canal contienne une opération `tick`.

4.2.4 Compilation

Comme expliqué précédemment, des mécanismes bas niveau variés peuvent être utilisés pour implémenter les opérations `input` et `output`, les plus courants étant les variables volatiles et les appels à des fonctions d'entrée-sortie. Pour des raisons de portabilité, notre compilateur adopte la seconde approche (qui peut d'ailleurs encapsuler la première).

Pour chaque fonction réactive, nous associons à chaque paramètre de type `in(t)` ou `out(t)` une étiquette $n \in \mathbb{N}$ l'identifiant de manière unique auprès de l'environnement. Ensuite :

- Nous transformons chaque paramètre de type `in(t)` en paramètre de type `(i32)->(t)`, i.e. en pointeur sur une fonction qui prend l'étiquette de la variable de type `in(t)` et produit un résultat de type `t`.
- Nous transformons chaque paramètre de type `out(t)` en paramètre de type `(i32, t)->(none)` (le premier argument étant l'étiquette).
- Nous transformons chaque opération `input` ou `output` en appel de la fonction correspondante. Nous utilisons l'opération MLIR `call_indirect`, qui est une variante de l'opération `call` dédiée aux pointeurs vers fonctions.

```
func @n(%i:in(i32),
        %o:out(i32))->() {
  br ^step
^step:
  %x = input(%i):i32

  output(%o:%x):i32

  tick()
  br ^step
}
```

```
func @n(%i:(i32)->(i32),
        %o:(i32i32)->())
       ->() {
  br ^step
^step:
  %x = call_indirect %i (0):
    ()->(i32)
  call_indirect %o (0,%x):
    (i32)->()
  call @tick() : ()->()
  br ^step
}
```

FIGURE 4.8 : Génération de code pour les opérations d'entrée-sortie.

Fig. 4.8 fournit un exemple : un programme réactif avec un canal d'entrée et un canal de sortie qui, à chaque cycle, copie son entrée sur sa sortie.

Lorsque la valeur produite par `input` ou prise en entrée de `output` est un pointeur sur une zone de la mémoire persistante [5], il faut y apporter une attention particulière pour éviter les erreurs mémoire (accès à des zones mémoires non-allouées, fuites mémoires, etc.). Dans notre implémentation, nous supposons que la zone mémoire sur laquelle pointe le résultat d'une opération `input` (resp. le paramètre d'une opération `output`) est allouée et désallouée par la fonction elle-même.

4.3 Modularité réactive

Dans cette section, nous présentons la manière dont nous manipulons les fonctions réactives comme des processus concurrents, communiquant les uns avec les autres et avec leur environnement. Cela revient à mettre en œuvre une nouvelle modularité réactive, distincte de la modularité de SSA fondée sur les fonctions transformationnelles.

4.3.1 De la modularité basée sur les fonctions à la modularité réactive

La barrière de cycle et les entrées-sorties cycliques vues précédemment nécessitent que les différents composants d'un programme synchrone partagent leurs ressources entre eux et avec l'environnement.

[5] Comme les variables de type `memref` dans MLIR.

En effet, arrivés à ce point, nous avons déjà vu que, du point de vue des communications entre composants, à chaque cycle, un noœud synchrone doit :

- Acquérir ses entrées, qui sont produites par son environnement ou sont transmises par lui.
- Réaliser une série de calculs, mettant éventuellement à jour son état interne.
- Écrire ses sorties, qui sont traitées par son environnement ou transmises par lui au composant chargé de les traiter.
- Rendre le contrôle à son environnement lorsqu'il a achevé tous les calculs pour le cycle courant.

Ce comportement n'existe pas en SSA : un programme SSA n'interagit avec son environnement qu'au début et à la toute fin de son exécution. Deux fonctions ne peuvent être actives en même temps que si l'une appelle (directement ou indirectement) l'autre, l'état du système étant une pile d'appels (tel que formellement défini en Section 3.3).

En comparaison, les modèles formels qui sous-tendent les formalismes réactifs et synchrones sont concurrents. Du point de vue le plus général, À l'image d'Esterel [81], *l'exécution de deux sous-modules d'une spécification réactive peuvent avancer de manière concurrente, se synchronisant et communiquant dans les deux sens.* Déterminer qu'une telle exécution ne bloque pas est en général indécidable, si les données entières sont autorisées, et NP-difficile (irréalisable en pratique) si le langage source utilise seulement des variables booléennes. De plus, l'implémentation de spécifications aussi générales peut être très inefficace en raison des synchronisations compliquées entre modules concurrents.

Pour cette raison, les langages synchrones ont imposé de simples contraintes structurelles permettant la compilation rapide et modulaire : dans chaque module, les calculs d'un cycle d'exécution doivent former un graphe de dépendances acyclique. De plus, l'exécution de ces opérations se fait toujours en réalisant toutes les opérations d'entrée avant les calculs, avec toutes les sorties groupées en fin de cycle. Ainsi, les opérations d'un cycle peuvent être accomplies de manière *atomique*.

```
func @main(%ic:in(i32),
           %oc:out(i32))->(){
  br ^step
^step:
  %i = input(%ic):i32
  %x1 = inst @sum "a" (%i:i32):i32
  tick()
  %x2 = inst @sum "a" (%i:i32)
  output(%oc:%x2):i32
  tick()
  br ^step
}
```

```
func @sum(%ic:in(i32),
          %ic:out(i32))->(){
  %s0 = constant 0 : i32
  br ^step(%s0:i32)
^step(%s:i32):
  %i = input(%ic):i32
  %s1 = addi %s, %i : i32
  tick(%s1:i32)
  %o = call @f(%s1):
    (i32)->(i32)
  output(%oc:%o): i32
  tick()
  br ^step(%s1:i32)
}
```

FIGURE 4.9 : Instanciation de fonctions réactives.

4.3.2 L'instanciation des fonctions réactives

Pour permettre la représentation de ce mécanisme dans notre extension de SSA, nous introduisons la notion d'*instance* d'une fonction réactive `@f`, qui est un processus (doté d'un état séparé) exécutant la fonction `%f` sous l'ordonnanceur du système. Les instances sont indentifiées de manière unique. Dans cette thèse, pour une meilleure lisibilité, les identifiants sont des listes de chaînes de caractères. Dans la mise en œuvre de nos cas d'étude, nous nous servons d'entiers.

Dans un programme synchrone, nous supposons que la première instance de la fonction réactive qui reçoit le contrôle lorsque le système démarre identifiée par la liste vide `[]`. Toutes les autres instances sont définies inductivement et reçoivent possiblement le contrôle durant l'exécution, au moyen de l'opération `inst` (syntaxe en Fig. 4.4). Si `i` est l'identifiant d'instance de la fonction `@f` qui contient l'opération "`inst @g str`", alors l'identifiant de l'instance de g est est `str::i`.

Nous fournissons en Fig. 4.9 un exemple de l'instantiation d'un sous-module. Le système est composé de deux fonctions réactives et de deux instances : l'instance `[]` de la fonction `@main` et une instance `["a"]` de la fonction `@sum`. Signalons que les deux opérations `inst` de `@sum` déclenchent un cycle d'exécution de la même instance (`["a"]`) [6]. L'instance `[]` lit l'entrée

[6] En présence de motifs d'activation périodiques, cela permet de dérouler la boucle d'activations sur une longueur égale au plus petit commun multiple des périodes, puis

unique fournie par l'environnement lors des cycles impairs, déclenche une réaction de l'instance `["a"]` à chaque cycle (en lui fournissant la valeur de `%i` en entrée) et produit en sortie la dernière sortie de `["a"]` lors des cycles pairs. L'instance `["a"]` de `@sum` a maintient un état accumulant la somme des entrées reçues lors des cycles impairs. Lors des cycles pairs, elle calcule et produit la variable `%o`.

4.3.3 Contraintes structurelles

Le nombre et les types des variables d'entrée et de sortie d'une opération `inst` doivent correspondre à la signature de la fonction réactive qu'elle instancie.

4.3.4 Compilation

Lorsque la conversion vers une fonction d'étape n'est pas souhaitée, nous transformons chaque instance en un processus s'exécutant sous un ordonnanceur coopératif. Ce mécanisme étend celui de Section 4.1.4 en clarifiant comment l'ordonnanceur passe le contrôle entre les instances : il décrit chaque fonction réactive comme un automate communiquant avec d'autres fonctions réactives et avec son environnement. Lorsqu'une opération `inst` est atteinte, les entrées et le contrôle sont transmis à l'instance, qui s'exécute ensuite jusqu'à atteindre une opération `tick` ; alors, son état est sauvegardé et elle retourne ses sorties à l'appelant (sémantique formelle en Section 4.4).

Par comparaison, la méthode de compilation basée sur la conversion vers les fonctions d'étape implémente l'instanciation de sous-modules par l'appel de fonction[9].

4.4 Sémantique formelle

La dernière étape de notre extension de SSA consiste à définir la sémantique formelle des spécifications SSA réactives, formées d'au moins une instance de fonction SSA réactive.

Avant d'en décrire les règles formelles, nous traitons le problème posé par la notion de l'*absence* synchrone.

---

d'appliquer une passe de propagation de constantes, ce qui élimine le test d'activation sur le compteur d'itération. On appelle cette transformation “expansion sur l'hyper-période”.

4.4.1 Absence et indéfinition

Nous partons de l'exemple en Fig. 4.7 (à gauche) et de son implémentation (à droite). Le passage de l'un à l'autre change légèrement le comportement interne du programme : à gauche, la variable `%x` est transmise uniquement lors des cycles impairs. Mais à droite, `%x` a été ajoutée à l'état du programme, qui est calculé est transmis à chaque cycle. Dès lors, dans le *basic block* `^false` de la fonction `@n_step`, sous la sémantique SSA, il faut retourner une valeur pour `%x` à chaque cycle, même si le programme source n'en précise pas une.

Dans ce cas, nous faisons le choix naturel de maintenir la valeur précédente, ce qui permet éventuellement, aux étages les plus bas de la compilation, de représenter l'état dans la mémoire persistante (par exemple, sous la forme d'une variable statique). Mais il aurait également été correct d'utiliser une constante quelconque de type `i32` (e.g. 0, ou 42) ou une valeur calculée dynamiquement, dans la mesure où cette valeur n'est jamais utilisée dans les calculs. La situation est similaire en Fig. 4.9 : la fonction `@sum` produit une valeur de sortie seulement aux cycles pairs, ce qui signifie que la valeur de `%x1` dans la fonction `@main` n'est jamais correctement initialisée. Pour autant, la spécification est entièrement déterministe, car cette valeur *absente/indéfinie* n'est jamais utilisée dans les calculs.

De telles spécifications sont communes dans la modélisation synchrone de systèmes multi-périodiques ou traitant différentes échelles de temps, ce qui explique pourquoi l'*absence* joue un rôle important dans la sémantique de tous les langages synchrones [7]. Nous verrons par exemple en Chapitre 5 que dans le langage Lustre, une variable *absente* lors d'une réaction synchrone a une valeur *indéfinie* et ne peut être utilisée dans les décisions ou dans les calculs. Cette définition correspond pleinement à celle de $\perp$ dans Section 3.3.

Cependant, les langages synchrones comme Lustre sont conçus pour supporter des analyses de faible complexité (connues sous le nom de *calcul d'horloges*) garantissant qu'une donnée non-initialisée n'est jamais utilisée dans les calculs (de la même manière que le formalisme SSA est conçu pour supporter l'analyse de dominance). Ces analyses exploitent le couplage étroit entre la propagation du contrôle (par le flot de données) et l'initialisation, tout autant que les exigences de synchronisation (appelées *contraintes d'horloges*) mises en oeuvre par les primitives du langage.

Mais notre extension réactive de SSA est de plus bas niveau. Elle vise seulement à permettre des représentations internes du compilateur, où l'implémentation efficace peut nécessiter *la représentation explicite et la mani-*

pulation de l'absence/indéfinition, ce qui n'est pas possible en SSA à cause du respect de la dominance.

Le même problème de représentation des comportements non-définis est central dans la compilation de langages comme C/C++. Ainsi, certaines implémentations de SSA, dont celle de LLVM[64], nommée LLVM IR, proposent une solution de contournement. Celle-ci consiste à étendre SSA avec deux types d'indéfinition supportant différentes analyses et approches d'optimisation[64]. Chacune d'entre elles est une valeur spéciale pouvant être affectée à une variable, ce qui permet de respecter les critères (syntaxiques) de SSA :

- `llvm.undef` représente *l'absence d'initialisation*.
- `llvm.poison` représente un comportement pathologique – par exemple, elle est produite en cas de dépassement du domaine sur une opération entière.

$$\frac{op(l) = \text{“}v = \texttt{undef}\text{“}}{(Run^{\mathsf{f}}(l,s), cs, m) \to (Run^{\mathsf{f}}(nxt(l), s[v \leftarrow \bot]), cs, m)}\ \textbf{(undef)}$$

FIGURE 4.10 : Sémantique de `undef`

Contrairement à LLVM IR, notre extension SSA n'a pas besoin d'ajouter plusieurs valeurs spéciales découvrant divers types d'erreurs. Nous nous contentons d'autoriser la manipulation explicite de l'indéfinition sans briser les règles de correction structurelles de SSA. Pour ce faire, nous introduisons l'opération `undef` (syntaxe en Fig. 4.4, sémantique en Fig. 4.10). Cette opération affecte $\bot$à sa variable de sortie. Pour comprendre comment cette valeur est propagée, considérons la fonction `@n_step` de Fig. 4.7. Comme expliqué précédemment, dans le *basic block* `^false`, le second argument de `return` pourrait prendre n'importe quelle valeur à la place de `%x`, car il n'est jamais utilisé dans les calculs. Utilisant l'opération `undef`, nous pouvons réécrire cette fonction pour éviter d'avoir à retourner une valeur définie, comme illustré en Fig. 4.11. Cette implémentation dévie légèrement du comportement original de Fig. 4.7 (à gauche), mais permet une génération de code plus efficace, comme détaillé plus bas.

```
func @n_step(%s:i1,%x:i32)->(i1,i32){
  cond_br %s, ^true,^false
^true:
  %x1 = call @f():()->(i32)
  %false = constant 0:i1
  return %false,%x1:i1,i32
^false:
  call @g(%x):(i32)->()
  %ux = undef(): i32
  %true = constant 1:i1
  return %true,%ux:i1,i32
}
```

FIGURE 4.11 : Utilisation de `undef` (en bleu) pour représenter l'absence.

Compilation de `undef` Il est notoirement difficile de maîtriser l'indéfinition dans LLVM [64]. Cependant, cette difficulté est une conséquence de l'objectif ambitieux de LLVM, qui vise à préserver par compilation non seulement les comportements définis, mais aussi ceux indéfinis du programme initial.

Dans notre cas, l'objectif est de simplement préserver les comportements *définis* du programme source, ceux dans lesquels aucun branchement, accès mémoire ou calcul n'utilise de valeurs indéfinies. Cette approche est cohérente avec le paradigme de la programmation synchrone, où l'absence de comportements indéfinis est supposée garantie par des analyses appliquées à la spécification de haut niveau (e.g. dans Lustre). Ainsi, `undef` peut être compilée vers *toute valeur SSA légale du dialecte cible* (dans le contexte de MLIR). En particulier :

- Remplacer `undef` par n'importe quel valeur définie (e.g. la constante 0 ou 42) permet la conversion du code en forme SSA standard, mais introduit des initialisations qui n'existent pas dans la spécification originale et qui peuvent compliquer l'optimisation.

- Remplacer `undef` par `llvm.undef` ou `llvm.poison` implique de viser LLVM IR et ses extensions SSA, mais n'introduit pas d'initialisations sémantiquement inutiles et donne accès aux transformations/optimisations tirant profit de l'indéfinition.

Pour montrer formellement la correction de la compilation, supposons que Int^* est le domaine sémantique de chaque variable, obtenu en étendant

$\overline{Int} = Int \cup \{\bot\}$ (defini en Section 3.3) aux valeurs spécifiques à la plate-forme cible. Par exemple, pour une compilation vers LLVM IR, nous définissons $Int^* = \overline{Int} \cup \{\texttt{llvm.poison}, \texttt{llvm.undef}\}$. Nous dotons Int^* de l'ordre partiel déterminé par $\bot \leq x$ pour tout $x \neq \bot$. Cet ordre partiel étend ponctuellement les valuations partielles des variables, et donc les états de l'exécution. Dès lors, nous avons :

Theorem 4.4.1. *Soient une fonction SSA non-réactive* [7] f *et* f' *le résultat du remplacement de toutes les instances de* `undef` *dans* f *par des valeurs de* $Int^* \setminus \{\bot\}$ *ou par d'autres variables (ne violant pas les règles de dominance). Soit un état d'exécution initial* s_0 *et une trace d'exécution* $s_0 \to \ldots \to s_n$ *de* f *qui ne bloque pas. Alors, il existe une trace unique* $s_0 = s'_0 \to \ldots \to s'_n$ *de* f' *telle que* $s_i \leq s'_i$ *pour tout* i*.*

Preuve : Étant donné que la trace de f ne bloque pas, les différences entre s_i et s'_i peuvent uniquement survenir au niveau des opérations `undef` du programme original, qui sont remplacées par des valeurs définies ou par `llvm.undef` ou `llvm.poison`. Ces différences peuvent uniquement être propagées par l'affectation, dans la mesure où elles n'atteignent ni décisions, ni accès mémoire, ni calculs. □

4.4.2 Règles de sémantique opérationnelle

Étant donnée une fonction réactive `f`, nous dénotons $in^{\texttt{f}}$ (respectivement $out^{\texttt{f}}$) l'ensemble ordonnée des canaux d'entrée de `f` (respectivement des canaux de sortie de `f`). Étant donnée une instance i, nous dénotons $r(i)$ sa fonction réactive.

L'**état d'exécution** d'une spécification réactive SSA est représenté par un triplet $< i, \mathcal{I}, m >$, où m est l'état de la mémoire partagée, i est l'identifiant de l'instance courante, et $\mathcal{I}$ est un dictionnaire associant à chaque identificant d'instance l'état de cette instance. L'état de l'instance i est $\mathcal{I}(i) = (s, cs, si, so)$, où s est l'état de la fonction $r(i)$ en cours d'exécution, cs est le contexte d'appel, $si : in^{r(i)} \to \overline{Int}$ est l'état des canaux d'entrée de i, et $so : out^{r(i)} \to \overline{Int}$ l'état des canaux de sortie.

Sous ces notations, les règles décrivant la sémantique opérationnelle sont fournies en Fig. 4.12. La règle (local) transforme les transitions SSA non-réactives des instances (dénotées $\to$ et définies en Figures 3.7 et 4.10) en

[7] Qui peut utiliser `undef`, mais pas `tick`, ni les I/O cycliques, ni la modularité synchrone.

$$\frac{\mathcal{I}(i) = (s, cs, si, so) \quad (s, cs, m) \rightarrow (s', cs', m')}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (s', cs', si, so)], m' >} \textbf{(local)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{``}v = \mathtt{input}(ic)\text{``}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s[v \leftarrow si(ic)]), cs, si, so)], m >} \textbf{(in)}$$

$$\frac{i = [] \quad \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{``}\mathtt{tick}\text{``}}{< i, \mathcal{I}, m > \underset{si'}{\overset{so}{\Longrightarrow}} < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si', \bot)], m >} \textbf{(system-tick)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{``}\mathtt{output}(oc : v)\text{``}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si, so[oc \leftarrow s(v)])], m >} \textbf{(out)}$$

$$\frac{\begin{array}{c}\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{``}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_k)\text{``} \\ \mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so')\end{array}}{< i, \mathcal{I}, m > \Rightarrow < i' :: i, \mathcal{I}[(i' :: i) \leftarrow (Run^{\mathtt{g}}(l', s'), cs', \bot[in_l^{\mathtt{nd}} \leftarrow s(w_l) \mid 1 \leq l \leq k], \bot)], m >} \textbf{(inst)}$$

$$\frac{\begin{array}{c}\mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so') \quad op(l') = \text{``}\mathtt{tick}\text{``} \\ \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{``}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_m)\text{``}\end{array}}{< i' :: i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(l, s[v_i \leftarrow so'_i \mid 1 \leq i \leq n]), cs, si, so)], m >} \textbf{(inst-tick)}$$

FIGURE 4.12 : Extension réactive de la sémantique opérationnelle SSA de Fig. 3.7. En rouge, les règles de modularité réactive.

transitions du système réactif (dénotées $\Rightarrow$). Toutes les autres règles impliquent des opérations réactives et donc une interaction avec l'ordonnanceur. Les règles (in) et (out) concernent les entrées-sorties de l'instance. La règle (system-tick) est la seule qui interagisse avec l'environnement par le truchement d'entrées-sorties et éventuellement de synchronisation temporelle. Les règles (inst) et (inst-tick) (en rouge) implémentent la modularité réactive. La première donne le contrôle à une instance lorsqu'une opération `inst` est atteinte, et la seconde reçoit le contrôle lorsque l'exécution de l'instance atteint une opération `tick`.

4.4.3 Correction de la compilation.

Bien que nous définissions une sémantique formelle pour les extensions SSA réactives, nous ne fournissons pas de preuve formelle du fait que leur compilation vers le code exécutable est correcte. Pour ce faire, il faudrait notamment prouver que les primitives du dialecte `sync` fonctionnent comme prévu, ce qui impliquerait de modéliser les plateformes cibles allant du métal nu aux systèmes d'exploitation temps-réel.

Mais même en supposant que ces primitives sont correctement implé-

mentées, il reste à prouver que l'ensemble des transformations SSA correctes préservent la sémantique des spécifications réactives.

Le problème est difficile, et engendre de nombreux sous-problèmes également difficiles. Par exemple, bien que nous fournissions une sémantique formelle pour les spécifications réactives SSA, cette sémantique ne couvre pas les extensions SSA spécifiques à MLIR. Or, celles-ci jouent un rôle important dans notre travail. Par exemple, comme exposé dans Section 4.1.3, l'opération `tick()` est définie comme ayant des effets de bord. C'est ce mécanisme qui empêche les appels de fonction[8] de commuter avec les opérations `tick()` et donc de déborder de leurs cycles respectifs.

Or, les effets de bord sont spécifiés dans MLIR au moyen de propriétés opérationnelles appelées *traits*. Ces constructions spécifiques à MLIR doivent être prises en compte par toute formalisation de la correction des transformations de code SSA. En d'autres termes, si l'objectif est de montrer formellement la correction de *tout* enchaînement de transformations de code SSA correctes pour toute chaîne de compilation basée sur SSA (y compris la notre), la formalisation de la sémantique SSA que nous proposons est insuffisante et doit être étendue.

Cependant, nous considérons *avoir montré que les transformations de code SSA existantes peuvent être appliquées pleinement dans la compilation de spécifications réactives.*

Le premier argument accréditant cette affirmation est l'ensemble de cas d'étude complexes exhibé au Chapitre 7, lesquels s'exécutent conformément au code source après de nombreuses passes de transformation et d'optimisation SSA/MLIR. Cette forme de validation est analogue aux tests de non-régression utilisés pour valider la plupart des compilateurs (MLIR compris).

Un second argument vient du fait que la définition du dialecte `sync` que nous proposons s'appuie sur la sémantique MLIR/SSA existante (e.g. le fait que `tick` ait des effets de bord non-spécifiés). Ces choix fournissent une base pour le raisonnement semi-formel, et pour une future formalisation complète des extensions de MLIR SSA.

4.5 Conclusion

Dans ce chapitre, nous avons proposé un ensemble de primitives étendant le formalisme SSA en permettant la description de comportements réactifs

[8] Qui ont implicitement des effets de bord dans MLIR.

synchrones. Ces extensions (types, opérations, transformations de code), ont été groupées au sein du nouveau dialecte MLIR **sync**. Pour chacune des nouvelles primitives, nous avons exposé les arguments qui nous ont conduit à privilégier l'approche choisie plutôt qu'une autre et, après avoir donné leur sémantique intuitive, nous avons défini la sémantique formelle du formalisme résultant. Celle-ci étend la sémantique SSA traditionnelle avec du contrôle réactif (automates communiquants) et avec l'introduction d'une notion d'absence/indéfinition.

Contrôle réactif. Il s'agit des primitives permettant de spécifier l'interaction des fonctions réactives avec leur environnement :

- L'opération `tick` (et l'opération de synchronisation `sync`) permettant de spécifier le découpage de l'exécution en cycles logiques discrets et de garantir l'affectation d'une opération à son cycle même lorsque le programme est soumis à des transformations de code comme les optimisations.
- Les types paramétriques `in` et `out`, et les opérations de lecture/écriture cycliques `input` et `output` permettant de spécifier la lecture et l'écriture cycliques des entrées-sorties dans le temps.
- L'opération `inst` permettant de spécifier l'instanciation d'une fonction réactive sous l'ordonnanceur du système, en lieu et place de la modularité basée sur les appels de fonctions transformationnelles.

Ces primitives nous permettent d'avancer l'état de l'art en compilation de langages réactifs synchrones. En effet, en plus de la compilation classique par *restructuration du contrôle* ramenant toute fonction réactive à un couple de fonctions step/reset appellées par un *driver*, nous proposons de représenter les fonctions réactives sous la formes de processus communiquant entre eux et avec leur environnement sous l'ordonnanceur du système, indépendamment de l'implémentation de ce dernier.

Indéfinition d'une variable. La spécification réactive demande la manipulation de variables dont la valeur n'est pas définie. Le fait que ces valeurs ne soient pas utilisées dans les calculs est assuré par des critères de correction haut niveau, comme l'analyse d'horloges. Pour pouvoir donner une valeur *indéfinie* à une variable, nous introduisons l'opération `undef`, qui permet de

spécifier l'*indéfinition* d'une variable sans enfreindre le critère de dominance SSA.

Nous montrons que, dans tout programme synchrone correct (i.e. où aucune valeur absente/indéfinie n'est jamais utilisée dans aucun calcul ou aucune décision, comme c'est le cas au terme du *calcul d'horloges* de Lustre présenté au chapitre suivant), il est possible d'implémenter la variable produite par l'opération `undef` par n'importe quelle valeur SSA valide (e.g. constante, zone mémoire non-initialisée) tout en respectant le principe SSA. Ce résultat fait le lien entre spécification et compilation de spécifications SSA réactives et avance l'état de l'art du traitement de l'indéfinition en compilation SSA.

Le langage résultant est une IR impérative/flot de contrôle synchrone, qui peut être vue comme un *back-end* générique pour la programmation SSA synchrone. Nous mettons cette approche à l'épreuve dans le chapitre suivant, où **sync** nous sert de socle pour étendre SSA au flot de données synchrone.

Chapitre 5

Flot de données synchrone en MLIR

Dans le chapitre précédent, nous avons étendu la forme SSA avec les constructions permettant la représentation de comportements réactifs synchrones. Nous avons sélectionné l'approche la plus générale possible, qui reste pleinement compatible avec la sémantique et l'algorithmique SSA (permettant l'implémentation sans avoir à modifier les comportements existants de MLIR). Mais le formalisme synchrone que nous venons de définir est de bas niveau, plus adapté à servir de représentation intermédiaire de compilation que de langage de spécification.

Dans ce chapitre, notre objectif est de proposer un nouveau dialecte synchrone de haut niveau permettant la spécification naturelle et conjointe des aspects réactifs et HPC d'applications *réalistes*. Les aspects réactifs de ces applications doivent être spécifiés avec des primitives synchrones de haut niveau, et non directement dans notre extension SSA bas niveau. Nous montrons que le noyau flot de données du langage synchrone Lustre [1] peut être embarqué sous la forme d'un nouveau dialecte MLIR, que nous nommons **lus**. Cette approche permet la spécification d'applications dont les aspects réactifs sont modélisés au niveau d'abstraction de Lustre, tandis que les traitements sur les données sont modélisés aux niveaux d'abstraction d'autres dialectes comme `tf` (graphes TensorFlow), `linalg` (algèbre linéaire), ou `affine` (boucles affines).

Durant la compilation, les opérations réactives de **lus** sont compilées vers

[1] Que nous avons choisi pour sa diffusion large tout autant que pour sa simplicité.

le niveau d'abstraction du dialecte **sync** introduit en Chapitre 4. Les opérations de **sync** sont ensuite compilées en suivant d'abord les règles définies au Chapitre 4, puis les passes de compilation déjà implémentées en MLIR. Ces dernières sont notamment responsables de la *synthèse de buffers*, i.e. la transformation des agrégats de données abstraits utilisés au niveau **lus** (e.g. tenseurs) en pointeurs sur la mémoire persistante faisant l'objet d'opérations d'allocation et de déallocation dont les usages pathologiques (fuites mémoires, accès à la mémoire non-allouée) sont éliminés.

Le résultat est un compilateur entièrement fonctionnel, que nous appellons mlir-lus, et que nous avons implémenté par extension du compilateur MLIR mlir-opt. Il permet la spécification et la compilation d'applications réactives haute-performance. Nous décrirons en Chapitre 6 l'intégration de mlir-lus à un *framework* HPC existant (Keras/TensorFlow), et en Chapitre 7 son évaluation sur des applications non-triviales.

5.1 Le langage Lustre

Lustre est le langage synchrone le plus largement utilisé[45, 8]. La description formelle de Lustre a été réalisée dans de nombreux travaux antérieurs[18, 9] et n'est pas l'objet de cette thèse. Nous n'en décrirons donc ici qu'une sémantique intuitive. Par ailleurs, nous considérons seulement son noyau purement flot de données correspondant au dialecte SN-Lustre de [12] auquel la totalité du langage peut être ramenée par traduction structurelle.

Comme le dialecte **sync** présenté dans le précédent chapitre, Lustre est un formalisme réactif synchrone, où le temps logique est divisé en une succession *a priori* infinie de cycles qui ne se superposent pas dans le temps. Mais contrairement à **sync**, Lustre est aussi un formalisme flot-de-données.

Une spécification Lustre est une suite de définitions de *nœuds*. Chacun de ces nœuds décrit un graphe flot de données sous la forme d'une liste d'équations. Dans cette section, nous décrivons d'abord les divers types d'équations. Ensuite, nous introduisons les horloges et le calcul d'horloges qui permet de garantir statiquement qu'aucune variable absente/indéfinie n'est utilisée dans les calculs.

```
node stepper_drv(inc:int)
        returns (pos:int)
var pos_pre,pos_inc,pos_tmp,
    cst:int; ck:bool;
let
  pos_pre = 0 fby pos;


  pos_inc = pos_pre+inc;
  pos_tmp = pos_inc-10;
  ck = (pos_tmp >= 0);
  pos = if ck then pos_tmp
              else pos_inc;

  cst = 0 when ck ;
  actuate(cst);





tel
```

```
func @stepper_drv()->() {
  %c0 = constant 0 : i32
  %c10 = constant 10 : i32
  br ^step(%c0:i32)
^step(%pos_pre:i32):
  %inc = call @input_inc() :
    () -> (i32)
  %pos_inc = addi %pos_pre, %inc : i32
  %pos_tmp = subi %pos_inc, %c10 : i32
  %ck = cmpi "sge", %pos_tmp, %c0 : i32
  %pos = select %ck,%pos_tmp,
              %pos_inc : i32
  cond_br %ck, ^act(%c0:i32), ^out
^act(%cst:i32):
  call @actuate(%cst) : (i32) -> ()
  br ^out
^out:
  call @output_pos(%pos) : (i32)->()
  call @tick() : () -> ()
  br ^step(%pos:i32)
}
```

FIGURE 5.1 : Un *driver* de moteur pas-à-pas programmé en Lustre (à gauche) et en MLIR SSA (à droite). Le contrôle conditionnel est en vert, les traitements de données en rouge, les I/Os cycliques et la synchronisation temporelle en violet, et la manipulation de l'état en bleu.

```
1 node n(i:int)returns(o:int)
2 var k:int;
3 let
4   k = 1;
5   o = i + k;
6 tel
```

FIGURE 5.2 : Une spécification Lustre élémentaire.

Cycle	0	1	2	3
i	12	42	-16	50
k	1	1	1	1
o	13	43	-15	51

FIGURE 5.3 : Une trace d'exécution du nœud Lustre en Fig. 5.2.

5.1.1 Les nœuds

Le comportement cyclique d'un nœud. Un nœud Lustre décrit un graphe flot-de-données d'équations reliées entre elles par des variables flot-de-données. Fig. 5.1 (à gauche) présente l'un de ces nœuds en face du programme SSA flot de contrôle équivalent (à droite), et donne une première intuition des différences entre Lustre et SSA. En particulier, le comportement cyclique d'un nœud Lustre, contrairement à SSA, est implicite et ne doit pas être implémenté par des instructions impératives. Un nœud décrit les équations que le programme doit vérifier à chaque cycle d'exécution synchrone. Il n'est donc pas nécessaire d'y spécifier explicitement une barrière de cycle : à chaque cycle, ces équations sont traversées une seule fois, dans un ordre compatible avec les dépendances induites par les variables

Chaque variable est soit l'entrée d'un nœud, soit la sortie d'exactement une équation (*propriété d'affectation unique*). Les variables décrivent donc des séquences de données (des flots de données), et prennent une valeur déterminée à chaque cycle. Les constantes sont un cas particulier des variables : elles décrivent des séquences qui prennent la même valeur à chaque cycle.

Le nœud Lustre en Fig. 5.2 décrit un composant qui, à chaque cycle, affecte à k la valeur 1 et à o la valeur $i + k$. Un comportement possible de ce nœud (i.e. une trace d'exécution) est présenté dans le chronogramme en Fig. 5.3.

Traversée des équations d'un nœud. Lorsqu'elle est traversée, une équation lit la valeur de ses variables d'entrée, réalise éventuellement des calculs internes, et affecte une valeur à ses variables de sortie. Dans la mesure où la sémantique d'un nœud n'est pas affectée par l'ordre syntaxique de ses équations, on peut toujours supposer (comme hypothèse de forme normale) que les équations sont déjà triées dans l'ordre de leur traversée. SSA et Lustre ont donc l'ordre topologique sur les dépendances entre les données en commun, bien qu'il soit requis dans un cas et non dans l'autre.

Les dépendances entre les équations (dont la correction est vérifiée par une *analyse de causalité* statique) sont définies comme suit : la variable `y` produite par l'équation p et utilisée par l'équation c détermine une dépendance $p \to c$ dans tous les cas sauf un : quand c est une équation de la forme "`x = k fby y`". Dans ce cas, $c \to p$; cette forme d'anti-dépendance assure que la valeur est lue avant d'être réécrite. Ce comportement permet de définir `fby` comme un opérateur de délai (initialisé) lisant la valeur qui a été assignée à sa variable d'entrée *lors du cycle d'exécution précédent.*

5.1.2 L'opérateur fby

En effet, malgré la séparation stricte entre les cycles d'exécution, il est nécessaire de passer des valeurs d'un cycle à l'autre. Faute d'un tel mécanisme, il serait impossible de spécifier de nombreux comportements usuels dans les systèmes embarqués ou ML (et notamment de spécifier l'*état* des RNNs). Ainsi `fby` est la seule primitive Lustre permettant de spécifier l'*état* d'un nœud synchrone, persistant entre les cycles.

Pour donner un exemple, considérons le cas d'un intégrateur, un programme ayant une seule entrée i et dont l'unique sortie o prend comme valeur la somme de toutes les entrées depuis le début de l'exécution.

Ce comportement suppose que le nœud Lustre garde une mémoire de ses résultats précédents. Il est décrit au moyen du nœud en Fig. 5.4, qui recourt à l'opérateur `fby` : ce dernier introduit un *délai initialisé.*

Comme l'indique la trace d'exécution, la variable `sum` est initialisée à 0, puis le résultat de la somme est calculé pour le premier cycle (12). Ce résultat est aussi transmis au cycle suivant (membre droit de `fby`). Au deuxième cycle, le résultat précédent (12) est récupéré dans la variable `sum` et sommé avec une nouvelle entrée (42), avant d'être transmis au cycle suivant, etc.

```
1 node n(i:int)returns(o:int)
2 var sum: int;
3 let
4   sum = 0 fby o;
5   o = sum + i;
6 tel
```

Cycle	0	1	2	3
i	12	42	-16	50
sum	0	12	54	38
o	12	54	38	88

FIGURE 5.4 : La spécification Lustre d'un intégrateur et une trace d'exécution.

```
1 node n(i:int; ck:bool)
2      returns(o:int)
3 var ic: int; sum: int;
4 let
5   ic = i when ck;
6   sum = 0 fby o;
7   o = sum + ic;
8 tel
```

Cycle	0	1	2	3	4	5	6
i	7	5	1	33	2	4	6
ck	f	f	t	t	t	f	t
ic			1	33	2		6
sum			0	1	34		36
o			1	34	36		42

FIGURE 5.5 : Affectation conditionnelle. Nœud et trace d'exécution.

5.1.3 Les opérateurs when et merge

L'exécution conditionnelle s'exprime en Lustre en suivant un paradigme flot de données. Dès qu'une équation reçoit ses entrées à un cycle, elle s'exécute. Similairement, une équation ne s'exécute pas à un cycle où elle ne reçoit pas ses entrées. Ainsi, pour contrôler à quels cycles une équation s'exécute, il faut contrôler quand ses variables d'entrée sont produites. Cela se réalise à l'aide de l'opérateur de sous-échantillonnage `when`. Ainsi `y = x when c` signifie que y est présente aux cycles où x et c sont tous deux présents et c a la valeur $true$. À ces cycles, y prend la valeur de x. Le nœud en Fig. 5.5 est un intégrateur qui ne réalise son calcul qu'au cycles où l'entrée est $true$. Cet effet est obtenu en sous-échantillonnant la valeur de i pour produire ic.

L'opérateur `merge` permet de combiner deux séquences qui ne sont jamais toutes deux présentes au même cycle. Ainsi, si y et z sont deux variables du même type et si la variable y est présente aux cycles où c est présente et $true$ et z est présente aux cycles où c est présente et $false$, alors `x = merge c y z` est une variable présente aux mêmes cycles que c et prenant soit la valeur de y, soit celle de z, suivant la valeur de c. À l'aide de `merge`, tout programme Lustre peut être normalisé en une forme où chacune des instructions `fby`

```
1 node n(i:int; ck:bool)returns(o:int)
2 var ic: int; sum: int;
3     si: int; so: int;
4 let
5   si  = 0 fby so;
6   ic  = i when ck;
7   sum = si when ck;
8   o = sum + ic;
9   so = merge ck o (si when (not ck));
10 tel
```

Expression/Cycle	t0	t1	t2	t3	t4	t5	t6
i	7	5	1	33	2	4	6
ck	f	f	t	t	t	f	t
ic			1	33	2		6
sum			0	1	34		36
o			1	34	36		42
si	0	0	0	1	34	36	36
so	0	0	1	34	36	36	42

FIGURE 5.6 : Normalisation de l'état du programme Lustre en Fig. 5.5 et la trace d'exécution du programme normalisé correspondant à celle en Fig. 5.5.

s'exécute à chaque cycle de calcul. Prenons comme exemple le programme en Fig. 5.5, où l'instruction `fby` (en ligne 6) est exécutée seulement aux cycles où *ck* est *true*. Par normalisation, il produit le programme en Fig. 5.6.

5.1.4 Les horloges

Lustre implémente le principe SSA. En particulier, une variable est produite à chaque cycle où elle est utilisée dans les calculs. Cette propriété est garantie par analyse statique du programme source, par un procédé nommé *analyse d'horloges* (*clock analysis*).

Par définition, une horloge est un prédicat associant à chaque cycle d'exécution d'un programme synchrone une valeur booléenne. On associe naturellement une horloge :

- à chaque variable. Il s'agit du prédicat qui est vrai aux cycles où la variable est présente, et faux aux autres cycles.

- à chaque équation. Il s'agit du prédicat qui est vrai aux cycles où l'équation s'exécute.

Langage d'horloges. Les primitives du langage Lustre limitent l'expressivité des prédicats qui peuvent servir d'horloges. Ainsi, la définition de ces prédicats part toujours de *l'horloge de base* présente à chaque cycle et notée ".". Parmi les autres primitives, seules `when` et `merge` impliquent des variables ayant des horloges différentes, et ces horloges sont liées par une relation de sous-échantillonnage. Ainsi, à la fois dans `x = z when c` et dans `z = merge c x y`, si on note `clk(x)` l'horloge d'une variable x, on va avoir `clk(x) = clk(z) and c`. Cette horloge sous-échantillonnée est formellement notée `clk(z) on c`. Toutes les horloges d'un programme Lustre ont donc la forme `. on` c_1 `on ... on` c_k où $c_1, \ldots, c_k$ sont des variables booléennes.

Contraintes d'horloges. Chacune des quatre primitives de Lustre impose le respect de relations particulières[2] par les horloges des variables impliquées. Ainsi :

- Tous les signaux d'entrée et de sortie d'un appel de fonction ou d'un `fby` doivent avoir la même horloge.
- Les deux variables d'entrée de `x = z when c` doivent avoir la même horloge. La variable x doit avoir l'horloge `clk(z) on c`.
- Dans `z = merge c x y`, les variables z et c doivent avoir la même horloge. La variable x doit avoir l'horloge `clk(z) on c`, et la variable y aura l'horloge `clk(z) on (not c)`.

Le calcul d'horloges consiste fondamentalement à résoudre le système formé par les contraintes associées à toutes les équations d'un nœud Lustre. Ce problème est, dans le cas général, NP-complet, car la satisfiabilité booléenne peut y être réduite. Pour fournir des solutions en un temps compatible avec un processus de compilation, les compilateurs existants choisissent de

[2]Ces relations sont choisies pour assurer qu'un programme Lustre correct peut aussi être vu comme un réseau de processus de Kahn (Kahn Process Network)[58] dont l'exécution ne nécessite pas de prendre des décisions fondées sur l'absence d'une variable à un cycle, permettant une implantation flot de données déterministe.

```
1 node n(i:int;j:int;c:bool;d:bool)returns(o:int)
2 var  ci: int; dj: int;
3 let
4   ci = i when c;
5   dj = j when d;
6   o = ci + dj;
7 tel
```

FIGURE 5.7 : Un programme Lustre rejeté par l'inférence d'horloge.

redéfinir l'analyse d'horloges comme un algorithme d'inférence de types dans un système de types *à la* Hindley-Milner[22].

Le calcul d'horloges va rejeter des programmes dont les horloges ne peuvent pas satisfaire les contraintes requises. Ainsi, en Fig. 5.7, de par sa définition, la variable ci a l'horloge `. on c`. Similairement, dj a l'horloge `. on d`. Cependant, l'équation en ligne 6 contraint ci et dj à avoir des horloges égales. Cela imposerait l'égalité des horloges `. on c` et `. on d`, ce qui implique l'égalité des variables c et d. L'algorithme de typage ne pouvant pas imposer l'égalité de deux variables d'entrée distinctes, le programme est rejeté comme incorrect.

5.2 Le dialecte `lus`

Dans cette section, nous décrivons la manière dont nous incorporons les primitives de Lustre dans MLIR, sous la forme d'un nouveau dialecte nommé **lus**.

Nous commençons par le niveau spécification, en exhibant la syntaxe de notre extension et la phase de normalisation qui est nécessaire pour la rendre compatible avec les propriétés de correction structurelle imposées par SSA.

Ensuite, nous présentons la compilation de **lus** vers **sync** et ensuite vers le flot de contrôle structuré de MLIR.

5.2.1 Spécification flot de données synchrone en dialecte lus

Nœud lus. La grammaire du dialecte **lus** étendant MLIR est présentée formellement en Fig. 5.8. Conformément à cette syntaxe, une spécification réac-

```
<lus_spec> ::= <node_or_mlir_def>+
<node_or_mlir_def> ::= <node> | <mlir_def>
<node> ::= lus.node norm? <node_name> <node_iface> <node_body>
<node_iface> ::= (<list(<tvar>)>) <node_state>? ->(<list(<type>)>)
<node_state> ::= state(<list(<tvar>)>)
<node_body> ::= {<op>*<term_op> }
<op> ::= <lus_op> | <mlir_op>
<lus_op> ::= <var>=lus.fby<tvar>,<tvar>:<type>
           | <var>=lus.when not? <tvar>,<tvar>:<type>
           | <var>=lus.merge<tvar>,<tvar>,<tvar>:<type>
           | <var>=lus.kperiodic <kword>
           | <list(<var>)>=<instance_op>
<term_op> ::= lus.yield(<list(<tvar>)>) <node_state>?
<instance_op> ::= lus.instance<node_name> (<list(<tvar>)>):<node_type>
<node_type> ::= (<list(<type>)>)->(<list(<type>)>)
<kword> ::= <bin>*(<bin>+)
<bin> ::= 0 | 1
<list(X)> ::= <nvlist(X)>?
<nvlist(X)> ::= X | <nvlist(X)>,X
```

FIGURE 5.8 : Syntaxe du dialecte MLIR **lus**. Les identifiants d'opération sont surlignés en bleu.

tive MLIR peut comporter non seulement des nœuds réactifs, mais aussi des définitions MLIR pré-existantes (constantes, fonctions non-réactives, etc.).

Les nœuds réactifs sont déclarés à l'aide de l'opération `lus.node`. Celle-ci définit le nom du nœud, son interface, et une région comprenant une suite d'opérations réactives et non-réactives finie par l'opération `lus.yield` (qui est un terminateur dans le sens des définitions de Section 3.2).

Comme pour un nœud Lustre, l'interface d'un nœud lus spécifie les entrées et les sorties du nœud. Les entrées sont des variables nommées pouvant être utilisées dans le corps du module. Pour les sorties, on ne spécifie dans l'interface que leur type. Les variables alimentant ces sorties sont spécifiées par l'unique opération `lus.yield` et leur type doit correspondre à la spéci-

```
1 lus.node n(%i:i32)->(i32) {
2   %c0 = constant 0: i32
3   %sum = lus.fby %c0 %o: i32
4   %o = addi %sum, %i: i32
5   lus.yield(%o: i32)
6 }
```

FIGURE 5.9 : La spécification **lus** de l'intégrateur Lustre de Fig. 5.4.

fication d'interface.

Par exemple, la spécification **lus** en Fig. 5.9, qui donne le codage MLIR/**lus** du programme Lustre en Fig. 5.4, a une variable d'entrée nommée %i et l'unique sortie de type i32 (entier 32 bits) est alimentée par la variable %o.

A la différence de Lustre, nos interfaces (et les terminateurs) peuvent aussi spécifier les variables d'état du nœud, produites lors de la normalisation décrite plus tard dans cette section.

L'unique région associée à un nœud **lus** contient les équations du nœud. Cette région MLIR doit contenir un unique *basic block*. Les vérifications réalisée lors du chargement par MLIR d'une spécification **lus** vont rejeter les définitions de nœuds ayant plusieurs *basic blocks*.

Les opérations de l'unique région d'un nœud MLIR/lus sont ses équations. D'après la syntaxe, celles-ci peuvent être :

- Des opérations réactives correspondant à celles déjà présentées dans la description de Lustre (`lus.fby`, `lus.when`, `lus.merge`).

- Des opérations MLIR n'appartenant pas au dialecte **lus**. Celles-ci sont traitées comme les appels de fonction Lustre. Toutes les entrées et toutes les sorties d'une telle opération doivent avoir la même horloge, et un pas de leur exécution consomme/produit une valeur sur chacune des entrées/sorties. Ces opérations peuvent être hiérarchiques (e.g. des boucles, tests, etc.), mais leur définition interne doit être transformationnelle. Autrement dit, aucune opération réactive ne peut apparaître dans leur définition.

- L'opération `lus.instance` complète la définition de la modularité flot de données, permettant l'instanciation de sous-nœuds, telle qu'elle est définie par Lustre[46]. La sémantique de l'instanciation est le plus intuitivement décrite comme une opération au niveau du flot de données

– l'opération d'instanciation est remplacée par le flot de données du nœud instancié.

- L'opération `lus.kperiodic` permet de définir des variables booléennes dont la suite de valeurs est ultimement périodique ce qui nous permet de représenter des motifs d'activation réguliers[21] nous facilitant ainsi la spécification et la compilation. Le motif ultimement périodique donné sous la forme "$w_1(w_2)$" spécifie la suite ultimement périodique "$w_1\ w_2^*$".

La représentation du flot de donnée Lustre sous la forme d'un unique *basic block* MLIR est naturelle, correspondant à l'intuition que l'opération `lus.node` traverse les opérations de sa région à chaque cycle d'exécution. Cependant, cette intuition se heurte aux contraintes de bien-formé de SSA, et notamment à la dominance. Ainsi, la spécification en Fig. 5.9, qui encode en MLIR/lus le nœud Lustre de Fig. 5.4, est cyclique (cycle formé par les équations en lignes 3 et 4) et sera rejetée si l'analyse de dominance est réalisée. Ce même problème se posera dans le codage de tout comportement impliquant un état dont le calcul dépend de son ancienne valeur, ce qui n'est pas acceptable dans beaucoup de domaines comme le contrôle embarqué, le DL, le traitement de signal...

Pour permettre à MLIR d'accepter cette spécification, nous nous servons du mécanisme MLIR de relaxation de la dominance décrit en Section 3.4. Ainsi, les nœuds du dialecte **lus** désactivent par défaut l'analyse de dominance sur la région associée. Cette analyse est réactivée lorsque l'attribut `norm` est spécifié dans l'interface du nœud, signifiant que celui-ci a été normalisé et que l'ensemble des analyses de correction SSA sont activées et que les transformations de code définies au niveau SSA sont possibles.

Le dialecte **lus** interdit l'utilisation des accès mémoire dans les nœuds **lus**.

5.2.2 Normalisation d'une spécification

Rendre le nœud en Fig. 5.9 compatible avec les règles de dominance SSA n'est pas simple. Nous réalisons cette transformation en trois étapes à réaliser dans l'ordre :

- Remplacer chaque opération `lus.fby` qui n'est pas sur l'horloge de base du nœud avec une autre qui l'est, plus un protocole d'ajustement d'horloge formé d'opérations `lus.when` et `lus.merge`. Cette transformation

vise à rendre explicite le maintien implicite de l'état pendant les cycles où l'horloge de l'équation `lus.fby` initiale est fausse.

- Remplacer les opérations `lus.fby` par des dépendances portées par les boucles et représentées au niveau des opérations `lus.node` et `lus.yield`. À ce titre, il est important de noter que les primitives du dialecte **lus** étendent les primitives Lustre correspondantes, pour permettre la représentation des nœuds normalisés respectant toutes les contraintes SSA. Ainsi, **lus** est à la fois une extension de SSA et une extension de Lustre.

- Ordonner les opérations du nœud par un tri topologique respectant les dépendances de données.

```
lus.node @n (%i:i32,%ck:i1) -> (i32) {
  %0  = constant 0 : i32
  %s  = lus.fby %0,%o : i32
  %ic = lus.when %ck,%i : i32
  %o  = addi %s,%ic : i32
  lus.yield (%o:i32)
}
```

```
lus.node @n (%i:i32,%ck:i1)
                    -> (i32) {
  %0  = constant 0 : i32

  %si = lus.fby %0,%so : i32
  %s  = lus.when %ck,%si : i32
  %ic = lus.when %ck,%i : i32
  %o  = addi %s,%ic : i32
  %st = lus.when not %ck,%si : i32
  %so = lus.merge %ck,%o,%st : i32
  lus.yield (%o:i32)
}
```

```
lus.node norm @n (%i:i32,%ck:i1)
   state (%sx:i32) -> (i32) {
  %0  = constant 0 : i32
  %1  = lus.kperiodic 0(1)
  %si = lus.merge %1,%sx,%0 : i32
  %ic = lus.when %ck,%i : i32
  %s  = lus.when %ck,%si : i32
  %o  = addi %s,%ic : i32
  %st = lus.when not %ck,%si : i32
  %so = lus.merge %ck,%o,%st : i32
  lus.yield (%o:i32) state (%so:i32)
}
```

```
func @n(in(i32),in(i1),out(i32))->(){
^reset(%syi:in(i32),%syck:in(i1),
       %syo:out(i32)):
  %0=constant 0:i32
  br ^step(%0:i32)
^step(%sx:i32):
  %1  = lus.kperiodic 0(1)
  %si = lus.merge %1,%sx,%0 : i32
  %i = sync.input(%syi):i32
  %ck = sync.input(%syck):i1
  %ic = lus.when %ck,%i : i32
  %s  = lus.when %ck,%si : i32

  %o  = addi %s,%ic : i32
  %u1 = sync.output(%syo,%o):i32
  %st = lus.when not %ck,%si : i32
  %so = lus.merge %ck,%o,%st : i32
  %u2 = sync.tick(%u1:none,%so:i32)
  %son = sync.sync(%so:i32,%u2:none)
  br ^step(%son:i32)
}
```

```
func @n(in(i32),in(i1),out(i32))->(){
^reset(%syi:in(i32),%syck:in(i1),
       %syo:out(i32)):
  %0=constant 0:i32
  br ^step(%0:i32)
^step(%si:i32):


  %i = sync.input(%syi):i32
  %ck = sync.input(%syck):i1
  cond_br %ck ^true(%i,%si:i32,i32),
             ^out(%si:i32)
^true(%ic:i32,%s:i32):
  %o  = addi %s,%ic : i32
  sync.output(%syo,%o):i32
  br ^out(%o:i32)
^out(%so:i32):
  sync.tick()

  br ^step(%so:i32)
}
```

FIGURE 5.10 : En haut, le nœud source. Au milieu, les deux phases de normalisation du nœud. En bas, les deux phases de compilation du nœud vers **sync**. Les éléments de **lus** sont en bleu, et les éléments de **sync** sont en rouge.

Ces étapes de normalisation sont mises en évidence en Fig. 5.10. Ici, le nœud Lustre est le code source. Le suivant montre le résultat de la première phase de normalisation. Finalement, le troisième montre le résultat des deux phases suivantes de normalisation. Le nœud-résultat a l'attribut `norm`, ce qui active les analyses et l'algorithmique SSA.

Analyse de dominance SSA et analyse de causalité Lustre. Une fois cette phase de normalisation accomplie, une spécification **lus** est syntaxiquement compatible avec SSA. Mieux encore, si la spécification normalisée viole les règles exigées par l'analyse de causalité Lustre (par exemple, en exhibant un cycle sur les dépendances), elle viole aussi l'analyse de dominance SSA. Dès lors, il n'est pas nécessaire de réimplémenter l'analyse de causalité Lustre en MLIR.

5.2.3 La compilation d'un nœud normalisé

La compilation de `lus` vers `sync`

Après normalisation, la conversion du style flot de données des nœuds **lus** vers le style flot de contrôle du dialecte **sync** présenté dans le chapitre précédent est réalisée en deux étapes illustrées en Fig. 5.10 :

- Passer de la modularité flot de données basée sur les nœuds à la modularité flot de contrôle basée sur les fonctions réactives.
- Réimplémenter en forme flot de contrôle l'exécution conditionnelle.

Compiler la modularité flot de contrôle basée sur les nœuds. Cette première étape est largement facilitée par le remplacement des opérateurs `lus.fby` par des dépendances portées par les opérateurs `lus.node` et `lus.yield` (spécifiées à l'aide du mot-clef `state`).

Il nous suffit donc de créer au plus haut niveau du nœud la boucle infinie spécifique au modèle d'exécution synchrone, et de transformer les dépendances mentionnées plus haut en dépendances portées par cette boucle infinie. Le corps de la boucle infinie se termine par une opération `sync.tick`, pour marquer les barrières de cycles d'exécutions.

Au même moment, les entrées et les sorties du nœud doivent être transformées en canaux d'entrées et de sortie de types `in(t)` et `out(t)` correspondants, et des opérations `input` et `output` doivent être créées dans le code.

Toujours dans cette première étape de transformation, nous opérons une séparation entre initialisation des variables et opérations à exécution cyclique. Les initialisations sont placées dans le premier *basic block* de la fonction résultante (nommé `reset`), les opérations à exécution cyclique étant placées dans le second *basic block*, nommé `step`.

En Fig. 5.10, l'application de cette première étape de transformation produit le code en bas à gauche.

L'exécution conditionnelle et l'absence. Une fois la première étape de compilation réalisée, les règles sémantiques des Figures 3.7, 4.10, et 4.12 peuvent déjà s'appliquer pour donner une sémantique aux programmes **lus**. Pour ce faire, il suffit de réaliser un codage explicite de l'absence avec des valeurs spéciales, ajoutées au domaine de chaque variable et de chaque opération. Cette valeur est affectée à chaque variable aux cycles où elle est absente. De manière similaire, lorsque l'horloge d'une opération est fausse, celle-ci doit recevoir (respectivement produire) des valeurs absentes sur chacune de ses entrées (respectivement sorties).

Ce codage est sémantiquement correct. Un codage similaire est d'ailleurs déjà employé dans le dialecte **tf_executor** de MLIR, où il sert à représenter les boucles de calcul de spécifications TensorFlow[100].

Cependant, il est aussi inefficace, car il nécessiterait de redéfinir chaque variable réactive pour ajouter à son type la valeur explicite d'absence (e.g. à l'aide d'un booléen), ainsi que le code qui permet de le manipuler.

Pour éviter ce comportement (à des fins d'efficacité) le flot de données transmis par les valeurs $\perp$ est matérialisé sous la forme de branchements déterminant, sur la base des valeurs des variables d'horloge (e.g. `%ck` dans Fig. 5.10) quand les opérations sont exécutées ou non.

L'analyse d'horloges et l'analyse de dominance garantissent que la spécification SSA résultante ne peut pas bloquer du fait d'opérations consommant des valeurs indéfinies [3] [4] ; à ce titre, la condition requise pour la compilation correcte des valeurs indéfinies, que nous avons détaillée en Section 4.4.1, est remplie.

[3] Mais elle peut toujours bloquer du fait d'une opération indéfinie pour les valeurs fournies, e.g. division par zéro.

[4] Prouver ce résultat dépasse également le cadre de cette thèse, mais on peut se référer à [8, 9] pour la formalisation et les preuves dans la définition classique de Lustre.

```
1 func @n(%syi:in(i32),%syck:in(i1),
2         %syo:out(i32))->(){
3   %0=constant 0:i32
4   %true = constant 0:i1
5   scf.while(%si = %0): i32 {
6     scf.condition(%true)
7   } do {
8     %i = input(%syi):i32
9     %ck = input(%syck):i1
10    %so = scf.if %ck -> (i32) {
11      %o  = addi %si, %i : i32
12      output(%syo,%o):i32
13      scf.yield %o: i32
14    } else {
15      scf.yield %si: i32
16    }
17    %u = tick(%so:i32)
18    %son = sync(%so:i32,%u:none)
19    scf.yield %son: i32
20  }
21  halt
22 }
```

FIGURE 5.11 : Une fonction réactive recourant au flot de contrôle structuré de MLIR.

Exploitation de fonctionnalités MLIR. Nous avons maintenant fini de présenter la définition générale de notre méthode de compilation, mais il nous reste à évoquer les fonctionnalités de MLIR moins fondamentales dont nous tirons profit pour implémenter cette méthode. En particulier :

- **Expressivité.** Les transformations de code décrites ci-dessus produisent directement des opérations basiques de SSA (e.g. branchements). Notre implémentation réalise cette transformation en deux étapes. La première produit des opérations de contrôle structuré (dialecte **scf** de MLIR), comme `scf.while` et `scf.if`. Les transformations existantes de MLIR se chargent de réaliser la compilation ultérieure. Ainsi, le code en Fig. 5.11 est équivalent au code en Fig. 5.10 (en bas à droite).

- **Implémentation HPC.** Les types de données (e.g. tenseurs) et opérations HPC (e.g. convolutions) sont implémentés au moyen des dialectes MLIR existants, qui concentrent de larges aspects du savoir-faire

HPC. Nos cas d'études mélangent opérations réactives du dialecte **lus** avec des opérations ML du dialecte **tf** (TensorFlow) au sein des mêmes nœuds. Ces nœuds peuvent être compilés vers du code exécutable sans séparation entre contrôle et traitements ML/HPC.

- **Vérifications.** Comme nous l'avons dit précédemment, l'analyse de causalité de Lustre est entièrement assumée par l'analyse de dominance SSA (de même le réordonnancement des équations dans l'ordre topologique induit par leurs dépendances est réalisé au moyen de l'algorithmique SSA existante). De plus, les vérifications sur les types de données sont réalisées directement par le typeur de MLIR.

Toutes ces fonctionnalités, déjà disponibles, limitent grandement l'investissement nécessaire au niveau ingéniérie. *In fine*, nous avons simplement eu à implémenter les passes d'analyse et de compilation spécifiques au modèle synchrone – analyse d'horloges, normalisation, compilation de **lus** vers **sync** et **scf** et de **sync** vers les dialectes standard de MLIR.

5.3 Conclusion

Dans ce chapitre, nous avons commencé par présenter la sémantique intuitive du noyau flot de données synchrone de Lustre qui est couvert par notre extension. Cette présentation est elle-même originale, car elle remet des aspects traditionnels de la sémantique Lustre dans une perspective “compilation”, notamment par l'identification de dépendances et d'anti-dépendances (en fin de Section 5.1.1), et en identifiant les aspects de la sémantique synchrone qui ne sont pas réductibles directement à des concepts de la compilation traditionnelle. Il s'agit notamment ici de la modularité temporelle et de l'absence.

Nous avons ensuite décrit les solutions que nous proposons pour incorporer Lustre comme un dialecte de MLIR. Ce dernier point prend appui sur deux points communs majeurs entre Lustre et SSA :

- L'un et l'autre implémentent le principe SSA.
- L'un et l'autre présentent un modèle d'exécution globalement séquentiel, localement concurrent.

La compilation du nouveau dialecte **lus** se fait en deux étapes. La première consiste à traduire le contrôle réactif flot de données **lus** vers le contrôle

réactif impératif du dialecte **sync**. Ensuite, la compilation procède comme nous l'avons exposé au chapitre précédent.

L'originalité de notre approche repose sur le lien qu'elle crée entre algorithmique de compilation traditionnelle et algorithmique de compilation synchrone. Nous montrons qu'un compilateur de langage synchrone peut être construit en réutilisant largement les algorithmes de compilation traditionnelle. Les quelques algorithmes spécifiques au synchrone flot de données et incontournables sont identifiés au long du chapitre.

Chapitre 6

Intégration

Dans ce chapitre, nous présentons d'abord la manière dont fonctionne un *framework* ML et les chaînes de compilation qui y sont embarquées. Nous concentrons notre analyse sur le *framework* TensorFlow[98], dont Keras[60] est l'API haut niveau.

Nous présentons ensuite la manière dont nous étendons ce *framework*, et en particulier son infrastructure de compilation, pour réaliser la spécification et la compilation conjointes des aspects HPC et RTE d'un même ANN/DNN. La compilation d'applications réactives incorporant des traitements HPC qui ne sont pas issues du ML, comme le *vocoder* que nous présentons en Section 7.2, ne présente que de légères variations par rapport à cette méthode.

Nous illustrons finalement cette approche par la la compilation réactive d'un RNN.

6.1 Structure d'un *framework* ML

Dans cette section, nous présentons la structure d'un *framework* ML traditionnel, puis décrivons la structure du *framework* réactif que nous proposons.

6.1.1 *Frameworks* ML traditionnels (transformationnels)

Nous commençons ici par montrer comment un *framework* attribue une sémantique opérationnelle à la spécification fondamentalement *polysémique* d'un graphe ML. Nous situons ensuite le caractère transformationnel ou ré-

actif d'une telle spécification, que les *frameworks* traditionnels résolvent dans le sens du transformationnel, parmi les différentes sources de polysémie possibles. Nous présentons finalement les chaînes de compilation qui sont au cœur de tels *frameworks*.

La polysémie d'un DNN

Un DNN est fondamentalement un graphe flot de données composé de différentes couches (*layers*), dont chacune est une fonction mathématique appliquant une transformation (e.g. convolution, *pooling*) à son entrée et produisant une sortie. Un tel graphe, néanmoins, ne suffit pas à caractériser un programme doté d'une sémantique opérationnelle. Nous signalons ici deux sources importantes de polysémie dans un DNN : les poids du réseau, qui font varier la sémantique du réseau par leurs valeurs mais aussi par leur statut distinct suivant que le réseau s'exécute pour l'apprentissage ou pour l'inférence, et le *batching* [1].

Les poids du réseau Les différentes couches d'un DNN sont paramétrées par des *poids* qui matérialisent le résultat de l'apprentissage. Pour schématiser, prenons un *perceptron*[87], un réseau spécifiant une seule couche et calculant $y = act(ax + b)$; x est la valeur fournie en entrée du réseau, y la valeur qu'il doit produire en sortie, et a et b sont les poids du réseau. La fonction d'activation *act* fixe le "seuil de stimulation" des neurones, à partir duquel ils répondent (et en-deçà duquel ils ne répondent pas). Pour compléter la définition de notre réseau et clarifier l'intuition, les valeurs a, x et b sont des matrices réelles ou entières représentées sous la forme de tenseurs, et une fonction d'activation typique est $ReLU$, qui applique ponctuellement la fonction

$$ReLU(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ x & \text{sinon.} \end{cases}$$

La valeur des poids est fixée durant une phase d'apprentissage, lors de laquelle le réseau évalue la pertinence de ses calculs. Dans l'apprentissage supervisé, il consomme des données annotées : pour chaque échantillon de données, on calcule l'erreur – la distance entre ce que le réseau calcule et ce qui est attendu. Ces mesures d'erreur sont utilisées pour modifier les poids

[1] Ces deux aspects ne résument pas la polysémie d'un réseau, laquelle se matérialise aussi, par exemple, sous la forme des algorithmes d'optimisation qui paramètrent le réseau.

d'une manière qui minimise l'erreur. Le plus souvent, cette mise à jour est accomplie par rétropropagation du gradient de la fonction d'erreur[88] au travers des différentes couches du réseau [2]. Le jeu de poids peut être à tout moment enregistré sur le disque sous une forme standardisée, déchargé du réseau, ou rechargé dans le réseau.

La phase d'inférence d'un tel réseau consiste à appliquer les différentes couches du réseau, pondérées par les poids résultant de l'entraînement, sur de nouvelles données. Dans l'inférence, les poids du réseau ne varient pas, se comportant comme des constantes.

La sémantique opérationnelle d'un même graphe varie donc suivant :

- La phase (entraînement ou inférence) dans laquelle on l'exécute.
- La valuation courante des poids du réseau.

Le *batching* Le *batching* est une pratique courante dans la conception de DNNs. Il s'agit de réaliser l'entraînement du réseau (et même souvent l'inférence), non sur des échantillons pris un par un, mais sur des “paquets” de n échantillons. Ainsi, un réseau dont la taille du *batch* est 5, lors de la phase d'entraînement, mettra ses poids à jour, non pas à chaque fois qu'il aura traité un seul échantillon, mais à chaque fois qu'il en aura traité 5. Outre le parallélisme de données accru auquel une telle méthode donne accès, elle permet d'appliquer des opérations statistiques à l'échelle d'un *batch* pour accélérer l'apprentissage et améliorer sa précision. Ainsi, une couche de *batch normalization*[52] (dont l'équivalent pour un RNN est la *layer normalization*[4]) apprend notamment ses coefficients à partir de la moyenne de ses entrées à l'échelle du *batch* – lequel est donc, dès lors, partie prenante de la sémantique fonctionnelle du réseau. Ces variations sémantiques lors de la phase d'apprentissage n'ont cependant pas de conséquence sur nos cas d'études, que nous exécutons exclusivement en inférence [3].

Transformationnel vs réactif

Le graphe flot de données d'un réseau peut admettre une sémantique réactive comme une sémantique transformationnelle : cette source de polysémie est

[2] Cet algorithme nécessite donc la différentiation (automatique) des opérations composant le DNN.

[3] En inférence, la taille de *batch* comme les coefficients des couches de *batch normalization* ont déjà été fixés par l'entraînement et n'engendrent donc aucune ambiguïté.

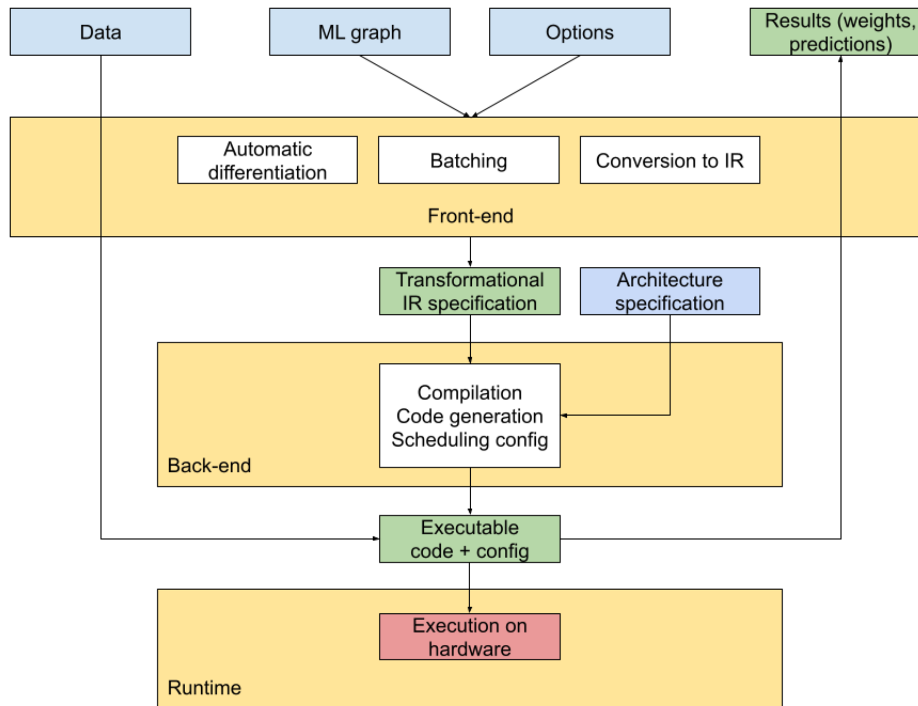


FIGURE 6.1 : L'architecture d'un *framework* ML traditionnel (transformationnel).

en fait présente depuis l'origine du paradigme flot de données, comme nous l'avons signalé en Section 1.2.4. Les *frameworks* ML traditionnels, néanmoins, lèvent cette ambiguïté en leur attribuant d'office une sémantique transformationnelle, comme l'illustre Fig. 6.1.

Au niveau du *front-end* (souvent un ensemble de bibliothèques Python), le graphe flot de données décrivant un DNN se voit attribuer une sémantique opérationnelle en fonction de ses paramètres d'utilisation : phase d'entraînement ou d'inférence, taille des *batches*, chargement d'éventuels poids pré-entraînés, etc. L'IR implémentant cette sémantique opérationnelle (souvent basée sur MLIR) est finalement générée sous la forme d'une fonction transformationnelle.

Ainsi, les deux modes fondamentaux dans lesquels on exécute un réseau

de neurones sont vus de manière transformationnelle, la fonction transformationnelle décrivant l'entraînement consommant des données annotées en entrée et produisant des poids entraînés en sortie, et la fonction transformationnelle décrivant l'inférence consommant des données non-annotées en entrées et produisant une étiquette [4] en sortie.

Au niveau du *back-end*, l'IR est compilée vers le code exécutable répondant à la spécification de l'architecture cible [5]. Dès lors, l'appel de la fonction compilée est assuré par du code externe, généralement au travers de l'API Python, chargé :

1. De lui fournir ses entrées (entrées annotées pour l'entraînement, entrées simples pour l'inférence) sous la forme d'un paquet de données unique.

2. De traiter son résultat (mise à jour des poids après une passe d'entraînement, affichage de la prédiction, etc.), lui aussi unique.

En particulier, le contrôle cyclique éventuel n'est pas spécifié au niveau du graphe, mais doit être programmé du côté du code appelant.

Chaînes de compilation

Les chaînes de compilation basées sur MLIR peuvent, dans un tel *framework*, avoir un grand nombre de cibles d'exécution. Nous nous concentrons ici sur la génération de code binaire ou de *bytecode* bas niveau pour l'exécution sur GPU ou CPU, impliquant la composition de nombreuses passes de *lowering* incrémentales depuis la spécification. Nous présentons en particulier deux chaînes de compilation (et de compilateurs) disponibles dans le *framework* TensorFlow, sur lesquelles nous nous appuierons ensuite pour définir un *framework* ML réactif. L'une cible la machine virtuelle d'IREE, et l'autre vise la génération de code purement binaire en passant par le *back-end* du compilateur LLVM. L'une et l'autre sont illustrées en Fig. 6.2.

[4] Ou un jeu d'étiquettes lorsque la taille de *batch* est supérieure à 1.

[5] Le format de code exécutable n'est pas forcément un binaire spécifique à une machine particulière. Il peut aussi être un *bytecode* spécifique à une machine virtuelle (VM), ou un mélange des deux. Ainsi, les compilateurs MLIR appartenant à l'environnement IREE[67] peuvent générer un mélange de fragments de code binaire GPU et de *bytecode* spécifiant la manière dont les fragments de code GPU sont exécutés sur le *device*.

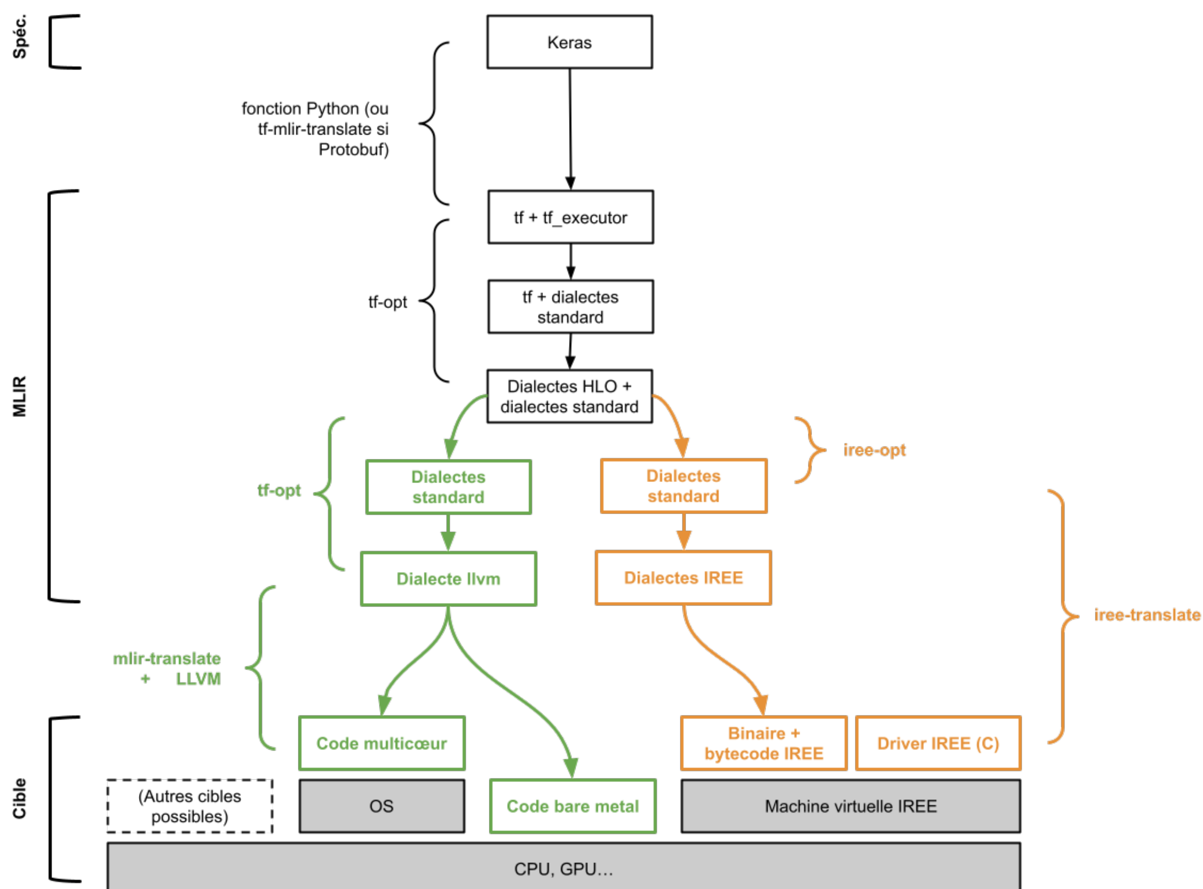


FIGURE 6.2 : Un sous-ensemble des chaînes de compilation disponibles dans le *framework* TensorFlow. En vert, la chaîne de compilation ciblant la génération de code purement binaire en passant par le *back-end* de LLVM. En orange, la chaîne de compilation ciblant la machine virtuelle IREE.

Compilateurs. Les compilateurs MLIR utilisés dans les chaînes de compilation que nous présentons sont distribués sous la forme d'utilitaires en ligne de commandes (mais certains d'entre eux sont également accessibles au travers de *bindings* Python). Ils sont au nombre de cinq :

- **tf-mlir-translate** permet de convertir un fichier en format Protobuf[82] [6] vers une spécification MLIR en dialectes **tf** et **tf_executor**. Le dialecte **tf** décrit les opérations TensorFlow de traitement des données (e.g. convolution), et le dialecte **tf_executor** fournit les primitives de contrôle utilisées par le *runtime* standard de TensorFlow (que nous n'avons pas représenté en Fig. 6.2).

- **tf-opt** est une extension de mlir-opt, le compilateur MLIR distribué par le projet LLVM [7], aux dialectes et transformations nécessaires pour réaliser le *lowering* d'une spécification TensorFlow. En effet, tf-opt permet :

  - de convertir les opérations du dialecte **tf_executor** vers les primitives de contrôle des dialectes standard de MLIR.
  - de convertir les opérations du dialecte **tf** vers les opérations de dialectes ML indépendants de la spécification haut niveau. Il peut s'agir du dialecte **tosa**, mais en Fig. 6.2, il s'agit des dialectes de la famille HLO (e.g. **mhlo**). Ces derniers sont initialement l'implémentation MLIR du format d'entrée du compilateur utilisé par TensorFlow pour l'algèbre linéaire (XLA), qui est en cours de migration vers MLIR [8], mais on les retrouve désormais dans des outils ne visant pas nécessairement l'utilisation de XLA (e.g. mlir-hlo-opt) ou dans d'autres *frameworks* que TensorFlow (e.g. JAX).
  - de traiter les dialectes standard définis dans mlir-opt, et en particulier de réaliser les passes de *lowering* bas niveau.

[6]Le format Protobuf est un format de sérialization d'une spécification ML utilisé par le *framework* TensorFlow.

[7]Les dialectes définis dans mlir-opt sont ceux que nous désignons comme les dialectes standard de MLIR.

[8]Notons à cet égard que, dans tf-opt, les transformations des dialectes haut niveau sont définies dans les deux directions. Ainsi, un fichier en format Protobuf peut être importé dans MLIR, compilé jusqu'aux dialectes HLO en passant par **tf** et **tf_executor**, faire l'objet de passes d'optimisations à ce niveau, puis être à nouveau converties vers **tf** et **tf_executor** et finalement exportées vers le format Protobuf.

- **iree-opt et iree-translate** appartiennent à l'environnement IREE (*Intermediate Representation Execution Environment*)[67], qui est formé de compilateurs MLIR, de bibliothèques C et d'une machine virtuelle distribuée sous la forme de différents *runtimes*. Il s'agit d'un *pipeline* spécialisé, conçu pour faciliter le déploiement de modèles ML sur une grande variété de cibles matérielles. Étant donnée une spécification en dialecte **mhlo** (ou **tosa**), iree-opt et iree-translate peuvent générer du code exécutable par la machine virtuelle d'IREE.
- **mlir-translate** permet d'exporter un code en dialecte **llvm** vers le format LLVM IR utilisé par les compilateurs basés sur LLVM (llc, clang, etc.).

Nous décrivons maintenant la manière dont ces compilateurs sont composés les uns avec les autres.

Importation de la spécification. La spécification Keras est d'abord convertie en fonction(s) MLIR, soit directement depuis l'API Python, soit en étant d'abord *sérializée* en format Protobuf puis importée dans MLIR par tf-mlir-translate. Le code généré, transformationnel, est un mélange des dialectes **tf** pour le traitement des données et **tf_executor** pour le contrôle.

La première étape consiste à éliminer les opérations du dialecte *tf_executor*, qui n'est pas conçu pour la compilation vers les dialectes de bas niveau. Cet ensemble de passes est réalisé par l'option `tf-standard-pipeline` de tf-opt.

Au terme de ces transformations, le traitement des données est toujours réalisé par les opérations du dialecte **tf**, mais le contrôle est désormais assuré par les opérations issues des dialectes standard de MLIR.

Le passage par les dialectes HLO. Toujours au moyen de tf-opt, les opérations du dialecte **tf** pour le traitement des données sont ensuite converties vers les dialectes de la famille HLO, au moyen de l'option `xla-legalize-tf` [9]. À partir de ce point, deux chaînes de compilation distinctes se forment, l'une ciblant le *back-end* LLVM et l'autre ciblant la machine virtuelle d'IREE.

La génération de code ciblant le *back-end* LLVM. Le compilateur tf-opt offre la possibilité de ramener les opérations des dialectes

[9] Celle-ci peut néanmoins nécessiter des transformations préliminaires, comme `tensor-list-opts-decomposition` pour traiter le type TensorList.

HLO aux opérations du dialecte standard **linalg**, au moyen de l'option `hlo-legalize-to-linalg`. Une fois cette passe accomplie, les transformations de code nécessaires pour convertir la totalité du code vers les opérations du dialecte **llvm** sont celles qui sont définies dans le compilateur mlir-opt. Il est nécessaire de les composer manuellement, ce qui implique de contrôler précisément chaque passe – y compris les passes d'optimisation (e.g. *tiling*, vectorisation) –, et peut mener à une ligne de commandes composée de plusieurs dizaines d'options. Nous ne les énumérons pas ici.

Une fois le code en dialecte **llvm**, il est possible d'appeler mlir-translate, qui l'exporte vers la représentation intermédiaire utilisée par le *back-end* de LLVM (LLVM IR), et en particulier les compilateurs llc et clang. La fin de la compilation est réalisée par ces derniers. Le code objet résultant peut ensuite faire l'objet d'une étape de *link* pour qu'il puisse être appelé par d'autres binaires et lui-même appeler des fonctions externes.

L'interfaçage MLIR/C implique néanmoins de comprendre les conventions d'appel de MLIR, et en particulier la représentation mémoire des *memrefs* qui décrivent la mémoire persistante en MLIR. Ces derniers ne sont pas réductibles à de simple pointeurs C mais sont l'abstraction de structures de données plus complexes (décrites en C au moyen de `struct`). L'exemple en Fig. 6.3, décrivant un *memref* de trois dimensions dont les éléments sont des entiers de 32 bits, permet ainsi de constater qu'un *buffer* agrège :

1. Un pointeur sur le début de la zone mémoire allouée (ligne 2).
2. Un pointeur sur le début de la zone mémoire alignée (ligne 3).
3. L'offset entre zone mémoire allouée et zone mémoire alignée, en général 0 (ligne 4).
4. La taille de chacune de ses dimensions (ligne 5).
5. Le *pas* entre les valeurs à l'intérieur de chaque dimension, en général 1 (ligne 6).

La génération de code ciblant la machine virtuelle d'IREE. Le code en dialecte **mhlo** peut également alimenter le compilateur IREE iree-opt [10].

[10]En fait, il peut même alimenter directement iree-translate, mais iree-opt offre davantage de possibilités de transformation du code haut niveau.

```
struct memref_3d_i32 {
  int32_t* allocated ;
  int32_t* aligned ;
  intptr_t offset ;
  intptr_t size[3] ;
  intptr_t stride[3] ;
};
```

FIGURE 6.3 : La représentation C d'un buffer MLIR de trois dimensions dont les éléments sont des entiers de 32 bits.

Ici, ce dernier réalise un pré-traitement du code [11] puis réalise son *lowering* vers les dialectes standard de MLIR, essentiellement **linalg** [12].

Le code résultant est ensuite traité par iree-translate qui, étant donnée la spécification de l'architecture cible [13], convertit la spécification en code exécutable [14], qui peut faire l'objet d'une dernière passe d'optimisation [15].

Par comparaison avec la chaîne de compilation précédente, il n'est pas nécessaire de composer manuellement les transformations de code, bien qu'une telle approche soit également possible.

Il faut cependant noter qu'iree-translate impose des limitations qui ne sont pas communes à tous les compilateurs MLIR, et en particulier qui ne sont pas exigées par tf-opt et mlir-opt. Par exemple, la version du compilateur que nous avons utilisé interdit l'appel à des fonctions externes, ou encore l'utilisation de variables globales. Ces limitations joueront un rôle important dans le choix des outils pour la définition de nos chaînes de compilation réactives.

Le code exécutable généré par iree-translate est un binaire englobé dans du *bytecode* IREE. Ce *bytecode* peut ensuite être appelé, soit par un *runtime* distribué par IREE[16], soit par un *driver* C implémenté par ailleurs et utilisant les fonctions de bibliothèque IREE [17]

[11]Option `iree-mhlo-to-mhlo-preprocessing`.

[12]Option `iree-mhlo-input-transformation-pipeline`.

[13]Option `iree-hal-target-backends=[target]`.

[14]Option `iree-mlir-to-vm-bytecode-module`.

[15]Option `iree-vm-bytecode-module-optimize`.

[16]Accessible en lignes de commandes, mais aussi au travers de *bindings* Python.

[17]À ce niveau, notons qu'il est également possible d'isoler le binaire du *bytecode* pour viser l'exécution *bare-metal*.

6.1.2 *Framework* ML réactif

Le *framework* réactif que nous proposons consiste fondamentalement à permettre de lever la polysémie transformationnel/réactif de la spécification flot de données d'un DNN dans le sens inverse de celui que choisissent les *frameworks* traditionnels, en attribuant une sémantique réactive à de tels réseaux. Le transformationnel est alors vu comme un cas particulier du réactif. Nous réalisons ici cette extension d'un *framework* traditionnel par l'extension de son infrastructure de compilation.

Vision d'ensemble

Ainsi, une IR réactive est générée à partir de la spécification et est ensuite compilée par la chaîne de compilation traditionnelle étendue à la compilation des opérateurs réactifs. Le code exécutable résultant reçoit ses entrées à chaque cycle (typiquement, dans un système RTE, des senseurs du système) et envoie ses sorties à chaque cycle (typiquement, dans un système RTE, aux actuateurs du système). L'architecture que nous propons est illustrée en Fig. 6.4.

Le *framework* réactif que nous présentons ici, au-delà du caractère transformationnel ou réactif du réseau, lève d'emblée la polysémie relative à la phase d'apprentissage ou d'inférence. En effet, seule la phase d'inférence y est disponible (les poids sont supposés pré-entraînés), car il faut, pour pouvoir implémenter la phase d'apprentissage, définir et implémenter les règles de différentiation automatique appliquées aux opérateurs du réseau. La définition de telles règles pour les opérateurs flot de données synchrone vus dans le Chapitre 5 est un travail en cours.

Par conséquent, le *batching* ne pose pas de problème. Lors de l'inférence, il s'agit simplement de traiter une dimension supplémentaire des données et un sous-ensemble réduit d'opérations supplémentaires (e.g. *batch normalization*).

Chaînes de compilation

Pour réaliser la compilation d'un DNN réactif, nous étendons les chaînes de compilation évoquées précédemment afin d'exploiter au maximum le savoir-faire qui y est matérialisé. Nous insérons en particulier deux outils dans le *framework* TensorFlow :

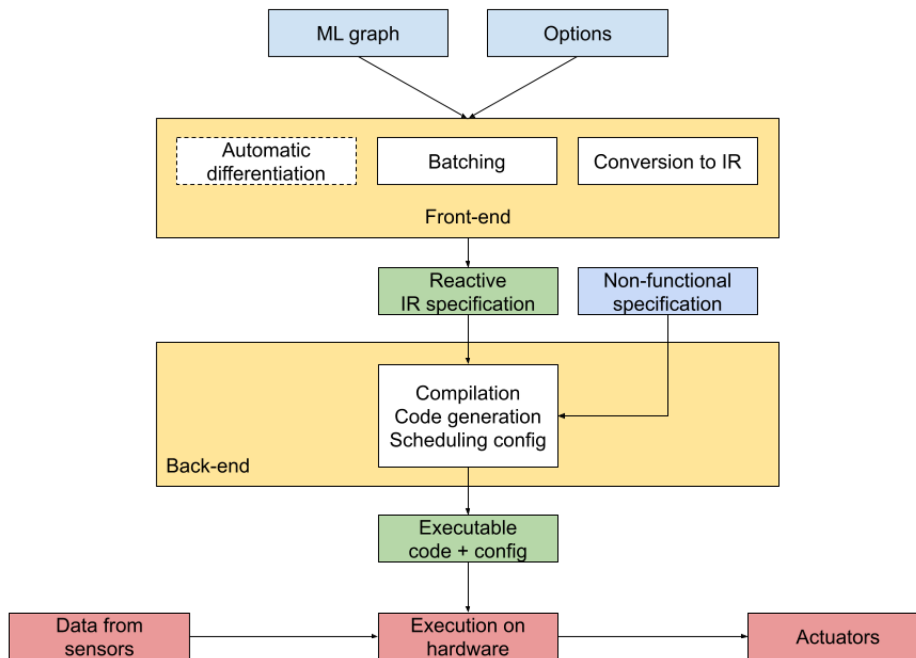


FIGURE 6.4 : L'architecture d'un *framework* ML réactif. La différentiation automatique d'une spécification réactive est un travail en cours.

- **keras-lus**, une bibliothèque Python programmée par nos soins, se substitue à l'API Keras. Elle génère ainsi une IR réactive à partir de la même spécification Keras. Cette IR réactive est composée du dialecte MLIR **lus** pour le contrôle réactif et du dialecte **tf** pour le traitement HPC des données.

- **mlir-lus**, que nous avons déjà présenté, étend mlir-opt aux dialectes **lus** et **sync** et aux transformations de code vues en Chapitre 4 et Chapitre 5.

Fig. 6.5 illustre cette approche.

Le compilateur mlir-lus, après avoir normalisé les nœuds **lus** générés par keras-lus, peut les compiler en suivant l'une ou l'autre des approches de compilation réactive que nous avons présenté précédemment :

1. Sous la forme de processus communiquant avec leur environnement externe et avec d'autres nœuds processus.

2. Sous la forme de couples de fonctions step/reset appelés par des *drivers* dédiés.

Une fois réalisée la compilation du contrôle réactif, ce *framework* réactif réutilise entièrement les chaînes de compilation du *framework* transformationnel qu'il étend.

Signalons que nous nous concentrons ici sur la compilation de l'IR générée par keras-lus, mais le compilateur mlir-lus peut naturellement traiter des spécifications dont les aspects HMPC/ML ne sont pas liés à Keras.

Options du compilateur mlir-lus. La ligne de commande appelant mlir-lus peut prendre 4 options différentes, chacune activant une étape de compilation distincte, ce qui permet de composer nos deux chaînes complètes de génération de code, mais aussi d'appliquer ces transformations séparément pour pouvoir tracer leur entrée et leur sortie :

1. L'option `normalize`, qui réalise les opérations de normalisation décrites en Section 5.2.2 : les opérations `lus.fby` qui ne sont pas sur l'horloge de base du nœud y sont placées, après quoi toutes les opérations `lus.fby` sont éliminées (l'état d'un nœud est alors représenté dans son interface et dans son opération terminale). Finalement, les opérations du nœud sont ordonnées suivant les dépendances des variables.

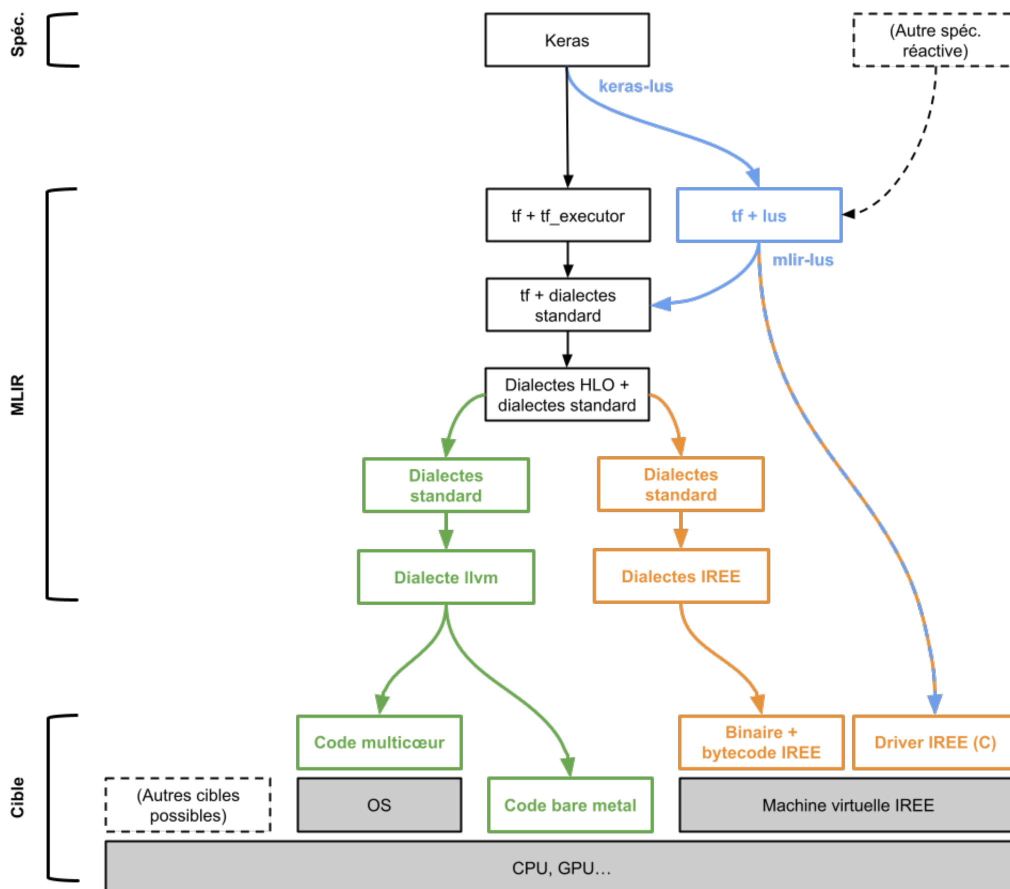


FIGURE 6.5 : L'extension réactive des chaînes de compilations présentées en Fig. 6.2. En bleu, notre extension. En vert, la chaîne de compilation ciblant la génération de code purement binaire en passant par le *back-end* de LLVM. En orange, la chaîne de compilation ciblant la machine virtuelle IREE. L'arc coloré en pointillés bleu et orange représente une fonctionnalité non encore incorporée dans mlir-lus : la génération du *driver* C utilisé pour appeler les fonctions step/reset lorsqu'on cible la machine virtuelle IREE.

2. L'option `to-sync-automata`, qui convertit chaque nœud **lus** en un processus communiquant. La boucle principale et l'exécution conditionnelle sont implémentées au moyen du dialecte standard **scf** (opérations `scf.while` et `scf.if`). Les signaux d'entrée-sortie, leur lecture et leur écriture, la barrière de cycle, l'instanciation d'un nœud et les valeurs indéfinies sont représentés par les opérations du dialecte **sync**.

3. L'option `invert-control`, qui convertit chaque nœud **lus** normalisé en une paire de fonctions step et reset. L'état du nœud est représenté dans les entrées et les sorties de la fonction step, et le contrôle réactif est assumé par le *driver* externe appellant les fonctions step et reset. En l'état de notre compilateur, cette passe présente deux limites d'implémentation :

    - Au lieu de représenter l'instanciation d'une fonction réactive par l'appel à la fonction step correspondante, nous réalisons l'*inlining* préalable des nœuds **lus** instanciés.
    - Nous ne générons pas automatiquement le *driver* C utilisant les fonctions de bibliothèque d'IREE et chargé d'appeler le code MLIR des fonctions step/reset. En l'état, ce dernier doit donc être programmé manuellement.

    Comme dans l'approche précédente, l'exécution conditionelle est implémentée par `scf.if`. La seule opération du dialecte **sync** qui apparaît est `sync.undef`. Cette option et la précédente sont mutuellement exclusives.

4. L'option `sync-to-std`, qui réalise le *lowering* des opérations du dialecte **sync** vers les dialectes de base de MLIR.

L'option `allow-unregistered-dialect`, définie dans le compilateur mlir-opt que nous étendons, est activée d'emblée dans mlir-lus. Cette dernière indique au compilateur qu'ils ne doit pas rejeter les spécifications où figurent des dialectes non-définis, comme c'est le cas du dialecte **tf** (mlir-lus étend le compilateur mlir-opt aux dialectes **lus** et **sync**, non au dialecte **tf**). Les opérations concernées doivent cependant obéir à certaines règles syntaxiques : elles doivent être écrites entre guillemets et leurs annotations de type doivent imiter celles des appels de fonction.

Compilation des nœuds lus vers des processus communiquants. Du fait des limitations propres à IREE évoquées précédemment, cette méthode, qui recourt à des fonctions externes, doit cibler le *back-end* de LLVM sans passer par iree-translate.

Fig. 6.6 expose ainsi une ligne de commandes possible pour compiler le RNN très simple qui sera présenté en Section 6.2 (Fig. 6.8) en suivant cette méthode. En ligne 1, keras-lus génère le code **lus**/**tf**. En ligne 2, le contrôle réactif est compilé vers les dialectes standard de MLIR, sans toucher au traitement des données par le dialecte **tf**. À partir de ce point, on peut utiliser une chaîne de compilation pré-existante. En ligne 3, les opérations du dialecte **tf** sont compilées vers les opérations du dialecte **mhlo**, qui sont elles-mêmes ramenées au dialecte **linalg** en ligne 4.

Les lignes 5, 6, 7 et 9 réalisent la synthèse de *buffers* (allocation et libération de la mémoire persistante) à partir de la représentation tensorielle de haut niveau. Bien que la compilation de langages synchrones repose usuellement sur l'allocation entièrement statique des ressources mémoire, nous mettons en œuvre un compromis différent. Si l'allocation et la libération des *buffers* implémentant les entrée-sorties d'une fonction sont réalisées par l'appellant (combinaison des options `func-bufferize` et `buffer-results-to-out-params`), notre stratégie d'allocation demeure dynamique en cela que les opérations d'allocation et de libération sont réalisées dans le corps de la boucle principale de chaque fonction réactive. Nous n'avons pas exploré[18] les passes de mlir-opt déclenchées par les options `buffer-hoisting` et `buffer-loop-hoisting` sur lesquelles aurait pu reposer une stratégie plus résolument statique.

Les lignes 8 et 10 réalisent le *lowering* des dialectes **linalg**, **affine**, **scf** de MLIR. Les lignes 11 et 12 génèrent un code en dialecte **llvm**, qui est exporté vers LLVM IR en ligne 14. Les options `canonicalize` en lignes 4 et 13 décrivent simplement les optimisations minimales de MLIR (e.g. propagation de constantes) : les optimisations plus avancées (e.g. *tiling*, vectorisation) dépendent d'options dédiées.

Compilation des nœuds lus vers des fonctions step/reset. La méthode de compilation vers des fonctions step/reset n'utilisant pas de fonctions externes, elle peut viser la machine virtuelle IREE.

[18] Par manque de temps, ce qui a impliqué de faire des choix. Ici, nous avons choisi d'explorer des méthodes d'allocation plus proches du ML traditionnel.

```
python rnn.py | \
  mlirlus --normalize --to-sync-automata --sync-to-std | \
  tf-opt --xla-legalize-tf=allow-partial-conversion \
    --hlo-legalize-to-linalg --canonicalize \
    --func-bufferize --buffer-results-to-out-params \
    --tensor-constant-bufferize --linalg-bufferize \
    --tensor-bufferize \
    --convert-linalg-to-affine-loops --lower-affine \
    --scf-bufferize --buffer-deallocation \
    --convert-scf-to-std --std-expand \
    --convert-memref-to-llvm --convert-math-to-llvm \
    --convert-arith-to-llvm --convert-std-to-llvm \
    --canonicalize | \
  mlir-translate --mlir-to-llvmir > rnn.llvmir
```

FIGURE 6.6 : Une ligne de commandes compilant le RNN spécifié en Fig. 6.8 en processus communiquant, vers le *back-end* LLVM.

```
python rnn.py | \
  mlirlus --normalize --invert-control --sync-to-std | \
  tf-opt -xla-legalize-tf=allow-partial-conversion | \
  iree-opt --iree-mhlo-to-mhlo-preprocessing \
    --iree-mhlo-input-transformation-pipeline | \
  iree-translate --iree-hal-target-backends=cuda \
    --iree-mlir-to-vm-bytecode-module \
    --iree-vm-bytecode-module-optimize > rnn.vmfb
```

FIGURE 6.7 : Une ligne de commandes compilant le RNN spécifié en Fig. 6.8 en fonctions step/reset, vers la machine virtuelle d'IREE.

Fig. 6.7 expose une ligne de commandes possible pour compiler le même RNN en suivant cette méthode. La ligne 2 génère, non plus un processus communiquant, mais un couple de fonctions step/reset. En lignes 4 et 5, on convertit les opérations du dialecte **mhlo** vers les dialectes standard de MLIR. En ligne 6, on spécifie l'architecture cible (ici, il s'agit d'un GPU). En ligne 7, on génère le code exécutable, auquel on applique une dernière passe d'optimisation en ligne 8.

L'allocation et la libération de la mémoire persistante sont ici déléguées à IREE et suivent donc une stratégie entièrement dynamique.

```
1  input = keras.Input(shape=(3,40),batch_size=1)
2  output = layers.LSTM(units=4,
3                       activation='tanh',
4                       recurrent_activation='softmax')(input)
5  model = keras.Model(input, output)
```

FIGURE 6.8 : La spécification Keras d'un RNN contenant une seule couche LSTM.

6.2 Transformations de code

Dans cette section, nous mettons en œuvre la chaîne d'outils que nous avons présentée en compilant un RNN simple, contenant une seule couche LSTM. Nous décrivons d'abord ce qu'est une couche LSTM, puis montrons la spécification MLIR que le *framework* TensorFlow génère à partir de la spécification Keras. Enfin, nous montrons la représentation réactive de ce RNN et détaillons les étapes de sa compilation réactive.

6.2.1 Description de l'exemple : LSTM

La couche récurrente LSTM est spécifiée, en Keras, comme n'importe quelle autre couche (convolution, etc.). Fig. 6.8 est ainsi la spécification d'un RNN contenant une seule couche LSTM et produisant, en sortie, un vecteur de 4 éléments (paramètre `units`, ligne 2). À ce niveau, un indice de l'approche transformationnelle du *framework* TensorFlow apparaît déjà : l'axe du temps est simplement spécifié comme la dimension externe des données d'entrée (ligne 1 : le réseau consomme un horizon temporel de longueur 3). Les paramètres `activation` et `recurrent_activation` (lignes 3 et 4) désignent des fonctions d'activation sur lesquelles nous reviendrons, car ce qui apparaît à haut niveau comme une simple couche compose en réalité plusieurs calculs.

Nous présentons maintenant ces derniers.

Long Short-Term Memory. Une couche LSTM, dont le flot de données est représenté en Fig. 6.9, traite des données d'entrée ordonnées dans le temps en les pondérant au moyen de ses deux mémoires internes :

- Une mémoire à long terme (LTM, *Long-Term Memory*), traditionnellement appelée *cell state*, qui représente l'état courant du calcul le long de l'axe du temps.

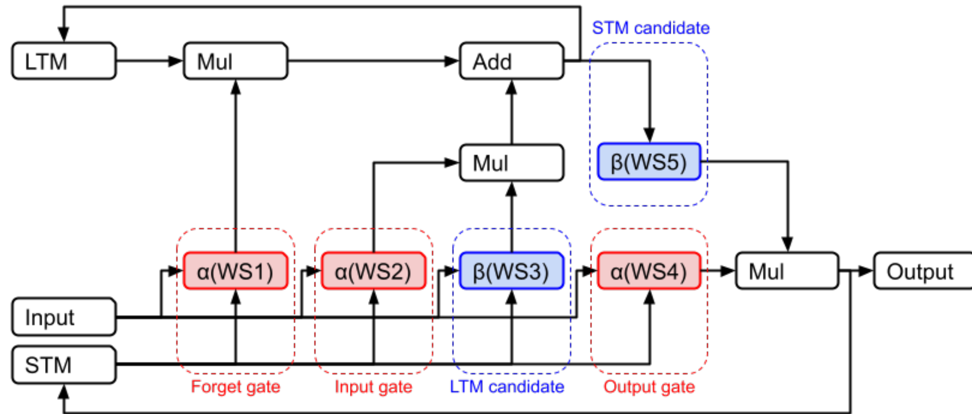


FIGURE 6.9 : Le flot de données d'une couche LSTM.

- Une mémoire à court terme (STM, *Short-Term Memory*), traditionnellement appelée *hidden state*, qui représente le résultat du calcul précédant immédiatement le calcul courant sur l'axe du temps. La valeur de la STM est non seulement transmise d'un pas de temps au suivant, mais elle est également retournée comme le résultat du LSTM.

Ces deux mémoires sont initialisées d'après des valeurs apprises durant la phase d'entraînement du réseau. Nous verrons que dans l'approche transformationnelle de TensorFlow, elles sont aussi réinitialisées à chaque fois qu'une séquence temporelle s'achève.

Le calcul avance en faisant passer l'entrée courante et la STM au travers de trois *portes* (Fig. 6.9, en rouge), qui sont autant de sommes pondérées (notées WS en Fig. 6.9) par des poids appris lors de l'entraînement. Chaque porte fait l'objet d'une même fonction d'activation, spécifiée au moyen du paramètre `recurrent_activation` vu en Fig. 6.8 et noté α en Fig. 6.9.

La production de valeurs *candidates* pour la mise à jour des mémoires est contrôlée par deux autres WSs (en bleu) faisant l'objet d'une autre fonction d'activation spécifiée au moyen du paramètre `activation` vu en Fig. 6.8 et

noté β en Fig. 6.9 [19].

Les trois portes d'un LSTM sont :

1. Une porte d'*oubli* contrôlant les informations de l'état passé de la LTM qui doivent être oubliées.
2. Une porte d'*entrée* contrôlant les informations candidates qui doivent effectivement être écrites dans la LTM.
3. Une porte de *sortie* contrôlant les informations candidate qui doivent effectivement être écrites dans la STM (et dans la sortie du LSTM).

6.2.2 Génération de code transformationnelle

Lorsque le *framework* TensorFlow génère le code MLIR correspondant à la spécification Keras en Fig. 6.8, ce qui n'était alors qu'un simple indice transformationnel est entièrement matérialisé. En effet, considérons le code en Fig. 6.10. Le LSTM est alors représenté sous la forme d'une fonction acquerrant en une fois la totalité des entrées correspondant à l'horizon de temps de 3 cycles défini précédemment (ligne 1), réalisant la totalité des calculs correspondant à cet horizon de temps (lignes 7 à 25) et produisant finalement sa sortie, la dernière valeur prise par la STM (ligne 26). Le contrôle cyclique du LSTM est représenté sous la forme d'une boucle itérant sur le tenseur d'entrée (ligne 7). La LTM et la STM sont passées d'un cycle au suivant sous la forme de dépendances portées par la boucle (lignes 8, 23, 24). Elles sont donc réinitialisées à chaque fois que la fonction retourne et qu'elle est rappelée.

Signalons que e code que nous présentons a été réarrangé par nos soins. En effet, la sortie brute de la génération de code, dans le contexte de cet exposé, pose ici une limite de place et de lisibilité : même lorsque les opérations du dialecte **tf_executor** ont été éliminées, elle contient près d'une centaine de lignes, dont certaines sont très larges (près de 1000 caractères). En particulier :

- Nous avons représenté la séquence d'entrée sous la forme d'un simple tenseur en lieu et place de la structure de données dédiée (`TensorList`) utilisée par TensorFlow et faisant l'objet d'opérations d'accès complexes.

[19] Signalons néanmoins que TensorFlow remplace la somme pondérée produisant la valeur candidate pour la STM par la fonction identité.

```
1  func @lstm (%serie:tensor<3x40xf32>)->(tensor<1x4xf32>) {
2   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
3   %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
4   %zero = arith.constant 0: i32
5   %size = arith.constant 3: i32
6   %one = arith.constant 1: i32
7   %s:2 = scf.for %i = %zero to %size step %one
8    iter_args(%STM=%STM0,%LTM=%LTM0) {
9    %input = tensor.extract_slice %serie[%i,0][1,40][1,1]: tensor<1x40xf32>
10   %34 = call @WS1(%input, %STM): tensor<1x4xf32>
11   %35 = tf.Softmax(%34): tensor<1x4xf32>
12   %47 = call @WS2(%input, %STM): tensor<1x4xf32>
13   %48 = tf.Softmax(%47): tensor<1x4xf32>
14   %25 = call @WS3(%input, %STM): tensor<1x4xf32>
15   %26 = tf.Softmax(%25): tensor<1x4xf32>
16   %56 = call @WS4(%input, %STM): tensor<1x4xf32>
17   %57 = tf.Tanh(%56): tensor<1x4xf32>
18   %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
19   %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
20   %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
21   %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
22   %output = tf.Mul(%26,%60): tensor<1x4xf32>
23   scf.yield (%output: tensor<1x4xf32>,
24              %LTM_up: tensor<1x4xf32>)
25   }
26  return %s#0: tensor<1x4xf32>
27 }
```

FIGURE 6.10 : La transcription (très) simplifiée du code MLIR généré par la chaîne TensorFlow standard pour représenter une couche LSTM.

```
1  func @WSX(%input: tensor<1x40xf32>,
2            %STM: tensor<1x40xf32>) -> (tensor<1x4xf32>) {
3    %kern = tf.Const{dense<XXX>}: tensor<40x4xf32>
4    %bias = tf.Const{dense<XXX>}: tensor<4xf32>
5    %rec_kern = tf.Const{dense<XXX>}: tensor<4x4xf32>
6    %v0 = tf.MatMul(%input, %kern): tensor<1x4xf32>
7    %v1 = tf.BiasAdd(%v0, %bias): tensor<1x4xf32>
8    %v2 = tf.MatMul(%STM, %rec_kern): tensor<1x4xf32>
9    %v3 = tf.AddV2(%v1, %v2): tensor<1x4xf32>
10   return %v3: tensor<1x4xf32>
11 }
```

FIGURE 6.11 : Le modèle des fonctions calculant les sommes pondérées dans le LSTM en Fig. 6.10.

- Nous avons représenté la boucle itérant sur les entrées sous la forme d'une simple opération `scf.for`, et non du `tf.While` plus complexe utilisé par TensorFlow.
- Le compilateur tf-opt génère nativement les opérations du dialecte **tf** sous la forme d'opérations inconnues pour faciliter l'interfaçage avec d'autres compilateurs. Cependant, la notation de ces dernières alourdit la lecture du code, nous nous en sommes donc passé. Dans le même ordre d'idée, nous avons allégé l'annotation des types.

Bien que le *framework* ne produise pas ce genre de modularité, nous avons également représenté les sommes pondérées par des appels de fonction. Les fonctions WS1, ..., WS4 sont des instances particulières du modèle en Fig. 6.11. Les poids des sommes pondérées y sont encapsulés, sous la forme de constantes, et sont différents à chaque fois. Nous avons masqué la valeur de ces constantes par "XXX".

6.2.3 Génération de code réactive

À partir de la même spécification Keras, nous aurions pu choisir de générer un code MLIR/lus plus proche de l'intuition réactive de Fig. 6.9, à la manière de kws-streaming (abstraction faite des opérations de traitement de signal liées au cas d'étude *Key Word Spotting*). Une telle approche produirait le code en Fig. 6.12. Le flot de données encapsulé dans le nœud `lus` est très proche de celui de Fig. 6.10 (le bloc central du code est identique), mais dans cette spécification, les échantillons d'entrée arrivent un par un (ligne 1) et une sortie

```
1   lus.node @lstm(%input: tensor<1x40xf32)->(tensor<1x4xf32>) {
2    %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
3    %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
4    %STM = lus.fby %STM0 %output: tensor<1x4xf32>
5    %LTM = lus.fby %LTM0 %LTM_up: tensor<1x4xf32>
6    %34 = call @WS1(%input, %STM): tensor<1x4xf32>
7    %35 = tf.Softmax(%34): tensor<1x4xf32>
8    %47 = call @WS2(%input, %STM): tensor<1x4xf32>
9    %48 = tf.Softmax(%47): tensor<1x4xf32>
10   %25 = call @WS3(%input, %STM): tensor<1x4xf32>
11   %26 = tf.Softmax(%25): tensor<1x4xf32>
12   %56 = call @WS4(%input, %STM): tensor<1x4xf32>
13   %57 = tf.Tanh(%56): tensor<1x4xf32>
14   %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
15   %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
16   %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
17   %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
18   %output = tf.Mul(%26,%60): tensor<1x4xf32>
19   lus.yield(%output:tensor<1x4xf32>)
20  }
```

FIGURE 6.12 : Une spécification réactive de LSTM inspirée de kws-streaming.

est produite à chaque cycle (ligne 19). La LTM et la STM sont maintenues par les opérations `fby` des lignes 4 et 5 ne sont jamais réinitialisées.

Cependant, une telle spécification décrit un autre calcul que celle générée par l'approche traditionnelle de TensorFlow (elle ne réinitialise jamais les mémoires et produit un résultat à chaque cycle). Nous avons donc plutôt choisi de spécifier et de compiler de manière réactive un LSTM produisant les mêmes résultats que l'implémentation de TensorFlow.

Considérons ainsi le code MLIR/lus en Fig. 6.13 [20]. Là aussi, les échantillons d'entrée arrivent et son traités à chaque cycle. Mais par ailleurs, cette spécification recouvre le comportement périodique du code généré par TensorFlow en Fig. 6.10 : le booléen `%clk`, Vrai tous les trois cycles, est utilisé pour réinitialiser la LTM et la STM au moyen d'opérations `select` (lignes 6 et 9) et pour conditionner la conditions des sorties (ligne 23).

[20] Ce dernier diffère légèrement de celui que génère effectivement keras-lus : les opérations inconnues ne sont pas présentées comme telles, les poids du réseau sont masqués au moyen de "XXX", les annotations de type sont allégées et la modularité nous permet de réduire le code du nœud

```
1  lus.node @lstm(%input: tensor<1x40xf32>)->(tensor<1x4xf32>) {
2   %period = tf.Const{dense<3>}: tensor<i32>
3   %clk = lus.instance @time_tick(%period): (i32)->(i1)
4   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
5   %STM1 = lus.fby %STM0 %STM_up: tensor<1x4xf32>
6   %STM = select %clk, %STM0, %STM1: tensor<1x4xf32>
7   %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
8   %LTM1 = lus.fby %LTM0 %LTM_up: tensor<1x4xf32>
9   %LTM = select %clk, %LTM0, %LTM1: tensor<1x4xf32>
10  %34 = call @WS1(%input, %STM): tensor<1x4xf32>
11  %35 = tf.Softmax(%34): tensor<1x4xf32>
12  %47 = call @WS2(%input, %STM): tensor<1x4xf32>
13  %48 = tf.Softmax(%47): tensor<1x4xf32>
14  %25 = call @WS3(%input, %STM): tensor<1x4xf32>
15  %26 = tf.Softmax(%25): tensor<1x4xf32>
16  %56 = call @WS4(%input, %STM): tensor<1x4xf32>
17  %57 = tf.Tanh(%56): tensor<1x4xf32>
18  %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
19  %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
20  %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
21  %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
22  %STM_up = tf.Mul(%26,%60): tensor<1x4xf32>
23  %output = lus.when %clk %STM_up: tensor<1x4xf32>
24  lus.yield(%output:tensor<1x4xf32>)
25 }
```

FIGURE 6.13 : Une spécification réactive de LSTM reproduisant le comportement du code généré par TensorFlow en Fig. 6.10.

Pour que nous puissions illustrer la compilation de l'instanciation d'un nœud, ce booléen n'est pas défini par une opération `lus.kperiodic (001)`, mais produit par l'instanciation du nœud `time_tick` en ligne 3.

Normalisation

La spécification en Fig. 6.13 peut ensuite faire l'objet de la passe de normalisation décrite en Section 5.2.2 et implémentée dans mlir-lus. Son résultat est exposé en Fig. 6.14. Les opérateurs `fby` ont disparu. L'état du nœud (i.e. la STM et la LTM) est représenté dans son interface (ligne 2) et dans son opération terminale (ligne 28). L'horloge k-périodique produite en ligne 4 permet de l'initialiser lors du premier cycle (lignes 8 et 11).

```
1  lus.node @lstm(%input: tensor<1x40xf32>)
2    state(%STM1:tensor<1x4xf32>,%LTM1:tensor<1x4xf32>)
3    ->(tensor<1x4xf32>) {
4   %f = lus.kperiodic 1(0)
5   %period = tf.Const{dense<3>}: tensor<i32>
6   %clk = lus.instance @time_tick(%period): (i32)->(i1)
7   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
8   %STM2 = lus.merge %f,%STM0,%STM1:tensor<1x4xf32>
9   %STM = select %clk, %STM0, %STM2: tensor<1x4xf32>
10  %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
11  %LTM2 = lus.merge %f,%LTM0,%LTM1:tensor<1x4xf32>
12  %LTM = select %clk, %LTM0, %LTM2: tensor<1x4xf32>
13  %34 = call @WS1(%input, %STM): tensor<1x4xf32>
14  %35 = tf.Softmax(%34): tensor<1x4xf32>
15  %47 = call @WS2(%input, %STM): tensor<1x4xf32>
16  %48 = tf.Softmax(%47): tensor<1x4xf32>
17  %25 = call @WS3(%input, %STM): tensor<1x4xf32>
18  %26 = tf.Softmax(%25): tensor<1x4xf32>
19  %56 = call @WS4(%input, %STM): tensor<1x4xf32>
20  %57 = tf.Tanh(%56): tensor<1x4xf32>
21  %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
22  %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
23  %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
24  %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
25  %STM_up = tf.Mul(%26,%60): tensor<1x4xf32>
26  %output = lus.when %clk %STM_up: tensor<1x4xf32>
27  lus.yield(%output:tensor<1x4xf32>)
28    state(%STM_up:tensor<1x4xf32>,%LTM_up:tensor<1x4xf32>)
29 }
```

FIGURE 6.14 : Le résultat de la normalisation de Fig. 6.13.

Conversion vers un processus concurrent

Le nœud normalisé peut être compilé vers le dialecte **sync** pour décrire un processus concurrent, comme présenté en Section 5.2.3. Le résultat de cette passe est présenté en Fig. 6.15. Les traitements sur les données sont déplacés dans le corps d'une boucle infinie (lignes 6 à 37). Les paramètres de la fonction réactive sont des signaux d'entrée-sortie, lus et écrits au moyen des opérations `sync.input` (ligne 10) et `sync.output` (ligne 29). L'initialisation des variables d'état est placée avant la boucle principale (ligne 3 et 4) et les variables d'état sont ensuite maintenues comme des dépendances portées par la boucle (lignes 6 et 37). Les horloges sont converties dans la forme flot de contrôle de l'exécution conditionnelle (lignes 28 à 33). La barrière de cycle explicite, dont l'ordre est garanti par la dominance [21], est matérialisée par l'opération `tick` (ligne 35).

Une dernière passe de compilation est nécessaire pour ramener le dialecte **sync** aux dialectes standard de MLIR. Son résultat est illustré en Fig. 6.16. Pour des raisons de place, nous avons coupé le traitement des données du LSTM (sommes pondérées, etc.). La coupure est en ligne 27.

On observe ici que la fonction, outre son identifiant d'instance, prend en paramètre des pointeurs sur fonction et non plus des variables de type signal (ligne 1 et 2). Pour lire et écrire les entrées cycliques, ces pointeurs sur fonction sont appelés au moyen de l'opération `call_indirect` (lignes 14 et 32).

L'instanciation du nœud `time_tick` est accomplie au moyen de la communication avec l'environnement au travers de *buffers* dédiés. Aux lignes 7 à 9 la nouvelle instance est déclarée, et la communication en tant que telle a lieu aux lignes 16 à 24. L'appelant alloue les *buffers* servant à la communication (lignes 16 et 17), écrit sur le *buffer* d'entrée les données qu'il fournit à la nouvelle instance (ligne 18), fournit les *buffers* de communication à l'environnement (lignes 19 et 20), et rend le contrôle à l'environnement (ligne 21). Lorsque le sous-automate a achevé ses calculs pour le cycle courant et écrit son résultat, le contrôle revient à l'appelant qui peut récupérer le résultat du calcul (ligne 22) et désallouer les *buffers* d'échange (lignes 23 et 24). Certaines des fonctions utilisées dans ce mécanisme sont entièrement générées par mlir-lus (`@time_tick_start`) et encapsulent les protocoles de communication requis, d'autres sont directement des fonctions externes que l'on

[21] Les variables de synchronisation (lignes 28, 29, 32, 34, 35) sont ici implémentées sous la forme d'entiers pour faciliter leur manipulation.

```
1  sync.func @lstm(%si: in(tensor<1x40xf32),
2                 %so: out(tensor<1x40xf32>)) -> () {
3   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
4   %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
5   %true = arith.constant 1: i1
6   scf.while(%STM2=%LTM0,%STM2=%LTM0)
7     :(tensor<1x4xf32>,tensor<1x4xf32>) {
8     scf.condition(%true)
9   } do {
10    %input = sync.input(%si):tensor<1x40xf32
11    %period = tf.Const{dense<3>}: tensor<i32>
12    %clk = sync.inst 2 @time_tick(%period:i32): i1
13    %STM = select %clk, %STM0, %STM2: tensor<1x4xf32>
14    %LTM = select %clk, %LTM0, %LTM2: tensor<1x4xf32>
15    %34 = call @WS1(%input, %STM): tensor<1x4xf32>
16    %35 = tf.Softmax(%34): tensor<1x4xf32>
17    %47 = call @WS2(%input, %STM): tensor<1x4xf32>
18    %48 = tf.Softmax(%47): tensor<1x4xf32>
19    %25 = call @WS3(%input, %STM): tensor<1x4xf32>
20    %26 = tf.Softmax(%25): tensor<1x4xf32>
21    %56 = call @WS4(%input, %STM): tensor<1x4xf32>
22    %57 = tf.Tanh(%56): tensor<1x4xf32>
23    %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
24    %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
25    %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
26    %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
27    %STM_up = tf.Mul(%26,%60): tensor<1x4xf32>
28    %u1 = scf.if %clk -> i32 {
29      %u0 = sync.output(%so,%STM_up):tensor<1x4xf32>
30      scf.yield %u0: i32
31    } else {
32      %u0 = sync.undef(): i32
33    }
34    %u2 = sync.tick(%STM_up:tensor<1x4xf32>,%LTM_up:tensor<1x4xf32>,%u1:i32)
35    %stm = sync.sync(%STM_up:tensor<1x4xf32>,%LTM_up:tensor<1x4xf32>,%u2:i32)
36    scf.yield %stm,%LTM_up:tensor<1x4xf32>,tensor<1x4xf32>
37  }
38  sync.halt
39 }
```

FIGURE 6.15 : Le résultat de la compilation de Fig. 6.14 vers le dialecte **sync**.

```
1  func @lstm(%inst:i32,%si:(i32->(tensor<1x40xf32)),
2             %so:((i32,tensor<1x40xf32>)->(i32)) -> () {
3   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
4   %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
5   %true = arith.constant 1: i1
6   %zero = arith.constant 0: index
7   %tt_inst = arith.constant 2: i32
8   %tt = constant @time_tick_start:(i32)->(i1)
9   call @sch_set_instance(%tt_inst,%tt):(i32,(i32)->(i1))
10  scf.while(%STM2=%LTM0,%STM2=%LTM0)
11    :(tensor<1x4xf32>,tensor<1x4xf32>) {
12    scf.condition(%true)
13  } do {
14    %input = call_indirect %si(%inst):tensor<1x40xf32
15    %period = tf.Const{dense<3>}: tensor<i32>
16    %tt_i = memref.alloc(): memref<1xi32>
17    %tt_o = memref.alloc(): memref<1xi1>
18    memref.store %period, %tt_i%zero: memref<1xi32>
19    call @sch_set_io_I(%zero,%tt_i):(i32,memref<1xi32>)->()
20    call @sch_set_io_O(%zero, %tt_o):(i32,memref<1xi1>)->()
21    call @inst(%tt_inst):()->()
22    %clk = memref.load %tt_o%zero: memref<1xi1>
23    memref.dealloc(%tt_i):memref<1xi32>
24    memref.dealloc(%tt_o):memref<1xi32>
25    %STM = select %clk, %STM0, %STM2: tensor<1x4xf32>
26    %LTM = select %clk, %LTM0, %LTM2: tensor<1x4xf32>
27    ...
28    %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
29    %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
30    %STM_up = tf.Mul(%26,%60): tensor<1x4xf32>
31    %u1 = scf.if %clk -> i32 {
32      %u0 = call_indirect %so(%inst,%STM_up):i32
33      scf.yield %u0: i32
34    } else {
35      %u0 = constant 0: i32
36    }
37    %u2 = call @tick(%STM_up:tensor<1x4xf32>,%LTM_up:tensor<1x4xf32>,%u1:i32)
38      : i32
39    %stm = call @sync(%STM_up:tensor<1x4xf32>,%LTM_up:tensor<1x4xf32>,%u2:i32)
40      : (tensor<1x4xf32>)
41    scf.yield %stm,%LTM_up:tensor<1x4xf32>,tensor<1x4xf32>
42  }
43  return
44 }
```

FIGURE 6.16 : Le résultat de la compilation de Fig. 6.15 vers les dialectes standard de MLIR.

suppose définies dans l'ordonnanceur (`@sch_set_instance`, `@sch_set_io_I`, `@sch_set_io_O`, `inst`).

L'opération `undef` est implémentée sous la forme d'une constante (ligne 35). L'opération `tick` est implémentée par l'appel à une fonction externe permettant la communication avec l'environnement (lignes 37 et 38).

À partir de ce point, la fin de la compilation peut être assumée par les compilateurs externes. Le code résultant, en dialecte **llvmir**, peut finalement être exporté vers LLVM IR et être traité par le *back-end* de LLVM, avant d'être *linké* avec l'ordonnanceur externe.

Conversion vers les fonctions *step*/*reset*

Pour pouvoir utiliser la chaîne de compilation bas niveau et la machine virtuelle d'IREE, il nous faut compiler la spécification en Fig. 6.14 vers des fonctions step et reset. Le résultat est exposé en Fig. 6.17. Les variables d'état sont initialisées par l'appel à la fonction `reset` (en haut). La fonction `step` (en bas) les reçoit du *driver* en plus de ses entrées et les lui renvoie en plus de ses sorties lorsqu'elles sont mises à jour (en bas, lignes 2 à 4 et 29/30).

Comme nous l'avons dit précédemment, la représentation de la modularité synchrone dans la chaîne de compilation ciblant les fonctions step/reset n'est pas implémentée dans la version courante de notre compilateur ; en pratique, nous y substituons des opérations d'*inlining*. Le fragment de code concerné (en ligne 7) a donc été rédigé manuellement afin d'illustrer ce mécanisme.

L'état du nœud instancié `@time_tick` est agrégé à l'état de `@lstm` (variables `%t1` et `%t2`), avant l'appel de la fonction instanciée. Cette dernière est la fonction step générée à partir du nœud instancié.

Il faut noter que, la fonction step étant une fonction classique, elle retourne toujours (lignes 29 et 30). La non-utilisation de la variable `%o` est donc de la responsabilité du nœud qui l'a instanciée.

À ce niveau, il est désormais possible de chaîner les compilateurs tf-opt, iree-opt et iree-translate pour générer du code exécutable pour la machine virtuelle d'IREE. Ce dernier sera appelé par le *driver* C décrit précédemment.

```
1  func @lstm_reset()->(tensor<1x4xf32>,tensor<1x4xf32,tensor<i32>) {
2   %STM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
3   %LTM0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
4   %init = tf.Const{value = dense<0>} : tensor<i32>
5   return %STM0,%LTM0,%init: tensor<1x4xf32>,tensor<1x4xf32>,tensor<i32>
6  }
```

```
1  func @lstm_step(%input:tensor<1x4xf32>,
2                  %STM2:tensor<1x4xf32>,
3                  %LTM2:tensor<1x4xf32>,
4                  %t1:tensor<i32>)
5    :(tensor<1x4xf32>,tensor<1x4xf32>,tensor<1x4xf32>,tensor<i32>) {
6    %period = tf.Const{dense<3>}: tensor<i32>
7    %clk,%t2 = call @time_tick_step(%period,%t1): i1,tensor<i32>
8    %STM = select %clk, %STM0, %STM2: tensor<1x4xf32>
9    %LTM = select %clk, %LTM0, %LTM2: tensor<1x4xf32>
10   %34 = call @WS1(%input, %STM): tensor<1x4xf32>
11   %35 = tf.Softmax(%34): tensor<1x4xf32>
12   %47 = call @WS2(%input, %STM): tensor<1x4xf32>
13   %48 = tf.Softmax(%47): tensor<1x4xf32>
14   %25 = call @WS3(%input, %STM): tensor<1x4xf32>
15   %26 = tf.Softmax(%25): tensor<1x4xf32>
16   %56 = call @WS4(%input, %STM): tensor<1x4xf32>
17   %57 = tf.Tanh(%56): tensor<1x4xf32>
18   %39 = tf.Mul(%35, %LTM): tensor<1x4xf32>
19   %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
20   %LTM_up = tf.AddV2(%39,%58): tensor<1x4xf32>
21   %60 = tf.Tanh(%LTM_up) : tensor<1x4xf32>
22   %STM_up = tf.Mul(%26,%60): tensor<1x4xf32>
23   %o = scf.if %clk -> tensor<1x4xf32> {
24     scf.yield %STM_up: tensor<1x4xf32>
25   } else {
26     %n = arith.constant dense<0.0>:tensor<1x4xf32>
27     scf.yield %n: tensor<1x4xf32>
28   }
29   return %o,%STM_up,%LTM_up,%t2
30     :(tensor<1x4xf32>,tensor<1x4xf32>,tensor<1x4xf32>,tensor<i32>)
31 }
```

FIGURE 6.17 : Le résultat de la compilation de Fig. 6.14 vers des fonctions step (en bas) et reset (en haut).

6.3 Conclusion

Dans ce chapitre, nous avons commencé par situer le caractère *transformationnel* ou *réactif* d'un ANN parmi les différentes polysémies (ambiguïtés constituantes) spécifiques à la spécification des ANNs : apprentissage vs inférence, taille de *batch*, etc.

Nous décrivons ensuite l'architecture d'un *framework* réactif faisant par défaut le choix inverse des *frameworks* classiques transformationnels. Nous prenons appui sur l'intégration sémantique entre SSA et la programmation réactive synchrone réalisée dans les Chapitres 4 et 5 pour proposer un tel *framework* réactif entièrement outillé, par extension du *framework* TensorFlow/Keras. Après avoir décrit un sous-ensemble des outils – en particulier des compilateurs – qui sont mis en œuvre dans ce dernier, nous situons notre extension au niveau des composants Keras (formalisme de spécification) et MLIR (IR et *back-end* de compilation).

En effet, la différence essentielle entre le *framework* réactif que nous proposons et un *framework* classique se situe au niveau de la chaîne de compilation. Cette dernière consiste alors à générer, à partir d'une spécification Keras, une spécification MLIR SSA dont le contrôle de haut niveau est assuré par les primitives flot de données synchrones du dialecte **lus** défini en Section 5.2, et dont le traitement des données est assuré par les opérations et transformations de code définies dans les dialectes HPC pré-existants (essentiellement le dialecte **tf** de TensorFlow). Ainsi, une fois le contrôle réactif compilé vers les dialectes standard de MLIR (sous la forme d'automates communiquants ou de fonctions step/reset), les chaînes de compilation standard du *framework* peuvent prendre le relais pour générer le code exécutable. Notre approche permet donc d'exploiter pleinement les opérations et transformations de code matérialisées dans les *frameworks* existants.

Nous avons en particulier exploré deux chaînes de compilation de bas niveau : l'une cible la machine virtuelle IREE et génère un mélange de *bytecode* IREE et de code binaire, l'autre cible l'exécution CPU et génère un code LLVM IR pouvant alimenter les compilateurs standard du projet LLVM.

Au plan de l'ingéniérie, nous montrons que la chaîne ciblant IREE offre un accès facilité aux transformations de code complexes (et notamment aux optimisations, donc aux gains de performances). Cependant, les contraintes qu'elle exige sur la spécification d'entrée ne permettent pas, en l'état, de représenter des automates communiquants à l'intérieur d'IREE ; les nœuds Lustre doivent donc être compilés sous la forme de fonctions step/reset.

Par comparaison, la chaîne ciblant LLVM IR n'impose pas de telles restrictions, mais les transformations de code (passes de *lowering* et d'optimisation) doivent y être composées manuellement.

Nous illustrons finalement le fonctionnement du *framework* réactif que nous proposons en mettant en œuvre la compilation d'une spécification Keras décrivant un RNN basé sur LSTM, dont nous détaillons l'architecture et les opérations. Un tel réseau nous permet de mettre à l'épreuve la capacité de notre extension à spécifier et à compiler conjointement le traitement HPC des données (e.g. sommes pondérées, fonctions d'activation) et le contrôle réactif haut-niveau (e.g.itération à état le long du temps) d'un même système.

Chapitre 7

Résultats

Dans ce chapitre, nous évaluons l'expressivité et les performances de notre extension à MLIR/SSA sur des cas d'études non-triviaux. Dans un premier temps, nous travaillerons sur deux cas d'études issus du ML : un RNN et un réseau profond (50 couches). Dans un second temps, nous présenterons un *vocoder*, une application RTE classique de traitement du son. Elle n'appartient pas à la famille des DNNs mais est représentative des applications de traitement de signal RTE réalisant des calculs intensifs (e.g. FFT) sur fond de contrôle réactif complexe.

7.1 Réseaux de neurones

Nous évaluons ici notre extension sur la phase d'inférence de deux réseaux de neurones :

1. Un RNN simple qui compose une couche LSTM avec trois couches Dense. Nous donnons sa spécification Keras en Fig. 7.1. Nous avons déjà présenté en Chapitre 6 ce qu'est une couche LSTM et la manière dont **lus** permet d'en décrire le comportement de manière réactive et ne revenons pas dessus ici.

2. Un Resnet50, qui est une variante à 50 layers convolutifs de la famille des réseaux résiduels[50].

Nous commençons par décrire ce qu'est un Resnet50 et la manière dont il est représenté sous la forme d'un nœud **lus**, puis nous discutons les performances du code généré depuis chacune de ces deux spécifications.

```
1  inp = keras.Input(shape=(5,1),batch_size=3)
2  x = layers.LSTM(units=100,
3                  activation='relu',
4                  recurrent_activation='sigmoid',
5                  time_major=True)(inp)
6  x = layers.Dense(units=50, activation="relu")(x)
7  x = layers.Dense(units=50,activation="relu")(x)
8  x = layers.Dense(units=1,activation=None)(x)
9  model = keras.Model(inp, x)
```

FIGURE 7.1 : La spécification Keras du RNN que nous utilisons pour évaluer notre extension.

7.1.1 Présentation de Resnet50

Les CNNs groupés dans la famille des réseaux résiduels (*resnets*) sont largement utilisés pour la classification d'images complexes en trois dimensions (hauteur, largeur et couleur). Intensifs en calcul, il se caractérisent tout à la fois par leur grand nombre de couches (50, 101, 152...) et par leur caractère non-linéaire, des *raccourcis* enjambant certaines couches lors du calcul.

Les réseaux résiduels

Les réseaux résiduels ont été conçus pour résoudre le problème de la disparition des gradients[40]. Dans le contexte des réseaux très profonds, on observe en effet que passé un certain seuil de saturation, les résultats de l'apprentissage se dégradent. Pour adresser ce problème, les réseaux résiduels introduisent des *raccourcis* entre les couches convolutionnelles, qui permettent d'empiler les couches sans perdre d'information. Ce comportement repose sur les *blocs résiduels*. L'intuition derrière ces blocs résiduels peut être résumée comme suit :

- Soit L une composition suffisamment petite (2 ou 3) de couches convolutives, dont chacune peut éventuellement faire l'objet d'une *batch normalization* et d'une fonction d'activation. $L(x)$ est le résultat produit par l'application de L à l'entrée x.
- Soit *shortcut* la fonction raccourci enjambant L. Il s'agit souvent de la fonction identité, mais il peut aussi s'agir d'une couche convolutive.
- Le bloc résiduel B réalise le calcul $B(x) = L(x) + shortcut(x)$.

Un *resnet* est essentiellement composé de blocs résiduels.

Resnet50 dans Keras

La définition de Resnet50 (un resnet à 50 couches convolutives) fournie dans Keras repose quant à elle sur un bloc résiduel de 3 couches convolutives. La fonction Python utilisée pour construire itérativement Resnet50, exposée en Fig. 7.2, consiste à ajouter un bloc au réseau qui lui est fourni en entrée (variable `net`, le Resnet50 en cours de construction). On remarque que la définition de la fonction *shortcut* dépend du paramètre `conv_shortcut` en ligne 2. Si ce dernier vaut `True`, alors une convolution composée avec une opération de *batch normalization* est ajoutée en entrée du bloc (lignes 4 à 8). S'il vaut `False`, le raccourci est la fonction identité.

Pour une entrée de forme (1,56,56,64), une option `filters` paramétrant la taille des filtres convolutifs fixée à 64, et une option `conv_shortcut` fixée à `True`, un tel bloc se représente naturellement sous la forme du nœud **lus** en Fig. 7.3. Le code est alors doté d'une sémantique réactive : à chaque cycle, le résultat produit par la fonction raccourci (en rouge) et le résultat produit par les couches internes du bloc sont sommés (en vert).

L'architecture complète du ResNet50 fourni par Keras est illustrée en Fig. 7.4. Elle se présente comme suit :

- Les données d''entrée passent par une couche convolutive préliminaire.
- Le résultat de la couche préliminaire passe par la composition de 16 blocs résiduels de 3 couches convolutives (3 x 16 couches = 48 couches), dont les tailles de filtres convolutifs sont variables. La fonction en Fig. 7.2 est donc appelée 16 fois lors de la définition Keras du Resnet50.
- Le résultat issu de la composition des blocs résiduels est finalement traité par une couche hyper-connectée (Dense) chargée de la classification finale.

Un tel réseau n'exhibe pas de contrôle complexe de type état persistant ou transmission conditionnelle des données. Le Resnet50 réactif que nous avons évalué consiste donc simplement en un nœud **lus** embarquant toutes les opérations qui composent le Resnet50 fourni par Keras.

7.1.2 Performances (Resnet50 et RNN)

Nous évaluons ici les performances de notre méthode réactive de spécification et de compilation sur le Resnet50 présenté ci-dessus et sur le RNN dont la

```
def block(net, filters, kernel_size=3, stride=1,
          conv_shortcut=True, name):

    if conv_shortcut:
        shortcut = layers.Conv2D(4 * filters, 1, strides=stride,
                                 name=name + '_0_conv')(net)
        shortcut = layers.BatchNormalization(epsilon=1.001e-5,
                                             name=name + '_0_bn')(shortcut)
    else:
        shortcut = net

    x = layers.Conv2D(filters, 1, strides=stride,
                      name=name + '_1_conv')(net)
    x = layers.BatchNormalization(epsilon=1.001e-5,
                                  name=name + '_1_bn')(x)
    x = layers.Activation('relu',
                          name=name + '_1_relu')(x)

    x = layers.Conv2D(filters, kernel_size, padding='SAME',
                      name=name + '_2_conv')(x)
    x = layers.BatchNormalization(epsilon=1.001e-5,
                                  name=name + '_2_bn')(x)
    x = layers.Activation('relu',
                          name=name + '_2_relu')(x)

    x = layers.Conv2D(4 * filters, 1,
                      name=name + '_3_conv')(x)
    x = layers.BatchNormalization(epsilon=1.001e-5,
                                  name=name + '_3_bn')(x)

    x = layers.Add(name=name + '_add')(shortcut, x)
    x = layers.Activation('relu', name=name + '_out')(x)

    return x
```

FIGURE 7.2 : La fonction Python que Keras utilise pour ajouter itérativement un même bloc résiduel à un réseau pointé par la variable `net`. La fonction raccourci est en rouge, la somme finale en vert.

```
lus.node @block(%input: tensor<1x56x56x64xf32>)
            -> (tensor<1x56x56x256xf32>) {
  %16 = tf.Const() {value = XXX} : tensor<1x1x64x256xf32>
  %17 = tf.Conv2D(%input, %16) : tensor<1x56x56x256xf32>
  %18 = tf.Const() {value = XXX} : tensor<256xf32>
  %19 = tf.BiasAdd(%17, %18) : tensor<1x56x56x256xf32>
  %21 = tf.Const() {value = XXX} : tensor<4x256xf32>
  %shortcut = tf.FusedBatchNormV3(%19, %21) {epsilon = 1.001e-05 : f32}
    : tensor<1x56x56x256xf32>
  %26 = tf.Const() {value = XXX} : tensor<1x1x64x64xf32>
  %27 = tf.Conv2D(%input, %26) : tensor<1x56x56x64xf32>
  %28 = tf.Const() {value = XXX} : tensor<64xf32>
  %29 = tf.BiasAdd(%27, %28) : tensor<1x56x56x64xf32>
  %31 = tf.Const() {value = XXX} : tensor<4x64xf32>
  %35 = tf.FusedBatchNormV3(%29, %31) {epsilon = 1.001e-05 : f32}
    : tensor<1x56x56x64xf32>
  %36 = tf.Relu(%35) : tensor<1x56x56x64xf32>
  %37 = tf.Const() {value = XXX} : tensor<3x3x64x64xf32>
  %38 = tf.Conv2D(%36, %37) :  tensor<1x56x56x64xf32>
  %39 = tf.Const() {value = XXX : tensor<64xf32>
  %40 = tf.BiasAdd(%38, %39) : tensor<1x56x56x64xf32>
  %42 = tf.Const() {value = XXX} : tensor<4x64xf32>
  %46 = tf.FusedBatchNormV3(%40, %42) {epsilon = 1.001e-05: f32}
    : tensor<1x56x56x64xf32>
  %47 = tf.Relu(%46) : tensor<1x56x56x64xf32>
  %48 = tf.Const() {value = XXX} : tensor<1x1x64x64xf32>
  %49 = tf.Conv2D(%47, %48) : tensor<1x56x56x256xf32>
  %50 = tf.Const() {value = XXX} : tensor<256xf32>
  %51 = tf.BiasAdd(%49, %50)  : tensor<1x56x56x256xf32>
  %53 = tf.Const() {value = XXX} : tensor<4x256xf32>
  %57 = tf.FusedBatchNormV3(%51, %53) {epsilon = 1.001e-05 : f32}
    : tensor<1x56x56x256xf32>
  %58 = tf.AddV2(%shortcut, %57) : tensor<1x56x56x256xf32>
  %59 = tf.Relu(%58) : tensor<1x56x56x256xf32>
  lus.yield(%59: tensor<1x56x56x256xf32>)
}
```

FIGURE 7.3 : La représentation du bloc résiduel produit par la fonction Python en Fig. 7.2 sous la forme d'un nœud **lus**. On suppose fixés la forme de l'entrée, la taille des filtres et la fonction raccourci. La fonction raccourci est en rouge, la somme finale en vert.

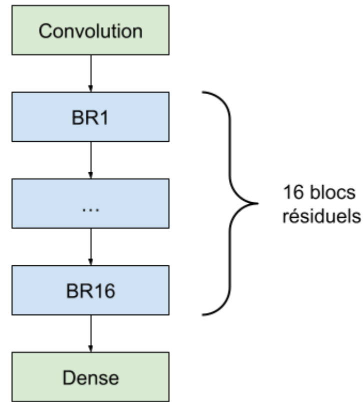


Figure 7.4 : L'architecture complète du Resnet50 fourni par Keras.

spécification est donnée en Fig. 7.1.

Nous avons ciblé l'exécution sur deux architectures distinctes :

1. Une machine GNU/Linux équipée d'un CPU Intel Core i5-7600 (x86_64) dont la fréquence est 3.50GHz. La chaîne de compilation ciblant LLVM produit du code objet, celle qui cible IREE génère un binaire englobé dans un *bytecode* qui recourt à des bibliohèques dynamiques.

2. Une machine GNU/Linux équipée d'un GPU Nvidia RTX A5000. Le code exécutable IREE s'interface avec deux *drivers* GPU distincts, Vulkan et Cuda.

Pour chaque cas d'étude, nous comparons 4 méthodes de spécification et de compilation distinctes :

1. Un réencodage du réseau réalisé par nos soins en formalisme Heptagon (une implémentation état-de-l'art de Lustre), dont le code C généré est ensuite traité par GCC -O3.

2. Une spécification Keras convertie en un nœud **lus** par keras-lus, avant d'être compilée sous la forme d'un processus concurrent. Le *back-end*, entièrement composé par nos soins, cible LLVM. Nous y appliquons peu d'optimisations : du *packing*, du *tiling*, mais pas de vectorisation.

3. Une spécification Keras convertie en un nœud **lus** par keras-lus, avant d'être compilée sous la forme de fonctions step/reset. Le *back-end* est l'environnement IREE.

4. Une spécification Keras compilée en composant les passes transformationnelles standard de TensorFlow et d'IREE jusqu'à la machine virtuelle IREE. Exception faite de la compilation du contrôle réactif, cette approche applique exactement les mêmes passes de compilation que la précédente.

Les mesures que nous donnons sont la moyenne du temps d'exécution par cycle, pour 10 cycles.

Resnet50. Les performances pour Resnet50 sur CPU sont transcrites dans le Tableau 7.1. Le Tableau 7.2 donne les performances sur GPU.

Table 7.1 : Temps d'exécution moyen par cycle pour ResNet50, sur CPU, en millisecondes.

Méthode	Temps
Heptagon + gcc-O3	5095
MLIR/lus -> LLVM IR	4954
MLIR/lus -> IREE	1818
MLIR/tf -> IREE	2059

Table 7.2 : Temps d'exécution moyen par cycle pour ResNet50, sur GPU, en millisecondes.

Méthode	Temps (Cuda)	Temps (Vulkan)
MLIR/lus -> IREE	57,1	9,7
MLIR/tf -> IREE	53,4	8,4

Ces résultats montrent que, sur un réseau très profond comme Resnet50, l'exploitation des opérations et transformations de code matérialisées dans MLIR/TensorFlow nous permettent d'égaler les performances de l'état de l'art flot de données synchrones lorsque nous composons toutes les passes de compilation manuellement, et de les dépasser largement lorsque nous nous reposons sur un *back-end* intégré comme IREE.

Ils montrent également que le contrôle réactif, ici minimal (Resnet50 n'embarque ni état persistant, ni exécution conditionnelle), n'engendre pas de pessimisation des performances par rapport à la chaîne de compilation transformationnelle équivalente.

RNN. Les performances pour le RNN en Fig. 7.1 sur CPU sont transcrites dans le Le Tableau 7.3. Le Tableau 7.4 donne les performances sur GPU. Là non plus, on ne constate pas de pessimisation des performances due au contrôle réactif.

Notons que les mesures de la chaîne transformationnelle ciblant IREE sont à chaque fois données sous la forme d'une paire : le membre gauche désigne les performances du même réseau spécifié avec un horizon temporel de 1 au lieu de 5 [1], le membre droit désigne les performances du RNN spécifié en Fig. 7.1 exactement tel quel [2].

Table 7.3 : Temps d'exécution moyen par cycle pour le RNN en Fig. 7.1, sur CPU, en millisecondes.

Méthode	Temps
Heptagon + gcc-O3	0,101
MLIR/lus -> LLVM IR	0,164
MLIR/lus -> IREE	0,242
MLIR/tf -> IREE	0,353/0,628

Table 7.4 : Temps d'exécution moyen par cycle pour le RNN en Fig. 7.1, sur GPU, en millisecondes.

Méthode	Temps pour CUDA	Temps pour Vulkan
MLIR/lus -> IREE	1,82	1,13
MLIR/tf -> IREE	2,71/2,93	2,74/4,46

Dans ce second cas, les performances sont meilleures : le code généré par la chaîne de compilation transformationnelle ciblant Cuda nécessite quasi-

[1] Il réalise donc la même quantité de calculs par appel que notre implémentation réactive par cycle, mais ne calcule pas le même résultat, puisque son état est réinitialisé à chaque cycle.

[2] Il réalise 5 itérations par appel, mais calcule bien le résultat demandé.

ment le même temps d'exécution pour traiter un horizon de temps de 1 que pour traiter un horizon de temps de 5. Nous supposons que cette dernière approche permet de mieux paralléliser le traitement des données. Pour autant, L'exploitation du parallélisme des données n'est pas aussi décisive que dans un réseau plus profond et traitant des données plus volumineuses comme Resnet50. C'est certainement pourquoi, sur CPU, la chaîne de compilation basée sur Heptagon et GCC, bien que se situant complètement à l'extérieur des *frameworks* ML, parvient à surclasser les autres méthodes.

7.2 Vocoder

Dans cette section, nous présentons notre troisième cas d'étude : un vocoder (PTV, *Pitch Tuning Vocoder* qui acquiert des échantillons sonores en temps-réel et produit en sortie un écho dont le timbre est modifié selon les paramètres fournis par l'utilisateur. Il s'agit d'une application traditionnelle de traitement de signal en temps-réel, caractérisée par son contrôle réactif complexe (e.g. délais, transmission conditionnelle) tout autant que par le traitement intensif qu'elle applique aux données (e.g. FFT, IFFT).

Nous illustrons la spécification et la compilation conjointes de ces aspects HPC[3] et RTE en présentant le nœud principal du programme, puis les nœuds instanciés par ce dernier (l'algorithme de variation du timbre et le traitement des commandes utilisateur).

Ce cas d'étude nous permet d'évaluer notre extension à deux titres :

- Le contrôle réactif complexe met à l'épreuve l'expressivité des primitives que nous avons incorporé dans MLIR/SSA.
- Le bon fonctionnement du *vocoder* montre que notre implémentation n'introduit d'erreur ni dans le contrôle réactif ni dans le traitement des données.

[3]Dans notre implémentation, les traitements HPC sur les données sont accomplis par des fonctions MLIR programmées par nos soins au moyen des dialectes standard de MLIR, mais ils peuvent généralement être implémentés au moyen de dialectes de plus haut niveau (comme ceux qui sont distribués avec les *frameworks* ML : **tf**, **mhlo**, etc.).

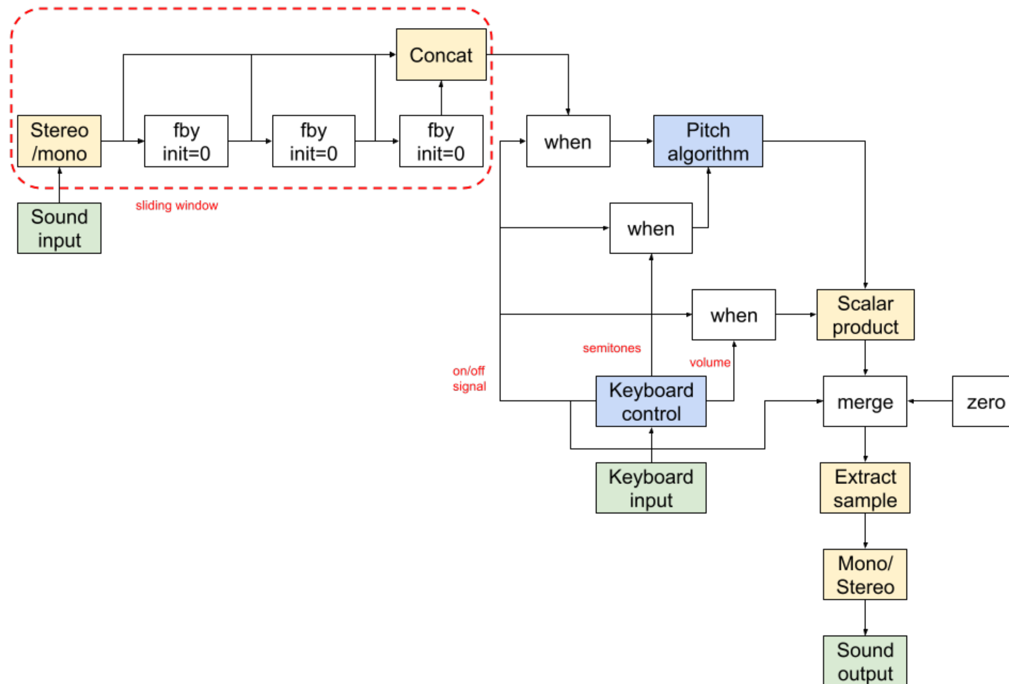


FIGURE 7.5 : Le flot de données du PTV. Entrées-sorties en vert, fonctions externes en jaune, instanciation de nœuds réactifs en bleu.

7.2.1 Nœud principal

Si l'on suppose implémentés le sous-automate encapsulant l'algorithme de variation du timbre et le sous-automate encapsulant la réaction aux commandes du clavier, le comportement haut niveau du PTV peut être décrit au moyen d'un nœud Lustre concis. Le flot de données de ce nœud est illustré en Fig. 7.5, et le code MLIR/lus correspondant est en Fig. 7.6 (pour une meilleure lisibilité, nous avons simplement allégé les types des appels de fonctions).

Les commandes que l'utilisateur saisit au clavier sont obtenues à chaque cycle. Une fois acquise, la saisie est traitée cycliquement par l'instance du sous-automate dédié (ligne 16), et permet :

- De désactiver l'écho (variable `%sndon`).

```
lus.node @pitch(%kbd:i8,%sndbufin:tensor<512xi16>)->(tensor<512xi16>) {
  // The sliding window
  %c0 = call @bzero_i16_256() : tensor<256xi16>
  %circ3 = call @stereo2mono(%sndbufin):(tensor<256xi16>)
  %circ2 = lus.fby %c0 %circ3 : tensor<256xi16>
  %circ1 = lus.fby %c0 %circ2 : tensor<256xi16>
  %circ0 = lus.fby %c0 %circ1 : tensor<256xi16>
  %sample = call @concat_samples(%circ0,%circ1,%circ2,%circ3)
    : tensor<1024xf32>

  // Keyboard controller that processes keyboard input and
  // determines:
  // - whether the sound should be processed (initially yes)
  // - the volume of the output sound (initially 100%)
  // - the pitch change, in semitones (initially +1 semitone)
  %sndon,%volume,%semitones = lus.instance @kbd_ctrl(%kbd) : (i1,f32,f32)

  // Based on the previous control value, subsample the input
  // to the pitch processing node, thus conditioning execution
  %sample_cond = lus.when %sndon %sample : tensor<1024xf32>
  %semitones_cond = lus.when %sndon %semitones : f32

  // Pitch processing pipeline
  %pitch_output = lus.instance @pitch_algo(%semitones_cond, %sample_cond)
   :(tensor<1024xf32>)

  // Sound volume control
  %vol_cond = lus.when %sndon %volume : f32
  %output = call @f32_tensor_float_product(%vol_cond,%pitch_output)
   :(tensor<1024xf32>)

  // If the pitch algo is disabled, I still have to output
  // something (outputs are on the base clock), so I output
  // zero.
  %zero = call @bzero_f32_1024(): () -> tensor<1024xf32>
  %output_merged = lus.merge %sndon %output %zero: tensor<1024xf32>

  // Write 256 samples from %output_acc to the soundcard
  %out = call @extract_samples(%output_merged) : (tensor<256xi16>)
  %sndbufout = call @mono2stereo(%out): (tensor<512xi16>)

  lus.yield(%sndbufout:tensor<512xi16> )
}
```

FIGURE 7.6 : Le nœud Lustre décrivant le comportement haut niveau du PTV.

- D'incrémenter/décrémenter le volume de l'écho (variable `%volume`).
- D'incrémenter/décrémenter le ton de l'écho (par demi-tons, variable `%semitone`).

Les entrées son sont également acquises à chaque cycle. Elles sont d'abord ramenées de deux canaux à un seul (ligne 4). Le délai entre l'entrée et la sortie est spécifié au moyen de la composition d'opérateurs `fby` qui forment une fenêtre glissante sur les entrées (lignes 3 à 9). La génération d'un tenseur nul (ligne 3) et la concaténation des 3 entrées retardées (lignes 5 à 7) avec l'entrée courante (ligne 4) pourraient également être spécifiées au moyen d'opérations du dialecte **`tf`** (`tf.Const` et `tf.Concat`).

La transmission des entrées à l'algorithme principal est conditionnée (lignes 20 et 21) au signal d'activation/désactivation émis par le pilote du clavier. Si l'écho est désactivé, l'inférence d'horloges empêche le déclenchement de ces calculs : dans ce cas, on récupère simplement un vecteur nul (lignes 35/36). Sinon, on modifie effectivement le timbre de l'échantillon en instanciant le sous-automate dédié (lignes 24 et 25), puis on fait varier le volume au moyen du produit du vecteur `%pitch_output` par le scalaire `%vol_cond` (lignes 29/30), qui pourrait également être spécifié par l'opération `tf.Mul`.

Il ne reste plus qu'à extraire le dernier quart de la fenêtre glissante (ligne 39) et à ramener le résultat à deux canaux (ligne 40) avant de l'écrire sur les sorties (ligne 42). La sortie du système pointe sur la sortie standard Unix.

Les fonctions implémentant les signaux d'entrée-sortie qui lisent la saisie clavier, lisent l'entrée standard Unix et écrivent sur la sortie standard Unix sont implémentées en C.

7.2.2 Nœuds instanciés par le nœud principal

Algorithme de variation du timbre. L'algorithme de variation du timbre qui forme le cœur du PTV est décrit par le graphe flot de données en Fig. 7.7, et le code MLIR/lus correspondant est en Fig. 7.8. Sans entrer dans le détail de cet algorithme (que nous utilisons sans l'avoir conçu), signalons :

- l'utilisation de `fby` (lignes 22, 31, 43), notamment pour lisser la *phase* des échantillons d'un cycle à l'autre (lignes 20 à 25).
- l'utilisation d'une FFT pour passer dans le domaine des fréquences (ligne 16), laquelle, programmée manuellement dans notre implémenta-

tion, pourrait également être spécifiée au moyen de l'opération `tf.RFFT` (*Real Fast Fourier Transform*).

Traitement des commandes utilisateur. Le traitement des commandes utilisateur, dont le flot de données est illustré en Fig. 7.9 et le code correspondant est en Fig. 7.10, représente les paramètres du vocoder (allumé/éteint, variation du timbre, volume) sous la forme de variables d'état maintenues par des opérations `fby` et initialisées par des constantes. Des flots constants modifient ces variables dès lors qu'une commande utilisateur, saisie au clavier, active l'une des opérations `select` en lignes 19, 20, 27, 28, 36 et 37 :

- 's' pour couper le vocoder ;
- 'a' pour activer le vocoder ;
- 'm' et 'n' pour incrémenter/décrémenter le timbre ;
- '-' et '+' pour incrémenter/décrémenter le volume.

7.3 Conclusion

Ce chapitre a été consacré au prolongement de l'étude expérimentale amorcée au chapitre précédent.

En plus des RNNs basés sur LSTM que nous avons déjà présenté en Section 6.2, nous introduisons deux cas d'étude supplémentaires, portant l'ensemble à 3 :

- ResNet50, un ANN profond (50 couches convolutives) de type *réseau résiduel*. Un tel ANN est intensif en calcul, bien que son contrôle réactif soit moins complexe que celui d'un RNN (i.e. pas de transmission de valeurs d'un cycle à l'autre).
- Une application HPC qui n'est pas un ANN : un *vocoder* de changement de timbre interfacé avec les entrées et sorties son de l'OS et implémentant des opérations de traitement de signal et un contrôle réactif complexes. Cette application nous permet de montrer que l'extension de MLIR et de SSA peut permettre la spécification et l'implantation réactives d'applications HPC dépassant le domaine du ML et couvrant les aspects centraux du pré-traitement et du post-traitement des données.

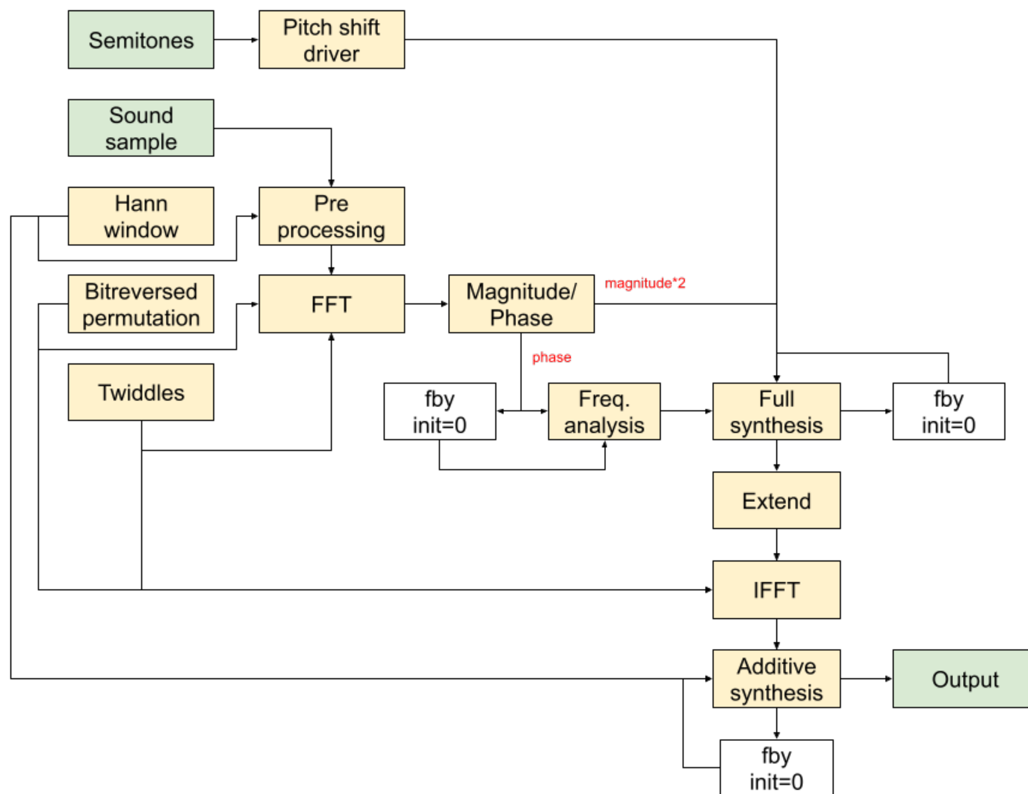


Figure 7.7 : Le flot de données de l'algorithme de variation du timbre. Entrées-sorties en vert, fonctions externes en jaune.

```
lus.node @pitch_algo(%semitones:f32,%sample:tensor<1024xf32>)
                 ->(tensor<1024xf32>) {  // %output_acc
  // FFT init
  %perm = call @bitrev_init() : () -> (tensor<1024xi32>)
  %twid = call @compute_twiddles(): ()->(tensor<1024xcomplex<f32>>)
  %hann = call @hann_window(): ()->(tensor<1024xf32>)
  %f0_512 = call @bzero_f32_512() : ()->tensor<512xf32>
  %f0_1024 = call @bzero_f32_1024() : ()->tensor<1024xf32>
  // Compute the pitch shift internal parameter
  %pitch_shift = call @pitch_shift_driver(%semitones): (f32)->(f32)
  // Apply the Hann window to the input sample and
  // convert to complex
  %win = call @pretreatment(%hann,%sample)
    :(tensor<1024xf32>,tensor<1024xf32>)->(tensor<1024xcomplex<f32>>)
  // Apply the FFT
  %win_fft = call @fft(%perm,%twid,%win)
    :(tensor<1024xi32>,tensor<1024xcomplex<f32>>,tensor<1024xcomplex<f32>>)
     ->tensor<1024xcomplex<f32>>
  // Compute magnitude*2 and phase
  %mag2,%phase = call @mag_phase(%win_fft)
    :(tensor<1024xcomplex<f32>>)->(tensor<512xf32>,tensor<512xf32>)
  %pre_phase = lus.fby %f0_512 %phase: tensor<512xf32>
  // Frequency analysis
  %analysis_freq = call @analysis_full(%phase,%pre_phase)
    :(tensor<512xf32>,tensor<512xf32>)->(tensor<512xf32>)
  // Synthesis of new frequencies
  %fft_pos,%sum_freq = call @synthesis_full(%pitch_shift,%pre_sum_freq,%mag2,
                                            %analysis_freq)
    :(f32,tensor<512xf32>,tensor<512xf32>,tensor<512xf32>)
     ->(tensor<512xcomplex<f32>>,tensor<512xf32>)
  %pre_sum_freq = lus.fby %f0_512 %sum_freq: tensor<512xf32>
  // Extend the result to a full sample vector, with 0 negative values
  %fft_pos_ext = call @extend_ifft_in(%fft_pos)
    :(tensor<512xcomplex<f32>>)->(tensor<1024xcomplex<f32>>)
  // IFFT to go back to time domain
  %ifft_out = call @ifft(%perm,%twid,%fft_pos_ext)
    :(tensor<1024xi32>,tensor<1024xcomplex<f32>>,tensor<1024xcomplex<f32>>)
     ->tensor<1024xcomplex<f32>>
  // (* Build the additive factor *)
  %output_acc,%rot_acc = call @additive_synthesis(%hann,%ifft_out,%pre_rot_acc)
    :(tensor<1024xf32>,tensor<1024xcomplex<f32>>,tensor<1024xf32>)
     ->(tensor<1024xf32>,tensor<1024xf32>)
  %pre_rot_acc = lus.fby %f0_1024 %rot_acc: tensor<1024xf32>
  lus.yield(%output_acc: tensor<1024xf32>)
}
```

FIGURE 7.8 : Le nœud Lustre décrivant l'algorithme de variation du timbre.

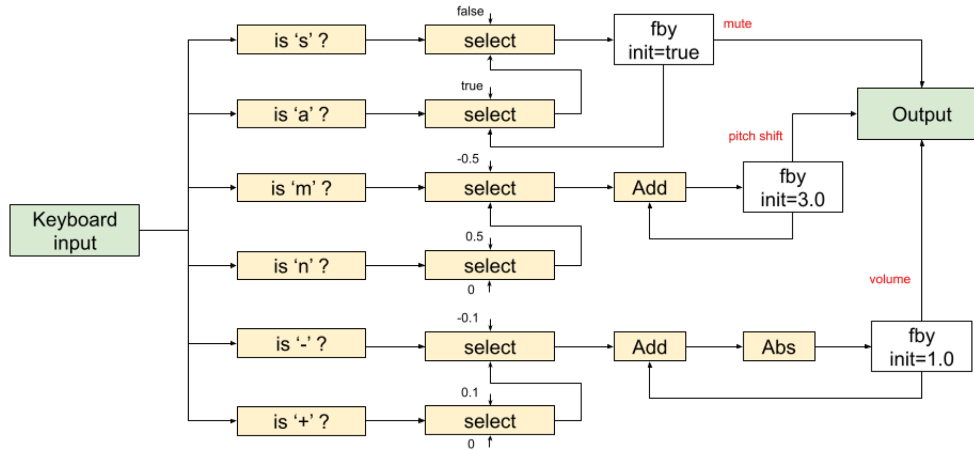


FIGURE 7.9 : Le flot de données du traitement des commandes utilisateur.

Nous utilisons ces 3 cas d'étude pour situer nos résultats par rapport à l'état de l'art de la compilation de langages synchrones et à celui de la compilation HPC. Nous montrons en particulier que notre extension :

- Dispose de l'expressivité nécessaire pour spécifier ces applications au bon niveau d'abstraction. Abstraction faite des applications qui ne peuvent faire l'objet d'une mise en œuvre réactive (à l'image des réseaux récurrents bidirectionnels ou des réseaux récursifs que nous avons évoqués au Chapitre 2), les formalismes que nous proposons enrichissent tout à la fois les mécanismes disponibles dans la programmation SSA et dans la programmation synchrone, sans devoir renoncer aux travaux cristallisés dans chacun de ces deux mondes.

- N'engendre pas de pessimisation des performances. Nous comparons en particulier les mesures produites par la compilation du RNN basé sur LSTM et par celle du Resnet50 en suivant notre approche, avec celles produites par la compilation classique (transformationnelle) de ces réseaux. Nous montrons que le contrôle réactif n'engendre pas de variation significative des performances.

  Nous arrivons à la même conclusion en comparant nos mesures avec celles produites par la compilation des même réseaux réencodés dans

```
1  lus.node @kbd_ctrl(%kbd:i8)->(i1,f32,f32) {
2    // Testing all possible input characters
3    %cha = arith.constant  97 : i8 // Letter 'a'
4    %chs = arith.constant 115 : i8 // Letter 's'
5    %chn = arith.constant 110 : i8 // Letter 'n'
6    %chm = arith.constant 109 : i8 // Letter 'm'
7    %pls = arith.constant  43 : i8 // Character '+'
8    %min = arith.constant  45 : i8 // Character '-'
9    %ca = arith.cmpi "eq",%kbd,%cha : i8
10   %cs = arith.cmpi "eq",%kbd,%chs : i8
11   %cn = arith.cmpi "eq",%kbd,%chn : i8
12   %cm = arith.cmpi "eq",%kbd,%chm : i8
13   %cpls = arith.cmpi "eq",%kbd,%pls : i8
14   %cmin = arith.cmpi "eq",%kbd,%min : i8
15   // Computing the mute Boolean output
16   %true = arith.constant true
17   %false = arith.constant false
18   %mute = lus.fby %true %pre_mute : i1
19   %x = select %ca,%true,%mute : i1
20   %pre_mute = select %cs,%false,%x : i1
21   // Computing the volume output
22   %f0 = arith.constant 0.0 : f32 // Min value
23   %f1 = arith.constant 1.0 : f32 // Max value
24   %fip = arith.constant 0.1 : f32 // Positive increment
25   %fim = arith.constant -0.1 : f32 // Negative increment
26   %volume = lus.fby %f1 %pre_volume : f32
27   %y = select %cpls,%fip,%f0 : f32
28   %inc = select %cmin,%fim,%y: f32
29   %vol_tmp1 = arith.addf %volume,%inc : f32
30   %pre_volume = math.abs %vol_tmp1 : f32
31   // Computing the pitch shift in number of semitones
32   %f3 = arith.constant 3.0 : f32 // Max value
33   %fcp = arith.constant 0.5 : f32 // Positive pitch increment
34   %fcm = arith.constant -0.5 : f32 // Negative pitch increment
35   %semitones = lus.fby %f3 %pre_semitones :f32
36   %z = select %cn,%fcp,%f0 : f32
37   %inc_semitones = select %cm,%fcm,%z: f32
38   %pre_semitones = arith.addf %semitones,%inc_semitones : f32
39   lus.yield (%mute:i1,%volume:f32,%semitones:f32)
40 }
```

FIGURE 7.10 : Le nœud Lustre décrivant le traitement des commandes utilisateur.

un langage flot de données synchrone (Heptagon) dont le compilateur génère du code C qui est ensuite traité par GCC. Ce résultat est d'autant plus net lorsque nous tirons profit de la chaîne de compilation et de la machine virtuelle agrégées dans l'environnement IREE.

Au terme de ce chapitre expérimental, nous considérons que l'approche que nous proposons ouvre la voie à des solutions réalistes pour spécifier et compiler conjointement les aspects HPC et RTE d'un même système.

Chapitre 8

Conclusion

Le travail que nous avons présenté tout au long de cette thèse vise à rapprocher les domaines HPC et RTE en permettant la spécification et la compilation conjointes des aspects HPC et RTE d'un même système. Comme nous l'avons montré en Chapitre 1, ces domaines se sont développés, jusqu'à aujourd'hui, de manière disjointe. Ils ont résolu des défis scientifiques et techniques considérables dans leurs champs respectifs, mais les formalismes, algorithmes et outils qu'ils ont élaboré ne sont pas, ou peu conçus pour fonctionner ensemble.

Dans le monde du ML, ce fait est à l'origine du paradoxe que nous avons introduit en Section 1.2.4. Ce dernier consiste en cela que la compilation traditionnelle des ANNs est transformationnelle, y compris pour l'embarqué, alors même qu'ils sont spécifiés au moyen du paradigme flot de données et que leur incorporation dans des systèmes RTE requiert une implémentation réactive. Dès lors, les *frameworks* ML courants exigent que la spécification d'un tel ANN réactif soit exprimée de manière transformationnelle (par exemple, en représentant le temps dans l'espace), qu'elle soit compilée comme une fonction transformationnelle, et qu'elle soit finalement implantée dans un système réactif manuellement. Nous avons détaillé cette approche, en particulier, en Section 2.2.2.

Différentes approches, au niveau spécification comme au niveau implémentation, se sont efforcées d'intégrer les points de vue respectifs du monde HPC et du monde RTE. Nous avons présenté certaines de celles qui sont appliquées dans le domaine du ML, en Sections 2.2.3 et 2.3. Sans se limiter au ML, notre contribution s'inscrit dans la continuité de cet effort d'intégration scientifique et technique.

8.1 Notre contribution

Nous partons du formalisme SSA, qui sert de *back-end* à plusieurs *frameworks* de compilation ML/HPC, et en particulier de son dialecte MLIR. Nous en donnons une description détaillée en Chapitre 3. Nous l'étendons ensuite aux formalismes synchrones venant du monde RTE pour permettre la représentation des comportements réactifs synchrones à deux niveaux d'abstraction :

- **Bas niveau (impératif).** Il s'agit ici d'une extension réactive de SSA et de sa sémantique opérationnelle. Ce niveau est bien adapté à la représentation de comportements en fin de chaîne de compilation, étant proche des mécanismes opérationnels d'implémentation. Nous étendons SSA avec les primitives nécessaires pour décrire l'interaction de composants synchrones entre eux et avec leur environnement dans le temps : séparation de l'exécution en réactions discrètes, entrées-sorties cycliques, modularité synchrone, absence d'un signal lors d'un cycle. Cette extension peut être vue comme un *back-end* SSA générique pour la programmation synchrone.

- **Haut niveau (flot de données).** Ce niveau de spécification est celui du *front-end*, qui doit permettre une spécification naturelle des algorithmes ML, de contrôle embarqué, de traitement de signal, etc. Pour cette raison, ce niveau suit un paradigme flot de données, avec un jeu de primitives emprunté au langage Lustre. La définition de ce niveau profite des extensions que MLIR apporte à SSA, et notamment de la possibilité de relaxer la dominance.

Les extensions spécifiques à chacun des deux niveaux sont groupées en deux dialectes MLIR : **sync** pour le bas niveau impératif (Chapitre 4) et **lus** (Chapitre 5) pour le haut niveau flot de données. Ces extensions peuvent être vues comme un aller-retour entre SSA/MLIR et la programmation synchrone :

- Pour la spécification, nous partons du socle SSA/MLIR pour arriver à la programmation synchrone.

- Le retour se fait par la compilation. Nous partons d'une spécification synchrone pour la ramener aux primitives SSA, ce qui permet la compilation conjointe du contrôle réactif et des traitements SSA sur les données.

Ce travail nous permet d'établir un lien entre ces deux familles de langages à trois niveaux : au niveau des formalismes, au niveau de l'algorithmique et au niveau des outils.

Lien entre les formalismes. Cette intégration sémantique et outillée exploite, au niveau **lus**, deux propriétés communes au flot de données synchrone et au formalisme SSA :

1. L'un et l'autre implémentent le principe SSA, bien que ce dernier soit garanti, d'un côté et de l'autre, par des contraintes différentes (critère de dominance vs calcul d'horloges).
2. L'un et l'autre sont globalement séquentiels et localement concurrents.
3. L'un et l'autre donnent à leurs variables une sémantique de valeur (et non pas de référence mémoire).

Dès lors, l'incorporation de formalismes synchrones dans SSA requiert :

1. De permettre dans SSA la manipulation explicite de valeurs absentes/indéfinies dont les vérifications de haut niveau (e.g. calcul d'horloges) garantissent qu'elle ne sont jamais utilisées dans les calculs ou les décisions.
2. De normaliser, vers une spécification entièrement compatible avec SSA, une spécification contenant des opérateurs `fby`, lesquels peuvent ne pas respecter pas le critère de dominance tout en respectant le principe SSA.
3. De permettre dans SSA la spécification de composants interagissant entre eux et avec leur environnement le long de cycles synchrones strictement disjoints.

Lien algorithmique. Ce travail permet une large réutilisation de l'algorithmique SSA dans les spécifications synchrones. En effet, il nous a suffi de définir les deux dialectes **lus** et **sync**, leur compilation vers les opérations de base de MLIR/SSA, la normalisation de **lus** et le calcul d'horloges de **lus** pour assurer la compatibilité avec SSA. Toutes les autres vérifications et transformations de code réutilisent l'algorithmique existante. Par exemple :

- L'analyse de causalité de Lustre est assurée par l'analyse de dominance SSA ;
- Les vérifications des types de données sont assurées par les algorithmes de typage du compilateur MLIR ;
- La synthèse de *buffers* est également assurée par l'algorithmique existante.

Lien entre les outils. En particulier, notre approche permet d'utiliser, dans une spécification flot de données synchrone, les opérations et transformations pour le traitement HPC des données qui sont matérialisées dans les compilateurs MLIR. Ce fait concerne les opérations issues des dialectes issus mlir-opt, mais également celles qui sont issues des dialectes définis dans un compilateur *domain-specific* comme tf-opt. La spécification conjointe des aspects HPC et RTE à laquelle cette propriété donne accès établit donc une *jonction* entre les outils manipulant les spécifications synchrones et les outils manipulant la représentation SSA.

Nous mettons cette approche en œuvre dans les Chapitres 6 et 7 et montrons qu'elle n'engendre ni pessimisation des performances par rapport aux approches HPC, ni perte d'expressivité par rapport aux approches RTE.

8.2 Travaux ultérieurs

Au-delà des bénéfices immédiats qui se matérialisent dans l'état actuel de notre contribution, cette dernière peut être prolongée dans le sens d'une intégration encore plus étroite des techniques ML/HPC et RTE.

Entraînement supervisé réactif. Parmi les transformations de code courantes dans un *framework* ML, il y a la différentiation automatique des opérateurs ML, sur laquelle repose l'algorithme de rétropropagation du gradient. Définir et implémenter les règles de différentiation s'appliquant à l'ensemble réduit d'opérateurs synchrones que nous introduisons permettrait donc de réaliser, non seulement la compilation conjointe des aspects RTE et HPC d'un même système, *mais également leur différentiation conjointe*. De telles règles de différentiation permettraient de faciliser la spécification et l'implémentation, non seulement de la phase d'inférence d'un ANN réactif, mais également de sa phase d'apprentissage.

Programmation d'agents RL complexes Les primitives générales de la programmation synchrone peuvent permettre de spécifier naturellement les comportements réactifs de plus en plus complexes des agents RL, et de les implémenter suivant des techniques unifiées. Ces comportements sont en particulier :

- De l'exécution conditionnelle ;
- Des comportements multi-périodiques ;
- La communication synchronisée de divers composants neuronaux ;
- Différentes formes d'état/de mémoire entre les cycles (e.g. état de l'agent pour les problèmes non-markoviens, *replay memory*).

Implémentation RTE

Le fait de pouvoir spécifier un ANN avec un formalisme flot de données synchrone donne accès aux méthodes issues du monde RTE, notamment pour l'allocation de ressources et l'ordonnancement (e.g. LoPhT[80]). Pour ce faire, il est néanmoins nécessaire de fournir un code prédictible, dont l'allocation de ressources est typiquement statique. Or, les *frameworks* ML courants adoptent la stratégie inverse. Dès lors, deux approches sont possibles pour générer un code dont les bornes en temps d'exécution, énergie consommée ou mémoire occupée sont plus précises :

- **Approche bibliothèque.** Une solution consisterait à compiler les opérations HPC haut-niveau vers des appels à une bibliothèque conçue pour la prédictibilité comme RTNeural[20]. Une telle approche reviendrait néanmoins à renoncer aux optimisations matérialisées dans les compilateurs MLIR en évolution constante.
- **Approche compilation.** Une autre solution consisterait à prendre appui sur l'extrême modularité des compilateurs MLIR pour substituer aux passes de compilation générant un code peu prédictible (e.g. allocation dynamique de la mémoire) des passes, pré-existantes ou non, générant un code plus prédictible (e.g. allocation statique de la mémoire).

Table des acronymes

Nous fournissons ici une table des différents acronymes utilisés dans cette thèse. Seuls les noms communs y figurent, et non les noms propres (e.g. MLIR, IREE, LLVM).

- **ANN** – Réseau de neurones artificiels (*Artificial Neural Network*).
- **BB** – Bloc de base (*Basic Block*).
- **CNN** – Réseau de neurones convolutionnel (*Convolutional Neural Network*).
- **DL** – Apprentissage profond (*Deep Learning*).
- **DNN** – Réseau de neurones profond (*Deep Neural Network*).
- **ESN** – *Echo State Network*, une classe particulière de réseaux de neurones récurrents.
- **FFT** – Transformée de Fourier Rapide (*Fast Fourier Transform*).
- **HPC** – Calcul Haute-Performance (*High-Performance Computing*).
- **IA** – Intelligence Artificielle.
- **IFFT** – Transformée de Fourier rapide inverse (*Inverse Fast Fourier Transform*).
- **IR** – Représentation intermédiaire (*Intermediate Representation*).
- **LSTM** – *Long-Short Term Memory*, une classe particulière de réseaux de neurones récurrents.
- **ML** – Apprentissage machine (*Machine Learning*).

- **MLP** – Perceptron multi-couche (*Multi-Layer Perceptron*).
- **OS** – Système d'exploitation (*Operating System*).
- **RL** – Apprentissage par renforcement (*Reinforcement Learning*).
- **RNN** – Réseau de neurones récurrent (*Recurrent Neural Network*).
- **RTE** – Temps-Réel Embarqué (*Real-Time Embedded*).
- **SSA** – *Static Single Assignment*.
- **SCFG** – Graphe flot de contrôle séquentiel (*Sequential Control Flow Graph*).
- **TCN** – Réseau convolutionnel temporel (*Temporal Convolutional Network*), une classe particulière de réseaux de neurones convolutionnels.
- **WCA** – Analyse au pire cas (*Worst Case Analysis*)
- **WCET** – Pire cas de temps d'exécution (*Worst Case Execution Time*).

Bibliographie

[1] Plato definition study report. https://platomission.files.wordpress.com/2018/05/plato2-rb.pdf, Ap 2017.

[2] Kws-streaming. https://github.com/google-research/google-research/tree/master/kws_streaming, Retrieved on 26/09/2022.

[3] Arinc 653 : Avionics application software standard interface. part 1 - required services. revision 3, 2010.

[4] J.L. Ba, J.R. Kirosand, and G.E. Hinton. Layer normalization. Research report arXiv :607.06450, ArXiV, Jul 2016. https://arxiv.org/abs/1607.06450.

[5] B. Bakker. Reinforcement learning with long short-term memory. In *Proceedings of the 14th International Conference on Neural Information Processing Systems : Natural and Synthetic*, pages 1475–1482, Jan 2001.

[6] G. Barthe, D. Demange, and D. Pichardie. A formally verified ssa-based middle-end : Static single assignment meets compcert. In *Proceedings of the 21st European conference on Programming Languages and Systems*, pages 47–66, Mar 2012.

[7] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. LeGuernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), Jan 2003.

[8] J.L. Bergerand, P. Caspi, D. Pilaud, N. Halbwachs, and E. Pilaud. Outline of a real time data flow language. In *Proceedings RTSS*, Dec 1985.

[9] D. Biernacki, co J.-L. Cola G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES '08 : Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130, Jun 2008.

[10] D.M. Botin-Sanabria, A-S. Mihaita, R.E. Peimbert-Garcia, M.A. Ramirez-Moreno, M.A. Ramirez-Mendoza, and J. de J. Lozoya-Santos. Digital twin technology challenges and applications : A comprehensive review. *Remote Sensing*, 14(6), March 2022.

[11] P. Boulet, A. Darte, G-A. Silber, and F. Vivien. Loop parallelization algorithms : From parallelism extraction to code generation. *Parallel Computing*, 24(3-4) :421–444, May 1998.

[12] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for lustre. In *PLDI 2017 : Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–601, Jun 2017.

[13] M. Brandalero, M. Ali, L. Le Jeune, H.G. Munos Hernandez, M. Veleski, B. Da Silva, J. Lemeire, K. Van Beeck, A. Touhafi, T. Goedemé, N. Mentens, D. Göhringer, and M. Hübner. Aitia : Embedded ai techniques for embedded industrial applications. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, Sep 2020.

[14] G. Brockman, V. CHeung, L. Petterson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. Technical Report arXiv :1606.01540, ArXiV, Jun 2016. `https://arxiv.org/abs/1606.01540`.

[15] T. Carle, M. Djemal, D. Genius, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Zhen Zhang. Reconciling performance and predictability on a many-core through off-line mapping. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014.

[16] T. Carle and D. Potop-Butucaru. Predicate-aware, makespan-preserving software pipelining of scheduling tables. *ACM Transactions on Architecture and Code Optimization*, 11(1) :1–26, Feb 2014.

[17] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *1999 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Oct 1999.

[18] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre : A declarative language for programming synchronous systems. In *Proceedings POPL*, 1987.

[19] G. Chen, C. Parada, and G. Heigold. Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014.

[20] J. Chowdhury. Rtneural : Fast neural inferencing for real-time systems. Technical Report arXiv :2106.03037, ArXiV, 2021. `https://arxiv.org/abs/2106.03037`.

[21] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, and F. Plateau. N-synchronous kahn networks : a relaxed model of synchrony for real-time systems. In *POPL 2006*, pages 180–193, 2006.

[22] J-L. Colaço and M. Pouzet. Clock as first class abstract types. In *EMSOFT 2003*, pages 134–135, 2003.

[23] Z. Cui, W. Chen, and Y. Chen. Multi-scale convolutional neural networks for time series classification. Research report arXiv :1603.06995, ArXiV, 2016. `https://arxiv.org/abs/1803.10122`.

[24] Y. Le Cun. *Quand la machine apprend*. Odile Jacob, 2019.

[25] Y. Le Cun. A path towards autonomous machine intelligence, 2022. `https://openreview.net/pdf?id=BZ5a1r-kVsf`.

[26] Y. Le Cun and Y. Bengio. *The handbook of brain theory and neural networks*, chapter Convolutional networks for images, speech, and time series, page 255–258. Oct 1998.

[27] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 1998.

[28] Y. Le Cun, B. Boser, J.S. Denker, R.E. Howard, W. Habbard, L.D. Jackel, and D. Henderson. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, volume 2, 1989.

[29] R. Cytron, J. Ferrante, B.K. Rosen., M.N. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, Oct 1991.

[30] D. Demange and D. Pichardie Y. Fernández de Retana. Semantic reasoning about the sea of nodes. In *CC 2018 : Proceedings of the 27th International Conference on Compiler Construction*, pages 163–173, Feb 2018.

[31] J. Dennis. First version of a dataflow procedure language. In *Symposium on Programming*, pages 362–376, Ap 1974.

[32] J. Dennis and D.P. Misunas. A preliminary architecture for basic data-flow processor. In *ISCA '75 : Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, Jan 1975.

[33] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. Efficient parallelization of large-scale hard real-time applications. Research report RR-9180, INRIA Paris, Jun 2018.

[34] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. *ACM Transactions on Architecture and Code Optimization*, 16(3) :24 :1–24 :27, Sep 2019.

[35] N.P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA '17 : Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[36] H.I. Fawaz, G. Forestier, J.Weber, L. Idoumghar, and P-A. Muller. Deep learning for time series classification : a review. *Data Mining and Knowledge Discovery*, 33, Jul 2019.

[37] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl. Blasfeo : Basic linear algebra subroutines for embedded optimization. *ACM Transactions on Mathematical Software*, 44(4), Dec 2018.

[38] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages : application to arrays in a Lustre compiler. In *LCTES '12*, May 2012.

[39] A. Gholami, S. Kim, Z. Dong, Z. Yao, M.W. Mahoney, and K. Keutzer. *Low-power computer vision*, chapter A Survey of Quantization Methods for Efficient Neural Network Inference. 2022.

[40] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *JMLR Proceedings*, volume 9, pages 249–255, 2010.

[41] I. Grondman, L. Busoniu, G.A.D. Lopes, and R. Babuska. A survey of actor-critic reinforcement learning : Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6) :1291–1307, November 1996.

[42] A. Gruin, T. Carle, H. Cassé, and C. Rochange. Speculative execution and timing predictability in an open source risc-v core. In *2021 IEEE Real-Time Systems Simposium*, pages 393–404, 2021.

[43] D. Ha and J. Schmidhuber. Recurrent world models facilitate policy evolution. In *Neural Information Processing Systems 2018*, 2018.

[44] D. Ha and J. Schmidhuber. World models. Research report arXiv :1803.10122, ArXiV, March 2018. `https://arxiv.org/abs/1803.10122`.

[45] N. Halbwachs. A synchronous language at work : the story of Lustre. In *Proceedings Memocode*, 2005.

[46] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. In *Proceedings of the IEEE*, volume 79, Sep 1991.

[47] N. Hallbwachs. *Synchronous Programming of Reactive Systems*. Springer, 1993.

[48] D. Harel and A. Pnueli. *Logics and Models of Concurrent Systems*, chapter On the development of reactive systems, pages 477–498. 1985.

[49] M. Hassan, A. M. Kaushik, and H. Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 335–246, 2017.

[50] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[51] S. Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8) :1735–1780, Dec 1997.

[52] S. Ioffe and C. Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. In *ICML'15 : Proceedings of the 32nd International Conference on International Conference on Machine Learnin*, volume 37, pages 448–456, Jul 2015.

[53] A.G. Ivakhnenko and V.G. Lapa. Cybernetic predicting devices, 1965. `https://www.gwern.net/docs/ai/1966-ivakhnenko.pdf`.

[54] R.A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4) :295–307, 1998.

[55] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Research report, German National Research Institute for Computer Science, Jan 2001.

[56] Jax. `https://jax.readthedocs.io/en/latest//`, Retrieved on 27/09/2022.

[57] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning : A survey. *Journal of Aritificial Intelligence Research*, 4, 1996.

[58] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, 1974.

[59] R. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3) :13–22, March 1993.

[60] Keras. `https://keras.io/`, Retrieved on 07/09/2022.

[61] B.R. Kiran, I. Sobh, V. Talpaert, and P. Mannion. Deep reinforcement learning for autonomous driving : A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6), Jun 2022.

[62] N. Kyrtatas, D.G. Spampinato, and M. Püschel. A basic linear algebra compiler for embedded processors. In *DATE '15 : Proceedings of the 2015 Design, Automation and Test in Europe*, March 2015.

[63] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager. Temporal convolutional networks for action segmentation and detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1003–1012, 2017.

[64] J. Lee, Y. Kim, Y. Song, C-K. Hur, S. Das, D. Majnemer, J. Regehr, and N.P. Lopes. Taming undefined behavior in llvm. In *PLDI 2017 : Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 633–647, Jun 2017.

[65] M. Lermer and C. Reich. Creation of digital twins by combining fuzzy rules with artificial neural networks. In *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, Oct 2019.

[66] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[67] H.C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik, and S. Laurenzo. Tinyiree : An ml execution environment for embedded systems from compilation to deployment. Technical Report arXiv :2205.14479, ArXiV, 2022. `https://arxiv.org/abs/2205.14479`.

[68] I. Lopez-Espejo, Z-H. Tan, J.H.L. Hansen, and J.Jensen. Deep spoken keyword spotting : An overview. *IEEE Access*, 10 :4169–4199, Jan 2022.

[69] D.B. Malkoff. A neural network for real-time signal processing. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, pages 248–255, Jan 1989.

[70] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5 :115–133, 1943.

[71] Multi-level intermediate representation compiler framework (MLIR). https://mlir.llvm.org/, Retrieved on 14/09/2021.

[72] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. Research report arXiv :1312.5602, ArXiV, Dec 2013. https://arxiv.org/abs/1312.5602.

[73] W. Niu, X. Ma, Y. Wang, and B. Ren. 26ms inference time for resnet-50 : Towards real-time execution of all dnns on smartphone. Research report arXiv :1905.00571, ArXiV, May 2019. https://arxiv.org/abs/1905.00571.

[74] K.J. Ottenstein, R.A. Ballance, and A.B. MacCabe. The program dependence web : A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *ACM SIGPLAN Notices*, 25(6) :257–271, Jun 1990.

[75] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3) :307–338, Sep 2011.

[76] H. Pearce, X. Yang, P.S. Roop, M. Katzef, and B. Strom. Designing neural networks for real-time systems. *IEEE Embedded System Letters*, PP(9), Jul 2020.

[77] K. Pertsch, Y. Lee, Y. Wu, and J.J. Lim. Demonstration-guided reinforcement learning with learned skills. Research report arXiv :2107.10253, ArXiV, Jul 2021. https://arxiv.org/abs/2107.10253.

[78] P.Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, Sep 2017.

[79] S. Pop. *The SSA Representation Framework : Semantics, Analyses and GCC Implementation.* PhD thesis, École des Mines, Paris,

2006. https://pastel.archives-ouvertes.fr/pastel-00002281/document.

[80] D. Potop-Butucaru. *Real-Time Systems Compilation.* PhD thesis, EDITE, 2015.

[81] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel.* Springer, 2007.

[82] Protobuf. https://github.com/protocolbuffers/protobuf, Retrieved on 07/09/2022.

[83] P.S.Roop, H. Pearce, and K. Monadjem. Synchronous neural networks for cyber-physical systems. In *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 1–10, 2018.

[84] Pytorch. https://pytorch.org/, Retrieved on 07/09/2022.

[85] S. Ramstedt and C. Pal. Real-time reinforcement learning. In *NIPS'19 : Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 3073–3082, Dec 2019.

[86] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *POPL '88 : Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, Jan 1988.

[87] F. Rosenblatt. The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6) :386–408, 1958.

[88] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(9), 1986.

[89] O. Rybakov, N. Kononenko, N. Subrahmanya, M. Visontai, and S. Laurenzo. Streaming keyword spotting on mobile devices. Research report arXiv :2005.06720, ArXiV, May 2020. https://arxiv.org/abs/2005.06720.

[90] S.Arık, M.Kliegl, R.Child, J.Hestness, A.Gibiansky, C. Fougner, R. Prenger, and A. Coates. Convolutional recurrent neural networks for small-footprint keyword spotting. In *Interspeech 2017*, pages 1606–1610, Aug 2017.

[91] R.W. Schlegel and A.J. Smit. Climate change in coastal waters : time series properties affecting trend estimation. *Journal of Climate*, 29 :9113–9124, 2015.

[92] J. Si, A.G. Barto, W.B. Powell, and D. Wunsch. *Handbook of Learning and Approximate Dynamic Programming*, chapter Reinforcement Learning and Its Relationship to Supervised Learning. 2004.

[93] R. Simone, J. Talpin, and D. Potop-Butucaru. *Embedded Systems Handbook 2005*, chapter The Synchronous Hypothesis and Synchronous Languages. 2005.

[94] Long short-term memory networks. `https://fr.mathworks.com/help/deeplearning/ug/long-short-term-memory-networks.html`, Retrieved on 11/08/2022.

[95] Static single assignment book. `https://github.com/pfalcon/ssabook`, Retrieved on 13/09/2022.

[96] R.S. Sutton and A.G. Barto. *Reinforcement Learning : An Introduction.* The MIT Press, 2015.

[97] C.W. Tan, F. Petitjean, E. Keogh, and G.I. Webb. Time series classification for varying length series. Research report arXiv :1910.04341, ArXiV, 2019. `https://arxiv.org/abs/1910.04341`.

[98] Tensorflow. `https://www.tensorflow.org/`, Retrieved on 07/09/2022.

[99] Tensorflow lite. `https://www.tensorflow.org/lite/`, Retrieved on 14/11/2022.

[100] The TensorFlow Authors. Implementation of control flow in tensorflow. `http://download.tensorflow.org/paper/white_paper_tf_control_flow_implementation_2017_11_1.pdf`, Nov 2017.

[101] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3), April 2007.

[102] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989. `https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`.

[103] R.P. Weicker. An overview of common benchmarks. *Computer*, 23(12), Dec 1990.

[104] J. Whitham and N. Audsley. Time-predictable out-of-order execution for hard real-time systems. *IEEE Transactions on Computers*, 59 :1210–1223, Sep 2010.

[105] R.J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4), September 1998.

[106] R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2), June 1989.

[107] X.Yang, P.Roop, H. Pearce, and J.W. Ro. A compositional approach using keras for neural networks in real-time systems. In *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1109–1114, Jun 2020.

[108] K. Zhang, Z. Yang, and T. Basar. *Handbook of Reinforcement Learning and Control*, chapter Multi-Agent Reinforcement Learning : A Selective Overview of Theories and Algorithms. 2021.

[109] H. Zhou and K. Schwan. Dynamic scheduling for real-time systems : Toward real-time threads. *IFAC Proceedings Volumes*, 24(2), May 1991.