Republique Algerienne Democratique Et Populaire
Ministère de l'enseignement Supérieur et de la Recherche Scientifique

Université Ferhat Abbas - Sétif -1-
Faculté De Technologie
Département D'électronique

Thèse intitulée :

Optimisations pour la synthèse haut niveau des systèmes embarqués configurables

---

Présentée par :

Mr DEBBI AIMAD EDDINE
Soutenue Le 15 décembre 2018

Pour l'obtention du diplôme de doctorat en sciences

Devant le jury composé de :

Président:	Pr. Hassam Abdelouahab	Professeur	U.F.A. Sétif 1
Encadreur:	Pr. Ferhat Hamida Abdelhak	Professeur	U.F.A. Sétif 1
Examinateur:	Pr. Hocini Abdesselam	Professeur	U.M.B. M'sila
Examinateur:	Pr. Rouabah Khaled	Professeur	U.B.I. B.B.A
Examinateur:	Pr. Boukezzoula Nacereddine	Professeur	U.F.A. Sétif 1
Examinateur:	Dr. Ayad Mouloud	M.C.A	U.M.O. Bouira

Promotion : 2017/2018

Résumé : Pour bien maitriser les complexités grandissantes des systèmes micro-électroniques et pour assurer des augmentations en productivité, les concepteurs de SoCs devront effectuer la modélisation, la conception, la synthèse et la validation en des niveaux élevés d'abstraction. La synthèse haut niveau ou le high level synthesis (HLS) répond adéquatement aux exigences de productivité auxquelles se trouvent confrontés aujourd'hui les confectionneurs des systèmes embarqués. Le perfectionnement de la qualité des solutions générées par le HLS conduira à promouvoir encore davantage son adoption à grande échelle auprès des industriels et aussi en milieu académique. Dans cette thèse on s'est attaché à déterminer les conduites à adopter lors du développement d'un outil HLS pour rendre beaucoup plus efficaces les solutions générées. Plus spécifiquement on propose des stratégies pour l'exploitation du parallélisme intrinsèque des algorithmes en vue de faire des améliorations franches en latence et en cadence. Les schémas d'interdépendances souvent assez complexes dans les algorithmes limitent potentiellement le pouvoir à faire des exploitations ultimes du parallélisme. On propose une plateforme logicielle où la partie frontale est formée d'un compilateur compatible C et un module d'instrumentation de code permettant une libération fiable du parallélisme. La dissociation du parallélisme est opérée en menant des discriminations explicites des dépendances dans les régions de code critiques faites à l'issue d'une élaboration de graphes de flots de données Data Flow Graphs (DFG). Nos analyses ont conduit à prévoir deux types d'architectures cibles dans la phase d'implémentation qui vient juste succéder aux étapes de transformations et de partitionnement. Il s'est avéré adéquat de considérer des instances de processeurs VLIW et aussi des unités de traitement ou Processing Units (PU) SIMD pour les vectorisations. Le potentiel d'instrumentation du Framework a été démontré en recensant le potentiel parallèle intrinsèque dans la suite de programmes formant le Benchmark CHStone.

Mots clés : Synthèse haut niveau (HLS), Accélérations, Parallélisme, Compilation, Potentiel parallèle intrinsèque, Dépendances de données, Graphes de flots de données (DFG), Instrumentation de code, VHDL.

Abstract : To master the growing complexities of microelectronic systems and to increase productivity, SoC designers should perform modeling, design, synthesis and validation in high levels of abstraction. High Level Synthesis (HLS) provides a good opportunity to overcome the productivity challenges facing embedded systems designers today. The improvement in the qualities of the solutions generated by HLS will lead to promote further its widespread adoption among industry and also in academia. In this thesis we suggested some policies to adopt during the development of an HLS tool to make the generated solutions much more efficient. More specifically, we proposed strategies enabling the ultimate exploitation of the intrinsic parallelism in algorithms to get high order of improvements in latency and throughput. The often complex interdependence schemes in algorithms limit potentially the ability to acquire ultimate benefits from intrinsic parallelism. In the present contribution, a compatible C compiler and a code instrumentation module are proposed as a software platform front-end side for parallelism releasing. The dissociation of parallelism is made by conducting explicit discriminations of dependencies in the critical code regions made after the elaboration of Data Flow Graphs (DFG). Our analyzes have led to consider two types of architecture targets for the implementations step, once we have made the transformation and partitioning steps appropriately. We have found appropriate to consider instances of VLIW processors as well as processing units or SIMD Processing Units (PU) for vectorization. The assessment of the intrinsic parallelism in the subset of the CHStone benchmark programs suite is carried out as a validity proof of the robustness of the proposed instrumentation kernel.

Key words : High Level Synthesis (HLS), Acceleration, Parallelism, Compilation, intrinsic Parallelism assessment, Data dependencies, Data Flow Graphs (DFG), Code instrumentation, VHDL.

Dedicace

A la mémoire de mon père.

A ma mère.

Remerciements

Je tiens à remercier le directeur de thèse, Mr Ferhat Hamida abdelhak, Professeur à l'université de Setif pour bien vouloir accepter de diriger le travail. Qu'il trouve ici toute ma profonde gratitude pour sa patience pendant une assez long durée.

J'adresse mes vifs remerciements à Mr Hassam abdelouahab, Professeur à l'université de Setif pour vouloir nous confier l'honneur de présider le jury d'évaluation de cette thèse. Mes reconnaissances vont également à Mr Boukezzoula nacereddine Professeur à l'université de Setif pour nous honorer et accepter de faire partie du membre du jury.

Mes remerciements sont aussi adressés à Mr Hocini Abdesselam Professeur à l'université de M'sila, Mr Rouabah khaled Professeur à l'université de B.B.A et Mr Ayad Mouloud Maître de conférence à l'université de bouira pour accepter de contribuer à l'évaluation du travail autant que membres du jury.

Je remercie également mon collègue Bakhti Haddi, docteur à l'université de M'sila et mon frère Debbi Ali, Professeur à l'université de M'sila pour le soutien continuel qu'ils mon accordé.

Table des matières

Liste des figures

Liste des tables

Liste des Abréviations

ADPCM Adaptive Differential Pulse Code Modulation

AES Advanced Encryption Standard

ANSI American National Standards Institute

ASIC Application Specific Inegrated Circuit

AST Abstract Syntax Tree

CDFG Control Data Flow Graph

DFG Data Flow Graph

DRF Distributed Register File

DSE Design Space Exploration

DSP Digital Signal Processors

DWARV Delft Workbench Automated Reconfigurable VHDL

EDG Expanded Dependence Graph

FIFO First In First Out

FPGA Field Programmable Gate Arrays

FSM Finite State Machine

GALS/LIS Globally Asynchronous Locally Synchronous/Latency Insensitive System

GCC GNU Compiler Collection

GNU GNU's Not UNIX

GPU Graphic Processing Unit

GSM Global System for Mobile

HLS High Level Synthesis

HPC High Performance Computing

IP Intellectual Property

IR Intermediat Representation

JPEG Joint Photographic Experts Group

LIFO Last In First out

LLVM Low Level Virtual Machine

LRDG Labeled RDG

LUT Look Up Table

MEMU Memoru Unit

NRE Non Recurring Engineering

PE Processing Element

PICO Program In/Code Out

PPA Pipeline of Processing Arrays

PRDG Polyhedral RDG

PU Processing Unit

R&D Research and Development

RDG Reduced Dependence Graph

ROCCC Riverside Optimizing Compiler for Configurable Circuits

RTL Register Transfer Level

SDR Software Defined Radio

SHA Secure Hash Algorithm

SoC System on Chip

SSA Static Signal Assignement

TCL Tool Command Language

TDA Total Distribution Arithmetic

UCLA University of California, Los Angeles

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuits

VLIW Very Large Instruction Word

Introduction

Le perfectionnement des méthodes de conception avait été pour longtemps une préoccupation d'envergure chez les industriels. En technologie VLSI, les densités et la complexité des circuits ont continué à augmenter de façon drastique. Les systèmes à synthétiser devenus potentiellement larges, les concepteurs de puces micro-électroniques étaient contraint de faire des abstractions sur les plans de la spécification, de la synthèse et de la validation. La synthèse haut niveau ou le High Level Synthesis (HLS) a été introduite pour garantir la maitrise de la conception des Systems on Chip (SoC) lorsque ces derniers tendent à se compliquer davantage.

Le HLS se reporte à la totalité du processus où est effectuée la génération automatique de spécifications matériels à partir d'une description comportemental. Le HLS offre d'importantes opportunités pour les industriels. En premier lieu, il leur confie presque le meilleur moyen qui permettra d'atteindre les rapports de productivité imposés par les enjeux de concurrence et qui sont continuellement en croissance. En milieu académique, les chercheurs utilisant le HLS auront l'occasion de vérifier la validité de leur solutions et leur travaux de synthèse dans des délais opportuns. Si les résultats de synthèse pour une proposition donnée ne semblent pas être assez satisfaisants, le chercheur trouvera toujours du temps pour renoncer à une autre synthèse ou à une autre amélioration de la première solution.

Les rapports de productivité achevés avec le HLS sont toujours de loin plus important que ceux acquis avec des synthèses manuels. Cependant, on n'est pas toujours certain et utilisant les outils HLS contemporains d'obtenir des solutions qui sont de qualité nettement meilleurs que celles des synthèses manuelles. Il y a plusieurs facteurs à considérer pour qualifier la qualité d'un travail de synthèse. Lorsqu'un concepteur établit une esquisse manuelle de synthèse, il ne prétend pas certainement à des utilisations démesurées ou abusives des ressources matérielles. Il y a de fortes chances à ce que les synthèses manuelles soient faites avec un minimum d'overheads de ressources par rapport aux synthèses automatiques. D'un autre coté, il y a aussi beaucoup de situations où l'on exige que les solutions générées doivent être perfectionnées en accélération et satisfaire des exigences en latence et en débit. Dans ces cas là, on favorisera bien les solutions qui pourront assurer une perfection en accélération même si elles ne sont pas aussi optimisées du point de vue de l'occupation en ressources.

Actuellement, les outils HLS devront intégrer encore davantage d'optimisations pour assurer un perfectionnement dans la qualité des solutions générées. C'est ainsi que ces outils parviendrons à avoir l'admittance parfaite auprès de tous les utilisateurs incluant industriels, partenaires académiques et particuliers.

Dans cette thèse on décrit les séries de mesures qui vont occasionner le bon perfectionnement en qualité des solutions HLS générées. Ces mesures sont validées par des implémentations menées au sein d'un Framework qu'on a réalisé. Plus spécifiquement on s'attachera aux options qui permettront de faire les améliorations nettes en cadence et en latence. On essayera particulièrement d'exploiter le parallélisme intrinsèque des algorithmes pour faire des accélérations.

Pour pouvoir faire du parallélisme au niveau d'un algorithme, il va falloir tout d'abord faire une analyse d'interdépendance pour déterminer les portions de code qui pourront être assujetti à un parallélisme. Cependant, les schémas d'interdépendances dans les algorithmes sont en générale assez complexes. Il n'est pas concevable de résoudre ces interdépendances de façon intuitive. La discrimination des portions indépendantes et leur dissociations doit être faite de façon instrumental.

À travers des cycles d'instrumentations de code appliqués à un ensemble d'applications arbitraire on est convenu à considérer trois dispositions différentes pour implémenter le parallélisme. Chaque forme de parallélisme a été censée être appliquée pour une variante de région de code particulière. Les régions de code composées par des boucles à nombre d'itérations restreint et qui contiennent des schémas d'interdépendances complexes sont déjà assez fréquentes dans les applications standards telles que (AES, SHA, JPEG, GSM, ADPCM...). Pour ces régions de code, on a prévu de faire l'interception explicite des interdépendances en construisant les graphes de flots de données (Data Flow Graphs DFGs). On établit ensuite des partitionnements particuliers des opérations pour former des sous-ensembles de traitements indépendants. Pour ces régions de code, l'utilisation des méthodes d'analyses et de transpositions d'indices de boucles ne va pas convenir parce que le plus souvent il y a des indirections pour les interdépendances, les structures sont assez complexes avec des irrégularités pour les indices de boucles, il y a abondamment de structures et variables dynamiques et surtout les coûts d'implémentations pourront être plutôt très importants tels qu'ils ne conduirons pas à des solutions vraiment valables.

Les méthodes d'analyse et de transpositions des indices de boucles peuvent convenir pour les boucles à fortes intensités d'itérations. Pour ce deuxième type de régions on a proposé un nouveau algorithme de transposition d'indices qui a la particularité d'être applicable dans beaucoup de scénarios et deuxièmement et surtout son coûts d'implémentation peut bien être consi-

déré comme parfait par rapport toujours aux coûts des autres algorithmes. Une troisième démarche particulière a été aussi prévue pour les cibles configurables. Il s'agit ici de faire une première analyse des interdépendances à un niveau de granularité plus large, en l'occurrence le niveau fonction par exemple. L'analyse est ensuite poussée davantage à l'intérieur des fonctions pour extraire davantage de parallélisme. L'implémentation de toutes ces procédures a été rendu possible grâce à l'instrumentation rigoureuse assurée par le Framework proposé. Le Framework intègre en partie frontale un compilateur compatible C propre qu'on a reconstruit « from scratch » sans adopter aucune plateforme tierce. Le produit est en totalité propre il n'est nullement lié à des dues ou des licences tierces. Le pouvoir d'instrumentation a été démontré en élaborant les mesures des potentiels parallèles intrinsèques dans la suite de programmes formant le Benchmark CHStone.

Le premier chapitre dans la présente thèse avait pour objectif de clarifier le concept du HLS et de montrer son intérêt. La première partie de ce chapitre a été consacrée pour recenser les points de vue de quelques opérateurs industriels en vue de déterminer les opportunités du HLS et aussi ses chalenges éventuels. Dans la deuxième partie du chapitre, on a évoqué un ensemble d'exemples d'outils HLS qui ont été proposés dans la littérature et qui étaient aussi adoptés chez certaines firmes particulières de façon concrète dans la production des puces. Les auteurs de tous ces outils HLS ont cherché surtout à livrer des plateformes qui devraient être les plus fiables possible. Chaque auteur avait proposé une logique de construction particulière qui pourrait conduire à plus de perfection des solutions générées.

Dans le deuxième chapitre ont été examinées les techniques fréquemment utilisées dans les outils HLS en vue d'optimiser les solutions générées en ce qui concerne le volume et l'occupation en ressources, la consommation d'énergie et surtout la perfection des temps de réponse et des cadences. Une attention particulière dans ce chapitre a été accordée aux méthodes utilisées pour faire des accélérations par exploitation du parallélisme. Une partie toute entière a été consacrée à l'analyse du parallélisme au niveau des boucles.

Dans le troisième chapitre, on avait présenté nos propres suggestions de plans pour l'exploitation du parallélisme intrinsèque en vue de faire des accélérations. On avait apporté des explications indiquant comment ces schémas sont concrétisées pour développer un Framework pour l'auto-parallélisation. On en avait décrit l'architecture et on avait rapporté des résultats d'instrumentation.

Chapitre 1

Le HLS ou la synthèse haut niveau : Opportunités et challenges

Sommaire

Préambule

Les changements rapides dans les protocoles et les applications, la forte orientation envers l'embarqué ont fait ressentir le besoin et l'exigence de concevoir, de synthétiser et de déployer les SoCs dans délais assez raccourcis. Il est maintenant possible de construire des puces microélectroniques qui exécutent en entier des applications importantes et complexes, mais il n'est pas tout à fait à portée de le faire sans dépenses budgétaires massives tout en respectant les limites de délais de production imposés par les aspects de concurrence et de commercialisation.

Pour bien maitriser les complexités croissantes des systèmes microélectroniques et pour assurer des augmentations en productivité, les concepteurs devront effectuer la modélisation, la conception, la synthèse et la validation en des niveaux élevés d'abstraction. Ils devront être en mesure et capable de synthétiser partiellement ou complètement le matériel et les logiciels d'un système embarqué à partir d'une description niveau système. La synthèse haut

niveau ou le HLS (High Level Synthesis) est la génération automatique d'une spécification matérielle d'un système numérique à partir d'une description comportementale.

Le HLS ou la synthèse haut niveau semble être une approche assez agréable qui va permettre l'obtention des niveaux d'amélioration les plus nettes en productivités et garantir pour les firmes de fabrication des SoCs l'avance concurrentiel. Elle permettra aussi d'explorer l'espace de conception de manière efficace en permettant de prouver la faisabilité des algorithmes en des délais raccourcis et permettra aussi d'optimiser les performances.

1.1 La réputation du HLS dans l'industrie (Les points de vue des acteurs industriels)

Nous avons choisi de reporter tout d'abord un ensemble d'opinions et des points de vues concernant le HLS recensé auprès des représentants de quelques firmes attachés à la production des SoCs. Il s'agit de STMicroelectronics, Samsung et France telecom. Ces points de vues aiderons normalement à estimer de façon un peu plus juste l'importance et la nécessité du HLS dans l'industrie, vont montrer quelques enjeux et atout majeurs et donnerons des indications de ce que seront les attentes futurs et les orientations envers lesquelles le HLS est sensé évoluer prochainement.

1.1.1 La Point de vue de Samsung

Les capacités d'un outil de synthèse de haut niveau sont dépendantes étroitement et liées au langage choisi et utilisé comme entrée. Le plus souvent ce sont des variantes C y compris SystemC qui sont adoptées. Ces langages diffèrent les uns des autres sur plusieurs aspects. C/C++ est plus convenable que SystemC lorsqu'on désire décrire le comportement du matériel à un niveau d'abstraction élevé. D'un autre côté, SystemC est plus efficace que C/C++ si l'on souhaite décrire le comportement matériel d'une manière plus précise à l'échelle du bit ou/et avec une spécification temporelle (timing). Un bloc matériel peut être décomposé en un corps de bloc et une interface. Le corps décrit le comportement du bloc alors que l'interface définit le mode de communication avec le monde extérieur du bloc. Une description de haut niveau est préférée pour les corps de bloc, alors qu'une description détaillée et plus précise et incluant un timing est souvent plus convenable pour l'interface du bloc. Ainsi, un outil de synthèse de haut niveau doit fournir les deux variantes de descriptions pour permettre la génération des solutions adéquates à la fois pour les bloques et les interfaces de bloques.

Les outils de synthèse de haut niveau doivent prendre en charge les syntaxes et les commandes communes de C/C++/SystemC qui sont généralement

utilisées pour décrire le comportement matériel au niveau de l'algorithme. Les constructions cibles dans ces langages sont des tableaux, des boucles, des structures dynamiques, des pointeurs, des classes C++, ou des templates C++. Certain outils de synthèse de haut niveau actuels ne peuvent prendre en charge que quelques sous-ensembles de ces structures et non pas tous l'ensemble. Bien que la synthèse de haut niveau soit destinée à convertir automatiquement une spécification algorithmique d'un comportement matériel en une description niveau transfert-registre (RTL pour Register Transfer Level), elle nécessite de faire de nombreux changements au niveau du code et l'adjonction d'entrées supplémentaires par les concepteurs. Un comportement identique spécifié dans des outils de synthèse de haut niveau différents résulte généralement en des conceptions RTL très différentes.

La prise en charge d'une conception comprenant plusieurs domaines d'horloge n'est pas tout à fait assurée par les outils de synthèse de haut niveau actuels. Les blocs avec différents domaines d'horloge doivent être synthétisés séparément, puis intégrés manuellement. La co-optimisation du chemin de données et la logique de contrôle constitue également un challenge. Certains outils permettent des solutions optimisées pour le chemin de données et d'autres arrivent à faire des logiques de contrôle adéquates, mais, le plus souvent ils ne fournissent pas des solutions qui seront à la fois optimisé au niveau de la logique de contrôle et au niveau du chemin de données.

1.1.2 La Point de vue de STMicroelectronics

L'amélioration de la productivité au niveau de la conception devient bien une nécessité confirmée. Les figures 1.1 et 1.2 montrent avec des chiffres les taux de croissance en besoins d'investissements nécessaires mesurées en nombre d'homme par année (men per year) recensé sur environ deux décades. Ces chiffres semblent vouloir être des illustrations des défis majeurs auxquels les industriels des SoC seront fort probablement confrontés dans un futur proche. Les industriels des SoC auront à suivre les rythmes de productivité imposés juste avec un investissement en personnel raisonnable. Ces industriels seront contraint donc d'adopter de nouvelles techniques de conception.

STMicroelectronics a adopté SystemC pour décrire en C/C++ les modèles fonctionnels qui prennent en charge un nombre de types de données, en l'occurrence le type bit adaptées à la fonction de synthèse. Les chargés de la conception affirment décrocher des facteurs de gains en productivité qui s'échelonne ente 5× et 10×. En outre, les chargés du design après avoir expérimenté la synthèse niveau C ne préfèrent guerre revenir au model RTL. Dans ce nouveaux modèle il est possible d'avoir des C-IP tout à fait indépendants des contraintes de mise en œuvre (technologie, débit, paramètres) et qui seront facile à modifier pour s'ajuster à de nouvelles spécifications éventuelles ou seront même facile à resynthétiser. Un autre bénéfice aussi assuré est due

1991	1994	1006	1998	2000	2002	2004	2006	2008	2010
#Gates / Die (50mm2) conservative numbers									
50K	250K	750k	1.5M	2.2M	4M	7.5M	15M	30M	60M
#Gates per Designer per year									
4k	6k	9k	40k	56k	91k	125k	200k	200k	200k
Men / Years per 50 mm2 Die									
~10	~40	~80	~40	~40	~43	~60	~75	~150	~300

FIGURE 1.1 – Les croissances en besoins de productivité [Urard 2008]

au fait que la spécification est aussi réduite en espace d'un facteur de 10.

1.1.3 La Point de vue de France Telecom (R&D)

Implémenter des algorithmes sur des circuits électroniques est une tâche fastidieuse qui implique l'ordonnancement des opérations. Alors que les algorithmes peuvent théoriquement être décrits par des opérations séquentielles, leur implémentation aura besoin de techniques additionnelles qui permettront de profiter du parallélisme inhérent et en conséquence améliorer la latence. Le parallélisme n'a pas été pour longtemps pleinement exploité. Les fournisseurs de processeurs se préoccupaient beaucoup plus du renforcement de fréquence de pilotage. Les accélérations ont été assurées en livrant chaque fois des opérateurs séquentiels plus rapides. l'exploitation du parallélisme reste cependant une alternative possible qui apportera évidemment des gains énormes pour les latences d'algorithmes.

Dans les laboratoires de recherche, les algorithmes innovants sont généralement assez complexes. La complexité d'un algorithme ou une proposition est considérée souvent comme un facteur fondamental qui favorisera carrément son adoption ou son exclusion de l'implémentation. Car, les durées d'implémentation limitent et restreignent l'espace de choix des propositions. Des solutions qui ont des complexités même non pas vraiment excessives auront à être écartées si les contraintes de durée d'implémentation sont inadéquates. Les outils HLS apportent aux chercheurs un moyen de tester un nombre d'algorithmes beaucoup plus important en accélérant drastiquement la phase de mise en œuvre. La faisabilité des algorithmes est alors facilement prouvée, et les algorithmes sont caractérisés plus rapidement en terme de latence, de vitesse et d'occupation en espace.

Alors que l'implémentation sur circuits était à l'origine réservée aux spécialistes du domaine, avec les outils HLS la synthèse sera à la portée d'une gamme plus large de développeurs. Dans le traitement du signal, par exemple,

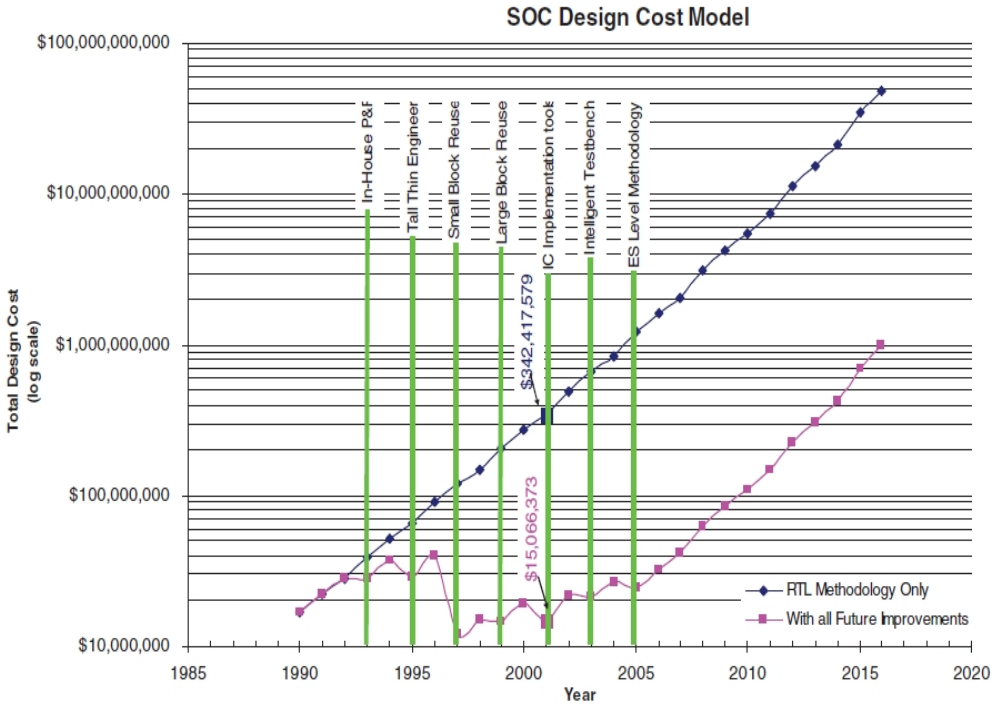

FIGURE 1.2 – Croissances des couts liés aux besoins de productivité [Gupta 2008]

on se permettra une mise en œuvre plus rapide d'algorithmes sur FPGA pour étudier leur comportement dans un environnement plus réaliste.

l'enjeu majeur dans les outils HLS serait de trouver les meilleurs partitionnements d'opérations à partir de la description algorithmique en préservant les bons compromis entre l'intervention utilisateur et la déduction automatique. Les interventions utilisateurs abusives devront être énormément mitigées et les résultats fournis par la déduction automatique devront inclure aussi un sous ensemble de solutions assez acceptables. On aura à décider par exemple si les contraintes architecturales et temporelles devront être incluses à l'intérieur du modèle original ou non. Il n'est pas étrange que du point de vue de l'utilisateur, il serait aussi important de garder le modèle original non synchronisé. Car, moins le modèle est modifiable, moins onéreux serait le cycle de synthèse qui en suit.

1.2 Le HLS par l'exemple

1.2.1 L'outil Catapult

Dans Catapult (*de Mentor Graphics*^{TM}) [Mentor], la transformation d'une description C++ envers une spécification RTL est guidée par l'utilisateur à

travers un processus de synthèse automatisé pour satisfaire des exigences en performance/espace/puissance. Catapult génère du RTL assumant qu'il y ait une connaissance détaillée du timing et des délais associés aux composants cibles pour éliminer une grande partie du travail de conjecture qui pourra être impliqué dans la génération de la micro architecture.

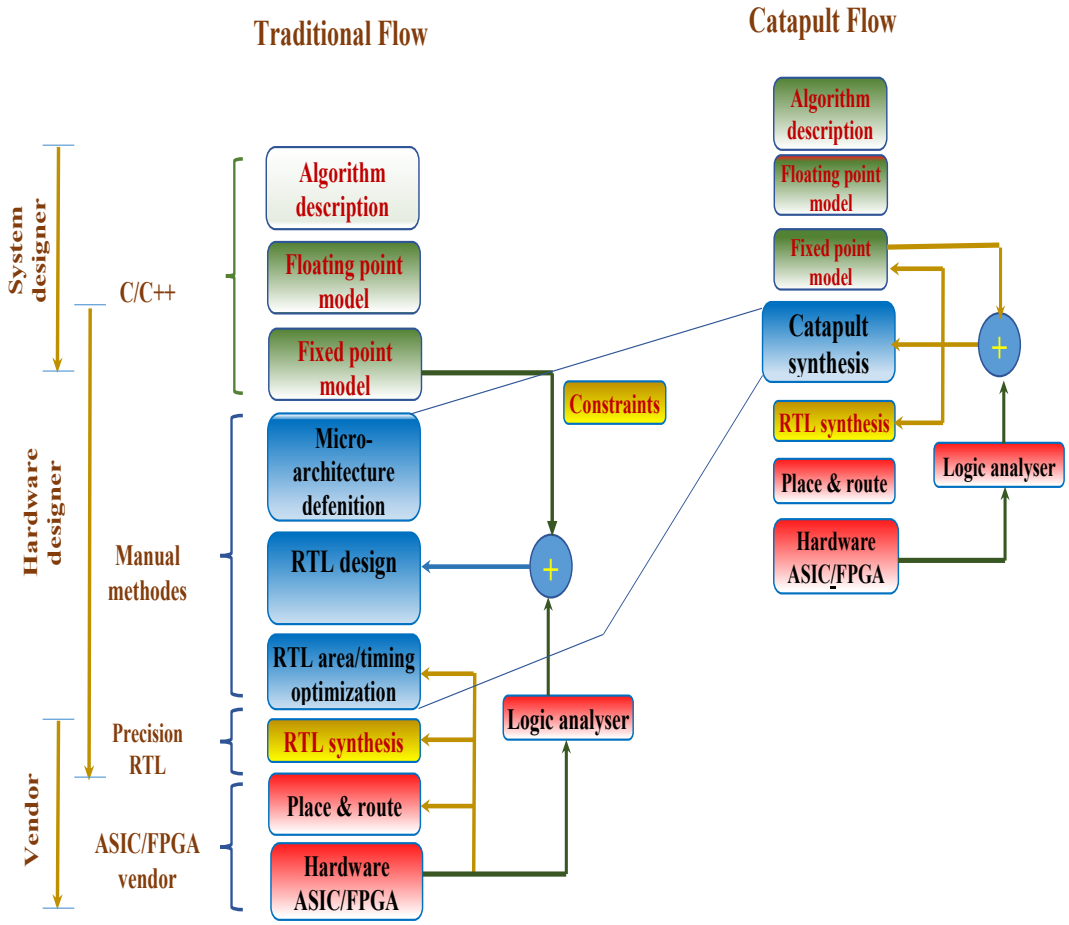

FIGURE 1.3 – Flot de synthèse dans Catapult (adaptée de [Mentor])

1.2.1.1 Le flot de synthèse

Les auteurs de Catapult affirment que l'outil permet de générer des solutions de qualité en des temps de conception considérablement réduits. La synthèse est menée en quatre étapes majeures.

1. Ecriture et test du code C

   Dans Catapult, les concepteurs commencent par décrire le comportement souhaité en utilisant purement le ANSI C/C++. Cette description est une spécification purement algorithmique qui n'inclut absolument aucune information qui concerne le timing, les aspects de concurrence ou de technologie cible. Cela rend beaucoup plus compact la représentation qu'avec la spécification RTL traditionnel qui est couramment donnée en des langages tels que VHDL, Verilog ou SystemC.

   Les ingénieurs peuvent se concentrer ainsi sur ce qui compte le plus : L'algorithme, et non pas les détails d'implémentation de bas niveau. Les vitesses d'exécution des programmes C++ compilés par l'hôte permettent une analyse et des vérifications approfondies qui peuvent être menés en des délais nettement raccourcis par rapport à ceux induits dans les simulations RTL.

2. Définition des contraintes de synthèse

   Une fois satisfait de l'algorithme, le concepteur définit les contraintes de synthèse. Ce processus ne prend en moyenne que quelques minutes et peut être répété plusieurs fois pour le même design. La première étape consiste à spécifier la technologie cible et la fréquence d'horloge souhaitée. Ces détails fournissent à Catapult les informations nécessaires pour construire un calendrier de conception.

   À l'étape suivante, des contraintes individuelles peuvent être appliquées à la conception des E/S, à la manipulation des boucles, à la gestion du stockage et des ressources. Avec cet ensemble de contraintes, le concepteur peut explorer l'espace de conception architecturale. Les directives de synthèse qui concernent les interfaces sont utilisées pour indiquer comment les groupes de données sont déplacés depuis et vers la cible. Les directives de boucle sont spécifiées pour accroitre les accélérations en exploitant le parallélisme inhérent. Les directives de mémoire sont utilisées pour contraindre et définir les plans de stockages. Les directives de ressources sont utilisées pour contrôler le nombre ressources dispensées pour la conception. Toutes ces contraintes peuvent être définies ou bien de manière interactive, grâce à l'outil intuitif fourni dans l'outil interface utilisateur graphique ou encore en mode batch avec des scripts Tcl (Tool Command Language).

3. Analyse conjointe Algorithme-Architecture

   Le synthétiseur Catapult fournit un ensemble complet d'algorithmes et d'outils d'analyse et de conception. Particulièrement il procure le diagramme de Gantt qui permet des aperçus complets sur les profils de boucle, les dépendances algorithmiques et les unités fonctionnelles dans

la conception. Dans cette perspective, l'algorithme est continuellement analysé en ce qui concerne le matériel cible et la vitesse d'horloge, car ces contraintes peuvent avoir des effets majeurs sur la structure d'un algorithme.
En utilisant les graphiques Gantt, les concepteurs peuvent facilement faire des ajustements dans la paire Algorithme-Architecture qui répondront aux objectifs initialement fixés. Ces vues sont également très utiles pour le suivi des goulets d'étranglement de conception et pour permettre les optimisations nécessaires dans des zones spécifiques. Avec ces outils d'analyse, les concepteurs peuvent toujours comprendre pleinement pourquoi et comment différentes contraintes de synthèse ont un impact sur la conception et sur les résultats réels. Les concepteurs maintiennent le contrôle en continu et interagissent itérativement pour converger vers les résultats optimaux.

4. Génération et vérification du code RTL
   Une fois les contraintes de synthèse appropriées définies, Catapult génère un code RTL adapté pour les outils de synthèse ASIC ou FPGA. Dans l'approche de synthèse traditionnelle, la génération du RTL à partir de la spécification est effectuée manuellement. Un processus qui peut nécessiter plusieurs mois pour être accomplis. Dans Catapult, la génération du RTL devient une affaire de quelques minutes. Catapult génère les netlists VHDL, Verilog ou SystemC qui sont en conformité avec les paramétrages de l'utilisateur.
   Différents rapports sont également produits. ils fournissent les strictes informations à la fois vis-à-vis du matériel et de l'algorithme. Catapult intègre aussi les moyens qui automatisent la vérification et la validation des sorties netlists HDL par rapport aux entées originales en C/C++. Ceci est accompli en enveloppant la sortie Netlist en des classes de modules SystemC à instancier avec le code C/C++ original. Les mêmes stimuli d'entrée sont appliqués à la fois au code original et le code synthétisé tout en positionnant un comparateur qui validera les résultats chaque fois que les sorties sont identiques.

1.2.1.2 Les optimisations dans Catapult

Dans Catapult, L'utilisateur pourra optimiser la synthèse en agissant principalement sur les quatre contrôles suivants :

1. Le codage C/C++
   Catapult prend en charge un très large sous-ensemble du langage C++ANSI. Les fonctions C/C++ de haut niveau synthétisées peuvent appeler

d'autres sous-fonctions, qui seront ou bien intégrées par une mise en ligne (inlining) ou bien conservées à leurs niveaux de hiérarchie. Le design peut également contenir des variables statiques qui préseve un certain état entre les invocations de fonctions. Les "if" et "switch" dans les structures conditionnelles sont aussi supportées, ainsi que les déclarations de boucles "for", "do" et "while". La restriction notable est que le code doit être statiquement déterminable, ce qui signifie que toutes ses propriétés doivent être définies au moment de la compilation. Par exemple, les allocations/désallocations de mémoire dynamique (malloc, free, new, delete) ne sont pas prise en charge. Les types de données composés tels que les classes, les structures et les tableaux sont entièrement pris en charge dans le processus de synthèse. Le paramétrage via les templates C++ est également supporté. La combinaison des classes et des modèles fournit un mécanisme puissant facilitant la réutilisation du design. Les concepteurs de matériel sont habitués aux types de données précis (échelle du bit) avec les langages VHDL et Verilog. Ce type de donnée est recommandé pour se permettre les solutions efficaces. Le concept de longueur arbitraire affecté aux types de données entiers et à virgule fixe (échelle du bit) offre un moyen facile pour modéliser la précision binaire statique avec un minimum d'overheads.

2. Synthétiser l'interface

   Dans le code source C/C++, les arguments passés aux fonctions primaires dans le haut de la hiérarchie sont les principaux paramètres pris en considération pour la définition des ports d'interface. Catapult établi une correspondance entre les arguments des fonctions C/C++ et les E/S du système cible. Ensuite, le concepteur devra appliquer les contraintes de synthèse d'interface pour définir les propriétés de chaque port matériel. Avec cette approche, les concepteurs peuvent construire n'importe quel type d'interface matérielle. Les directives de synthèse d'interface permettent le contrôle d'autres paramètres : la bande passante, la synchronisation, les handshakes et d'autres aspects de protocoles. L'utilisateur peut aussi par exemple :

   — Définir une prise de contrôle partielle ou complète, sur les signaux d'interface.
   — Mapper des tableaux sur des fils, des mémoires, ou des bus.
   — Ajouter au design des drapeaux optionnels start/done, des signaux d'E/S spécifiques au matériel, tels que l'horloge, des triggers de réinitialisation, d'activation ou de suspensions.

   Les signaux n'ont pas besoin d'être modélisés et sont ajoutés automatiquement en fonction des contraintes de l'utilisateur. De cette façon, l'algorithme C/C++ synthétisé reste purement fonctionnel et n'est pas

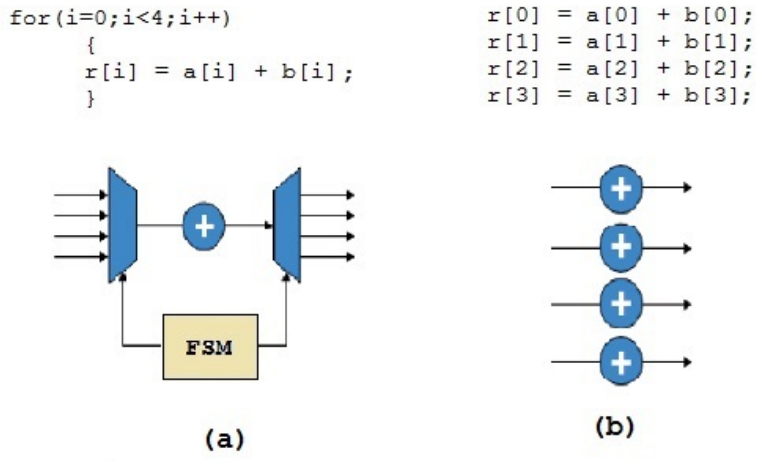

FIGURE 1.4 – Dés-enroulement de boucle pour parallélisme (a). Additions en séquence. (b). Additions en parallèle.

contraint à intégrer des informations spécifiques à l'interface. Le même code peut être réutilisé encore plusieurs fois en fonction de nouvelles exigences d'interface en terme de (bande passante, protocole, etc.)

3. Le contrôle des boucles

   Le dés-enroulement de boucles permet d'exposer le parallélisme qui existe à travers les différentes itérations. Les boucles sont dés-enroulées partiellement ou intégralement. L'exemple de la figure 1.4 présente une simple boucle pour l'addition de deux vecteurs à quatre valeurs chacun. Si la boucle est maintenue sans être dés-enroulée, Catapult génère une architecture en série. Un seul additionneur sera affecté pour implémenter les quatre ajouts.

   L'additionneur est exploité donc en temps partagé, et la logique de contrôle dédiée est construite en conséquence. En supposant que le multiplexage, l'addition elle-même et le démultiplexage tiennent tous ensemble en un cycle horloge, alors l'exécution de la boucle en totalité nécessitera quatre cycles horloge. Si la boucle est entièrement dés-enroulée comme montrée dans la partie b., on devra utiliser quatre additionneurs, mais le temps d'exécution sera nettement réduit. Le même traitement tiens juste en un cycle horloge.

   Le dés-enroulement est appliqué en définissant des contraintes de synthèse qui conduira principalement à produire quatre copies du corps de la boucle. Catapult exploite donc le parallélisme au niveau opérations pour construire une implémentation entièrement parallèle du même al-

gorithme. On peut opter aussi au dés-enroulement partiel si la synthèse est compromise par des contraintes de ressources ou de puissance.

Listing de code 1.1

Boucles non fusionnées		Code résultant par fusion des boucles

$for(i = 0; i < 32; i++)$ {
$\quad a[i] = b[i] * c[i];$
}
$for(i = 0; i < 16; i++)$ {
$\quad z[i] = a[i] + x[i];$
}
	$\Longrightarrow$	$for(i = 0; i < 32; i++)$ {
		$\quad atmp = b[i] * c[i];$
		$\quad if(i < 16)$
		$\quad\quad z[i] = atmp + x[i];$
		}

La fusion de boucles permet aussi d'exposer le parallélisme. Cette technique s'applique à des boucles séquentielles pour créer une boucle unique équivalente. Cette transformation est utilisée pour réduire la latence et la consommation en ressources. Le concepteur décide s'il devrait activer la fusion ou la désactiver pour préserver la lisibilité du code et l'indépendance du matériel. Dans le listing de code 1.1 est donné un exemple d'algorithme qui admet l'application de la fusion. L'outil de synthèse par cette technique assure à un certain point une économie en ressources de stockage. Lorsque la fusion est appliquée, dans cet exemple les valeurs de "a" produites dans la première boucle peuvent être directement consommées par la deuxième boucle. Le stockage du vecteur "a" en totalité n'est pas alors nécessaire et une minimisation en espace est possible.

4. Synthèse hiérarchique
   La bonne intégration des blocs individuels dans un sous-système est l'un des principaux défis dans la conception de puces. Avec le concept de la synthèse hiérarchique, Catapult prétend permettre une considérable simplification de conception et d'intégration de sous-systèmes complexes et des multi-blocs de façon tout à fait fiable.
   Alors que le dés-enroulement de boucles exploite le parallélisme au niveau des instructions et la fusion de boucles exploite le parallélisme au niveau des boucles, la synthèse hiérarchique vise à exploiter le parallélisme au niveau des tâches. Dans Catapult, l'utilisateur peut spécifier quels appels de fonctions doivent être synthétisés en tant qu'entités hiérarchiques. Les arguments de la fonction hiérarchique définissent les données en flux du système, et Catapult va construire toute la logique de synchronisation et de communication inter-bloc. La synthèse hiérar-

chique incorpore le concept de pipelining et l'applique aux fonctions pour accroitre la cadence du système synthétisé.

1.2.2 L'outil Autopilot

AutoPilot [Cong 2011] (d'*AutoESL Design Technologies*^{TM} acquise par *Xilinx*^{TM} [Xilinx a] en 2011) était prévu pour réformer xPilot [Cong 2006], un outil développé à l'origine à l'University of California, Los Angeles (UCLA). Il permet de générer automatiquement du code RTL efficace à partir de descriptions C, C++ ou SystemC et optimiser simultanément la logique, les interconnexions, la performance et la puissance. Les expériences préliminaires montrent des résultats intéressants pour un large éventail d'applications, y compris la synthèse matérielle, l'exploration de l'espace de synthèse et le calcul reconfigurable accéléré. Beaucoup de facteurs favorise aujourd'hui le recourt a l'automatisation de la conception. Les propriétés essentielles d'Autopilot sont :

— Le processus de conception est unifié avec C/C++/SystemC. AutoPilot accepte trois types d'entrées standard C et les dérivés du C : C, C ++ et SystemC. Il prend également en charge une variété de modèles d'abstraction incluant le modèle fonctionnel pur et non temporisé, le modèle partiellement temporisé, le modèle transactionnel, et le modèle comportemental ou structurel entièrement temporisé. Cette prise en charge d'un éventail large de langages et de modèles d'abstraction permet à AutoPilot de cibler aussi une gamme étendue d'applications nécessitant soit une synthèse matérielle ordinaire ou bien une synthèse spécifique orientée au calcul reconfigurable haute performance.
— Utilisation de compilateurs à technologie de pointe. AutoPilot incorpore en avant plan un compilateur C/C++ commercial de qualité dans le cycle de synthèse. Beaucoup de techniques de compilation de pointe (intra-procédural et inter-procédural) sont utilisés pour analyser, transformer et optimiser agressivement le code fourni en entrée.

1.2.2.1 Flot de synthèse

AutoPilot génère du RTL à partir du code C,C++ et/ou SystemC synthétisable fourni en entrée. Le cycle est composé de quatre grandes phases :

- La compilation et l'élaboration.
- La transformation de code avancée.
- La synthèse du comportement et de la communication.
- La génération de la microarchitecture.

La figure 1.5 décrit l'allure globale du cycle de conception dans AutoPilot. Dans la première étape, la description du comportement est analysée par un

parser compatible GCC, qui prend en charge les types de données bits. Lorsque l'entrée est donnée en SystemC, l'élaboration est invoquée pour définir les ports, les canaux, et les topologies d'interconnexion et aussi pour construire une synthèse riche en détails prise comme modèle.

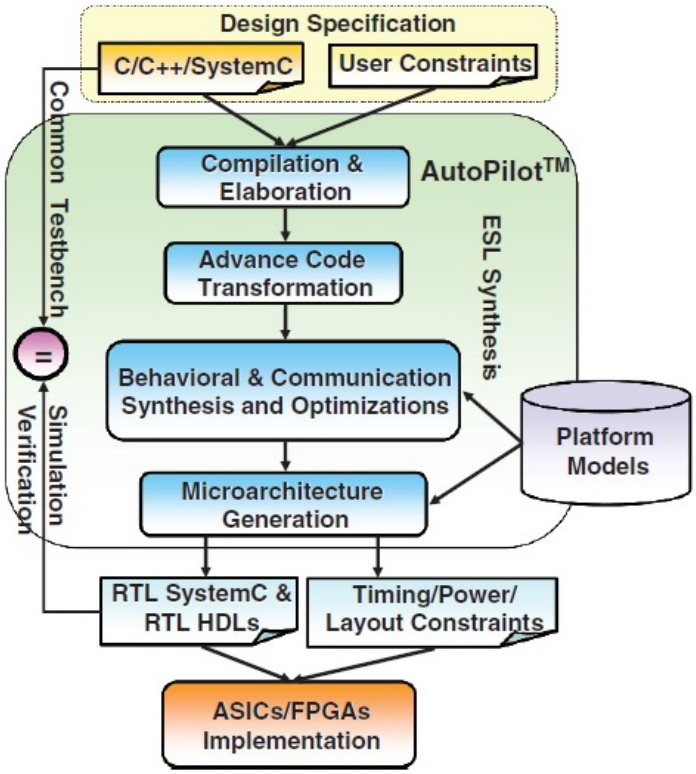

FIGURE 1.5 – Flot de synthèse dans Autopilot [Cong 2015]

AutoPilot applique un ensemble de transformations et d'analyses de codes avancées incluant des techniques de compilation telles que la propagation de constante, l'élimination du code mort. La phase de transformation du code est suivie par la phase de synthèse du système cible de base. AutoPilot effectue des synthèses en incluant des modèles de plates-formes et en œuvrant avec des optimisations centrées sur les interconnexions. Ainsi, un grand nombre de contraintes relevant à la fréquence, la latence et le débit sont alors prise en compte pour générer les microarchitectures optimisées.

Dans la partie frontale d'AutoPilot on s'attache en particulier à trois aspects principaux qui sont : Le langage, les optimisations de compilation et la modélisation de plateformes.

Les langages dans AutoPilot offrent un support pour la plupart des types de données usuelles, les opérateurs conventionnels, les structures, les classes et les structures de contrôle. AutoPilot dans ses versions initiales n'autorise pas l'utilisation de pointeurs dynamiques, les allocations dynamiques de mémoire

et les récursivités de fonctions. Les concepteurs peuvent contrôler entièrement les précisions de données d'une spécification C/C++. AutoPilot prend en charge les types de données à virgule flottante simple et double précision. Il prend en charge aussi les types de données à virgule fixe qui ne sont pas nativement supporté dans les langages C et C++ standards.

Une variété d'optimisations en compilation sont appliquées à la description comportementale pour réduire la complexité du code, pour maximiser la localité des données et pour exposer plus de parallélisme. Les transformations et analyses appliquées sont des variantes de l'ensemble suivant :

* Les optimisations conventionnelles telles que la propagation des constantes, l'élimination du code mort et l'élimination des redondances dans les sous-expressions communes.
* Les réductions en couts d'implémentations par le remplacent des opérations coûteuses (par exemple, les multiplications et les divisions) avec des opérations plus simples (des décalages, des additions et des soustractions).
* Restructuration du code à grain non fin par des transformations au niveau des boucles en appliquant les dés-enroulements, la fusion et l'aplatissement.
* Les analyses des alias et les analyses de dépendance pour réduire les largeurs de données et contrôler les dépendances.

Dans le processus de modélisation, AutoPilot établit une pré-caractérisation des composants. Il désigne les caractéristiques des principaux éléments du système. On spécifie les temporisations, les occupations en zones et la puissance pour les différents types de ressources qui sont entre autres des unités arithmétiques, des additionneurs, des multiplieurs, des mémoires ou piles de registres.

1.2.2.2 Le module de synthèse d'AutoPilot

Le module doit assurer quatre fonctions : l'ordonnancement des traitements, la liaison des ressources, le pipelining, et la synthèse de l'interface.

- L'ordonnancement
  Un ordonnanceur polyvalent est implémenté dans le système AutoPilot pour exploiter le parallélisme au niveau de la description comportemental et pour déterminer les instants auxquels les différents calculs et communications doivent tenir. L'agencement inclut un modèle de programmation régi par formulations mathématiques. L'ordonnanceur modélise l'ensemble de contraintes liées au temps de cycle, les contraintes de latence, les contraintes de débit et les contraintes des temps d'E/S.
- Liaison des ressources
  La liaison de ressources engendre une analyse du réseau d'intercon-

nexion et la logique de pilotage. Son objectif consiste en la réalisation des liaisons optimisée pour des blocs de mémoires, des unités fonctionnelles telles que les unités arithmétiques entières et flottantes, des fonctions transcendantales, des boîtes noires IP, des bancs de registres, etc.
AutoPilot aussi inclut des algorithmes pour générer des microarchitectures dédiées à la liaison de ressources. Il prévoit par exemple une option pour générer une microarchitecture pour les bancs de registres distribués DRF (Distributed Regiter-file).

⋆ Le pipeline
AutoPilot durant les phases d'ordonnancement, durant les phases de liaisons de ressources et durant la génération des micro-architectures applique plusieurs formes de pipeline. Il effectue le pipeline de boucles et le pipeline de fonctions.

⋆ La synthèse de l'interface
Dans AutoPilot les concepteurs ne sont pas obligés d'inscrire explicitement les détails des synchronisations spécifiques à une interface cible dans le code source. Les concepteurs peuvent utiliser les paramètres de fonction standard pour exposer le entrées et sorties aux circuits externes. Par exemple, en fonction des interfaces de communication spécifiées dans la bibliothèque de plateformes, une opération de stockage sur un pointeur scalaire (par exemple, * p = x) peut être transformée en une connexion directe par câble, une écriture FIFO, ou un transfert de bus en pipeline ou prise en rafale.

1.2.3 L'outil Pico

La synthèse dans PICO (Program In/Code Out) [Schreiber 2002, Kathail 2002] fait engager une technologie de compilation agressive, un modèle parallèle basé sur les réseaux de processus Kahn, et des modèles de matériel soigneusement choisis. PICO s'attache au cycle complet de la conception en couvrant les aspects de l'exploration de l'espace de conception (DSE Design Space Exploration) et prévoit des supports pour la vérification et la validation.

La synthèse des SoC pour les applications assez complexes telles que les codecs multistandards et les modems sans fil 3G utilisés dans la nouvelle génération de dispositifs de communication doit satisfaire à la fois des exigences en performance, en espace et en puissance. Il n'est pas tout à fait facile de satisfaire ces exigences tous ensemble à la fois. Par exemple, la radio définie par logiciel SDR (Software Defined Radio) pour le modem sans fil 4G nécessite 10 à 100 Gop (giga opérations par seconde) pour des limites de puissance qui s'échelonne dans la plage 100-500mW, et ce qui fait environ $100 Mop.mW^{-1}$.

Les solutions déjà proposées sur le marché telles que les processeurs universels ou les DSP ne peuvent pas satisfaire ces exigences extrêmes. Les DSP

intégrés ne peuvent pas garantir les hautes performances recommandées. Alors que même si les DSP haut de gamme tels que les IBM Cell processeurs peuvent assurer à un certain point des niveaux de performance assez suffisants, leurs consommations en énergie (qui est dans l'ordre des $10Mop.mW^{-1}$) n'est pas tout à fait acceptable.

La solution consiste à créer des processeurs personnalisés et spécifiques aux applications et des systèmes dédiés pour répondre convenablement au triplet d'exigences (performance-puissance-espace). Typiquement les implémentations matérielles directes peuvent atteindre $100 - 1000Mop.mW^{-1}$ avec un ordre de 2-3 fois d'amélioration en surface et en puissance par rapport aux processeurs embarqués ou DSP. La personnalisation, cependant, a son coût. La synthèse manuelle des applications complexes utilisant les méthodologies de conception classiques sont très coûteuses en termes de temps de conception et en coût d'ingénierie non récurrente (NRE) menant à des SoCs qui nécessitent des soutiens financiers considérables et des durées d'achèvement se prolongeant sur plusieurs années.

La communauté concernée par la conception des SoCs en majorité est parvenue à la conviction que la conception conjointe matériel/logiciel (le co-design), la synthèse de haut niveau et la réutilisation IP haut niveau sont tous ensemble nécessaire pour combler les écarts de productivité en conception. Les applications volumineuses comme les codecs vidéo multistandards contiennent souvent un nombre important de blocs de traitement et des flux de données complexes. La synthèse de telles applications à partir d'une spécification algorithmique en C nécessite au préalable un examen attentif des types d'architectures qui pourront s'accommoder plus à l'automatisation. Les trois choix possibles sont : Les accélérateurs dédiés, les processeurs reconfigurables et les systèmes hybrides.

Les accélérateurs dédiés constituent le meilleur choix lorsque les cibles doivent être à faible puissance. Ils sont non-programmables mais peuvent offrir un nombre limité d'exécution multimodale basée sur la configuration paramétrée. L'automatisation peut se faire en adoptant une approche ascendante ou une approche descendante. Dans l'approche ascendante les blocs individuels sont conçus séparément. Les instructions C et les blocs de base sont mappés sur le chemin de données menant à une structure et à une interconnexion potentiellement irrégulière. Le chemin de données est contrôlé par une machine à états monolithique qui contrôle le flux entre les blocs de base et qui peut être assez complexe.

Dans l'approche descendante, on effectue une prise de vue globale de l'ensemble de l'application et on examine les optimisations qui pourront être occasionnées à travers les blocs et on essaye de trouver le modèle d'architecture approprié pour le chemin de donnée cible. Les processeurs personnalisés ou (spécifiques aux applications) peuvent garantir des performances aussi

meilleures par rapport aux processeurs à usage général. Mais ils restent toujours des circuits programmables. Ils sont favorables surtout dans deux situations. Premièrement, si les exigences en performance ne sont pas très élevées et les contraintes de puissance ne sont pas aussi sévères. Deuxièmement, si les cibles doivent maintenir un degré de flexibilité, sont toujours assujetti à des changements et ne sont pas tout à fait standardisés. Dans une implémentation hybride seront combinés ensemble des processeurs programmables embarqués pour les parties non critiques, des blocks spécialisés pour les parties où sont exigés des accroissements de performances et des minimisations en puissance.

Aspects à prendre en considération

— Le choix de l'architecture et de la micro-architecture ont un grand impact sur la performance, l'occupation en espace et la puissance. Mais il n'est pas facile de prédire de manière fiable cet impact. La synthèse de haut niveau facilite et peut servir de support au DSE (Design Space Exploration) pour trouver un design optimal.

— Un système de synthèse de haut niveau doit être capable de générer des conceptions qui peuvent être compétitives en performance aux conceptions produites manuellement pour qu'il soit largement acceptable dans des environnements de production.

— On ne va pas s'attendre à ce que les concepteurs écrivent les modules de test (les Test-Bench) pour le RTL généré par un système de synthèse. Ils devront plutôt vérifier leur conception au niveau C en utilisant les Test-Bench en C. Le système de synthèse devrait alors automatiquement générer soit un Test-Bench RTL incluant des vecteurs de test ou encore des Test-Bench C-RTL pour une co-simulation. En outre, le système de synthèse devrait fournir aussi un mécanisme pour tester les situations et les conditions non descriptibles par les Test-Bench en C.

— Le RTL généré devrait être cohérent avec les flots de synthèse et méthodologies RTL existantes. le RTL généré devrait être absolument juste est valide pour ne pas engager les concepteurs dans des missions de débogages irréalistes. Car le code généré le plus souvent est assez complexe, voir même, il est non intelligible.

— L'un des avantages intéressant du HLS est son aptitude à faciliter la réutilisation d'une même spécification en C pour différentes conceptions.

1.2.3.1 Flot de synthèse dans PICO

La figure 1.6 décrit le flot de synthèse dans PICO. On utilise une approche hybride telle que discutée précédemment (la section 1.2.3). Typiquement, durant la première étape du processus de synthèse est opéré un partitionnement matériels/logiciels de haut niveau de la fonctionnalité souhaitée.

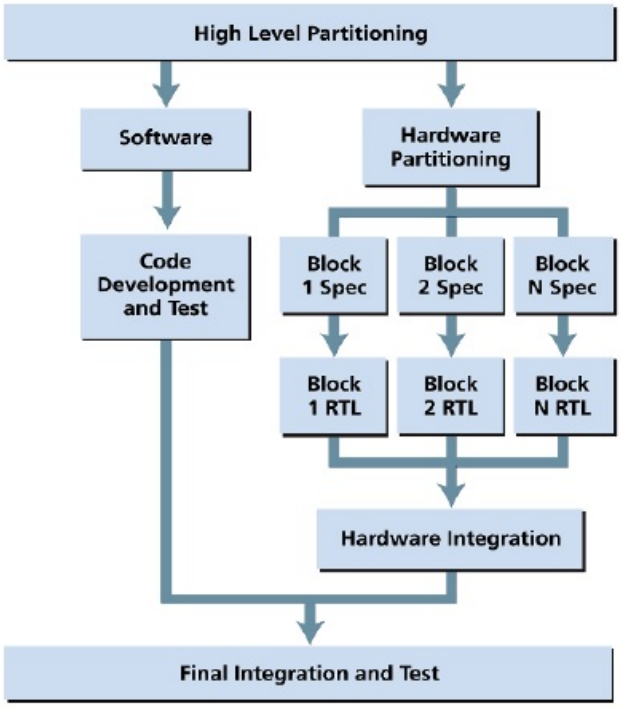

FIGURE 1.6 – Flot de synthèse dans PICO [Aditya 2008]

La génération et la vérification RTL est opérée à partir d'une spécification de haut niveau donnée conjointement avec des informations architecturales qui permettront d'atteindre les niveaux de performances et de puissance exigées et qui peuvent tenir aussi sur les limites d'espace préalablement imposées. Les informations architecturales additionnelles peuvent être fournies à l'outil HLS de différentes manières. Quelques outils basés sur SystemC obligent l'utilisateur à modéliser le partitionnement matériel et les interfaces souhaités directement dans les spécifications qui seront fournis en entrée.

D'autres outils nécessitent que l'utilisateur spécifie des détails architecturaux détaillés et des informations sur les différents composants du matériel en cours de conception en utilisant une interface graphique ou un fichier de conception séparé. Ceci a l'avantage de donner à l'utilisateur un contrôle total de la conception, mais il augmente la charge de la spécification en entrée et rend la spécification moins générale et moins portable. Ceci limite aussi les degrés de liberté de l'outil et fait restreindre ses aptitudes à faires des

optimisations qui permettront d'atteindre les objectifs préalablement fixés. Dans ce cas, l'intervention de l'utilisateur est souvent requise pour le contrôle des interfaces des implémentations multi-bloc aux phases d'intégration et de vérification.

Multiples implémentations à différents compromis de coûts/performances peuvent être générées à partir de la même spécification fonctionnelle, donnant lieu à une possibilité de la réutilisation de l'entrée comme IP algorithmique flexible. L'approche fondamentale dans PICO consiste à utiliser un compilateur de parallélisation avancé conjointement avec un modèle d'architecture configurable optimisé à la compilation pour générer du matériel comme indiqué à la figure 1.7.

La spécification comportementale est fournie en utilisant un sous-ensemble de C ANSI, avec des contraintes de conception supplémentaires, telles que le débit et fréquence d'horloge. La solution RTL est alors créée en deux étapes. Dans la première étape, un compilateur optimisateur analyse l'entrée donnée en haut niveau de spécification pour exposer et exploiter le parallélisme qui se révèlera suffisant pour répondre aux exigences du débit préalablement imposé. Dans la deuxième étape, un synthétiseur architectural configure l'architecture modèle en fonction des besoins de l'application et des objectifs liés à la physique du système cible souhaitée qui sont entre autres le temps de cycle et le coût. La figure 1.8 décrit le flot de conception global pour la création de

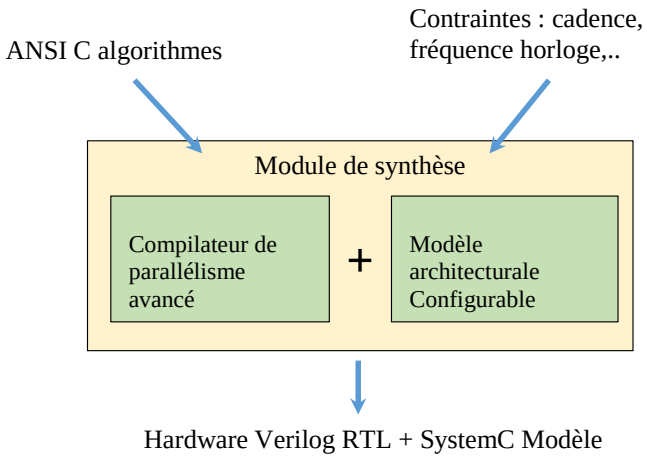

FIGURE 1.7 – Modèle de synthèse dans PICO

blocs RTL à l'aide de PICO. Le système PICO génère le RTL synthétisable, les modules de test (Test-Bench) personnalisés, les scripts de synthèse et de simulation, ainsi que les pilotes logiciels (les drivers) d'intégration pour fonctionner sur le processeur hôte. Le RTL généré peut alors être pris en charge par des outils de simulation, de synthèse, de localisation et de routage dans le SoC grâce à des scripts configurés automatiquement.

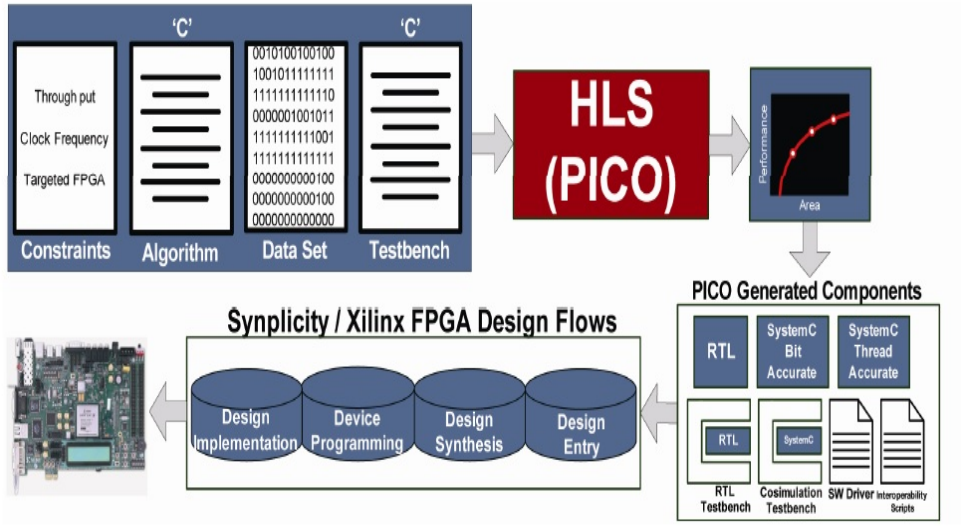

FIGURE 1.8 – Vue d'ensemble globale pour le flot de synthèse dans PICO [Oruklu 2012]

Les contraintes et l'ensemble de l'information qui concernent l'aspect technologique de la cible sont intégrés à la bibliothèque des macro-cellules (macrocells) que le système PICO utilise comme base de données pour les blocs de construction matérielle futures. Cette bibliothèque consiste en des composants RTL synthétisables pré-vérifiés et paramétrés tels que des registres, des additionneurs, des multiplicateurs et des modules d'interconnexions soigneusement fabriqués à la main pour offrir le meilleur rapport qualité/coût. Ces macrocellules (macrocells) sont ensuite paramétrées indépendamment pour diverses cibles technologiques dans les bibliothèques pour définir les différents compromis coût-performance. PICO utilise ces données de caractérisation pour estimer en interne les performances et les coûts en ressources.

1.2.3.2 Le modèle de programmation dans PICO

Un autre objectif important du modèle de programmation de PICO est de permettre à l'utilisateur de spécifier la fonctionnalité matérielle en tant que programme séquentiel. PICO extrait le parallélisme de la spécification fournie en entrée en vue d'atteindre les performances requises sur la base d'une analyse de dépendances de programmes et en mettant en considération les contraintes déjà assignées. Une large gamme d'applications issues de domaines variés tels que l'audio, la vidéo, l'imagerie, la sécurité, le sans fil, et les réseaux peuvent être exprimées en tant que programme C séquentiel qui opèrent des traitements et des transformations sur des tableaux ou des flux de données. Il y a énormément de parallélisme dans ces applications à différents niveaux de granularité et PICO prétend à l'exploiter utilisant diverses techniques.

PICO convertit le code de procédure C en un pipeline matériel où chaque boucle imbriquée est exécutée sur un bloc matériel différent. Cela permet au parallélisme de niveau procédure d'être exploité par un pipeline augmentant ainsi considérablement le débit global.

Au niveau de boucle, PICO fournit un mécanisme pour exprimer le streaming des données qui est synchronisé par négociation bidirectionnelle. Dans le programme C cela se manifeste sous la forme d'un appel intrinsèque d'une fonction qui écrit des données dans un flux et un autre appel d'une autre fonction qui lit les données depuis ce flux. Les flux (Streams) sont utilisés pour communiquer des données entre n'importe quelle paire de boucles aussi longtemps que la causalité temporelle entre la production et la consommation de données est maintenue.

Dans un seul bloc matériel implémentant l'imbrication de boucle, PICO exploite le parallélisme de niveau itérations en faisant l'analyse de dépendance détaillée et en opérant des transformations permettant d'exécuter en parallèle plusieurs itérations même en présence de récurrences sévères. Par la suite, l'itération de boucle transformée est ordonnancée en utilisant des techniques de pipeline logiciel qui exploitent le parallélisme de niveau instruction.

1.2.3.3 Le modèle d'exécution dans PICO

Le compilateur PICO essaye d'exploiter le parallélisme inhérent de l'application sans violer la sémantique séquentielle. Ceci est accompli en adoptant le modèle d'exécution parallèle connu dit réseaux de processus de Kahn [Schreiber 2002, Kathail 2002]. PICO parallélise un programme séquentiel en mappant les boucles à un réseau de processus Kahn implémenté en dur comme bloc matériel communiquant via des flux.

Chaque bloc matériel, exécute aussi une implémentation parallèle du code de la boucle correspondante utilisant toujours les techniques de pipeline, ce qui permet d'exploiter encore le parallélisme de niveau itération et instruction.

1.2.3.4 Modèle d'architecture généré par PICO

La structure générale du système généré par PICO à partir d'une procédure C est décrite par la figure 1.9. Ce modèle architectural est appelé un pipeline de tableaux de traitement (PPA) pour Pipeline of Processing Arrays. Chacune des boucles en niveaux supérieurs d'imbrication des procédures C est affectée à un bloc matériel ou un tableau de traitement (PA) qui communique avec d'autres PA via un ou plusieurs flux, ou via des mémoires partagées.

1.2.3.5 Les transformations C à RTL

La création de modules PPA à partir des procédures C est opérée à travers un ensemble de phases d'analyse et de transformations qui dans l'essentiel

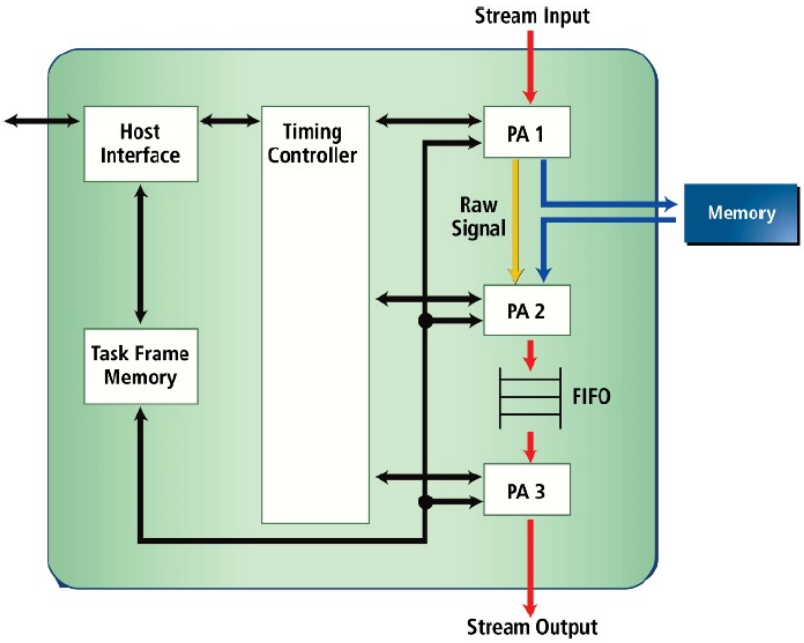

FIGURE 1.9 – Modèle d'architecture en pipeline of processing Arrays PPA [Aditya 2008]

sont :

— Phase de pré-traitement
Dans cette phase l'entrée est analysée pour identifier les boucles qui pourront être mappées en parallèle et ceux qui ont des interdépendances. Notamment, c'est le Framework Omega Library qui est utilisé pour identifier les récurrences critiques et pour parfaire les transformations qui élimineront ces récurrences.

— Perfectionnement de boucles
Le programme C peut contenir beaucoup de boucles imbriquées multidimensionnel et irrégulières. PICO analyse les taux de gains en débit en particulier qui pourront être atteint si certaines transformations sont appliquées pour contourner les irrégularités.

— Ordonnancement des itérations
En générale le résultat serait une boucle unidimensionnelle qui pourra être mappée à un processus qui devra assurer la cadence préalablement désignée.

— Allocation d'unités fonctionnelles
PICO alloue un ensemble optimal d'unités fonctionnelles pour le chemin

de données de chaque PA basé sur la combinaison exacte des opérations utilisées dans l'application, le débit souhaité et les fonctions disponibles comme macrocell dans la bibliothèque. En outre, des heuristiques supplémentaires sont utilisées pour partager davantage ces unités.

— Ordonnancement des opérations
Le compilateur PICO planifie et fait mapper les opérations de chaque boucle en appliquant le concept du modulo ajusté au débit souhaité (préalablement calculé). Les étages des pipelines sont insérés soigneusement en tenant en compte les caractéristiques temporelles des unités fonctionnelles issues de la bibliothèque choisis pour une technologie cible donnée.

— L'allocation des registres
PICO utilise un système de fichier de registres distribué personnalisé appelée ShiftQs pour s'accommoder avec les pipelines et le concept du modulo utilisé pour l'ordonnancement des opérations.

1.2.4 L'outil Gaut

La complexité croissante des applications de traitement du signal (DSP) et leurs exigences en débits grandissants nécessitent des implémentations matérielles efficaces. En effet, pour les applications DSP, les solutions logicielles pures basées sur les architectures multiprocesseurs ne sont pas vraiment approuvables. Alors que les accélérateurs matériels optimisés ou encore les coprocesseurs composé d'un ensemble de blocs de calcul communiquant par liens point à point sont aussi souvent souhaités.

Les SoC intégrés comme cœurs DSP auront besoin de nouveaux outils de conception permettant la synthèse à un niveau d'abstraction assez élevé (le niveau algorithmique). Les concepteurs de CI auront ainsi l'occasion de se concentrer sur la fonctionnalité et les performances cibles plutôt que sur le débogage RTL. Les concepteurs pourront se permettre de passer plus de temps à explorer l'espace de conception avec plusieurs scénarios "Et si ". Ils obtiendront une gamme d'alternatives et de choix où ils auront l'opportunité de sélectionner l'architecture offrant le meilleur compromis de puissance/vitesse/espace.

GAUT [Martin 1993, Coussy 2011] est un outil académique et open-source de synthèse haut niveau HLS dédié aux applications de traitement du signal. À partir d'une spécification algorithmique précise donnée en C/C++, GAUT extrait le parallélisme avant de procéder aux taches d'allocation, et d'ordonnancement. Dans Gaut, il est recommandé de spécifier le débit et la période d'horloge comme principales contraintes de synthèse. GAUT génère ensuite une architecture potentiellement pipelinée composée d'unités de traitement,

d'unités de mémoire et des unités de communication utilisant les interfaces GALS/LIS (Globally Asynchronous Locally Synchronous/Latency Insensitive System).

1.2.4.1 L'environnement de développement

La synthèse de haut niveau permet la recherche (semi) automatique des solutions architecturales qui respectent les contraintes spécifiées tout en optimisant les objectifs de conception. Pour être efficace, la synthèse doit s'appuyer sur une méthode de conception qui tient en compte les spécificités des champs d'applications. Du moment où GAUT est à l'origine créé pour cibler particulièrement les applications de traitement du signal en temps réel, Il a été optimisé pour servir les applications où l'on a de fortes intensités de calculs dominantes et des flux de données réguliers et périodiques.

L'algorithme à synthétiser dans GAUT est fourni en entrée en une description C. Le débit (spécifié par un intervalle initial représentant l'écart des débuts des itérations successives) et la période de l'horloge sont des contraintes de synthèse qui doivent être obligatoirement désignés. Le mappage de la mémoire et les diagrammes de temporisation des E/S sont des contraintes de synthèse facultatives. L'architecture matériels que GAUT génère est composée de trois unités fonctionnelles principales : une unité de traitement PU, une unité mémoire MEMU et une unité de communication et d'interface.

PU est un chemin de données data-path composé d'opérateurs logiques et arithmétiques, d'éléments de stockage, et un contrôleur (FSM) pour la logique d'aiguillage. Les éléments de stockage de l'unité centrale peuvent être des mémoires aux sémantiques (FIFO, LIFO) et/ou registres. La MEMU est composée de bancs de mémoires et des contrôleurs associés. Le COMU gère la synchronisation du processeur et des d'opérations mémoire via l'interface (GALS/LIS).

Comme décrit dans la figure 1.10, GAUT synthétise d'abord le Processing Unit PU. Ensuite, il génère l'unité de mémoire et l'unité de communication. Pendant la synthèse de l'unité centrale, GAUT sélectionne d'abord les opérateurs arithmétiques, ensuite il cherche les meilleures mappages qui permettront de satisfaire les contraintes et les objectifs initiaux de synthèse. L'optimisation des registres et de la mémoire, est basée sur des techniques de prédiction. Les voies de communication sont aussi optimisées.

Pour valider l'architecture générée, un test-bench est généré automatiquement pour permettre l'analyse des résultats obtenus par application de stimulus. Le stimulus peut être incrémental, aléatoire ou défini par l'utilisateur rendant ainsi possible une comparaison avec les spécifications algorithmiques initiales. La vérification peut être appliquée séparément et individuellement à l'unité de traitement. Dans ce cas, la mémoire et les unités de communication sont générées en tant que composants VHDL dont le comportement est

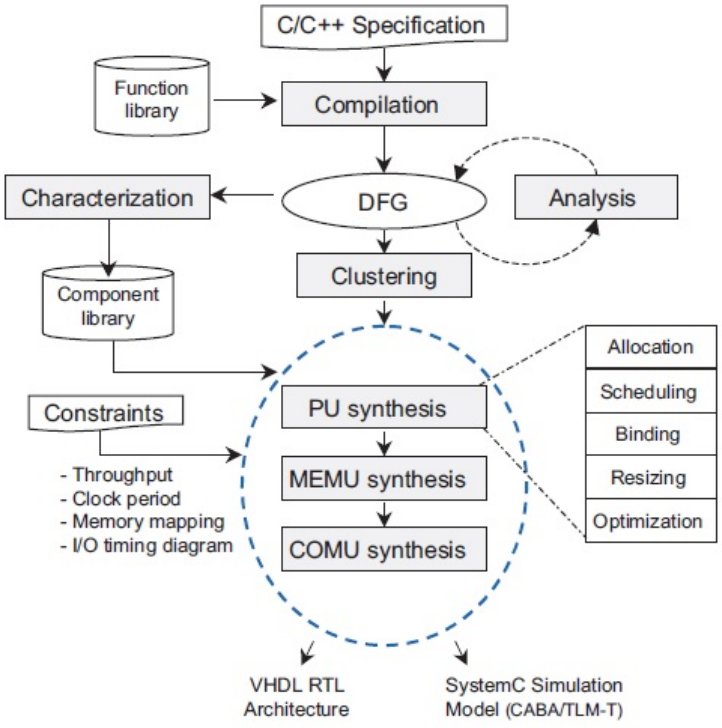

FIGURE 1.10 – Flot de synthèse dans Gaut [Coussy 2010]

décrit comme une machine à états finis avec un chemin de données. GAUT génère non seulement des modèles VHDL mais aussi des scripts nécessaires pour compiler et simuler le design avec le simulateur Modelsim. Il peut également comparer les résultats de deux simulations (produites par différents comportements temporels (E/S, pipeline ...)).

1.2.4.2 Le flot de synthèse

La description est fournie en entrée sous forme de fonction C/C++ qui est issue d'une classe appropriée dans la bibliothèque de Mentor Graphics. Le rôle du compilateur est de transformer la spécification initiale C/C++ en une représentation qui exprime les dépendances de données entre les opérations. Dans le Framework GAUT est engendrée l'utilisation des compilateurs gcc/g++ 4.2 pour extraire et donner une représentation en un graphe de flux de données (DFG).

Le compilateur gcc/g++ comprend trois composantes principales. Une partie frontale qui effectue l'analyse lexicale, syntaxique et sémantique du code. Une partie centrale nommée "GIMPLE" où sont opérées les optimisations de code. Une troisième partie finale où sont aussi menées les optimisations qui concernent spécifiquement le matériel. Le fichier source en entrée subira principalement quatre traitements.

Un prétraitement pour les directives préprocesseur. La construction de l'arbre abstraite de syntaxe (AST) pour chaque fonction du fichier source. L'arbre AST (Abstract Syntax Tree) est ensuite converti en un Control Data Flow Graph (CDFG) unifié appelé GENERIC qui n'est pas en fait encore véritablement approprié pour une optimisation. La représentation GENERIQUE est ensuite tassée en un sous-ensemble appelé forme GIMPLE ; les fausses dépendances de données sont éliminées avec la technique d'assignation de signal statique (SSA pour Static Signal Assignement) et aussi divers techniques d'optimisations scalaires (élimination du code mort, propagation de plage de valeurs, élimination de la redondance). Les optimisations de boucle (fusion de boucle, déroulement partiel de boucle, invariant de boucle, pelage de boucle) sont aussi appliquées. La forme GIMPLE est finalement traduite en une représentation interne propre à GAUT.

Chapitre 2

Les optimisations dans les outils HLS

Sommaire

Préambule

Les techniques d'optimisations constituent un aspect important qui devrait revêtir une attention particulière lors du développement d'un outil HLS. Ces techniques contribuent de façon assez nette à l'amélioration de performance de l'outil cible. Les taux de performances achevées en ce qui concerne la latence, le débit du système généré, l'occupation en espace et la puissance sont fermement liés au choix des optimisations à appliquer et surtout au degré de leurs sophistications. Dans ce chapitre on s'intéressera surtout aux optimisations liées à l'exploitation du parallélisme intrinsèque dans les algorithmes.

En principe, ces dernières sont capables d'offrir des doublements en performances pour la latence et le débit. Le HLS doit prendre en considération tout l'ensemble des autres contraintes, celles qui concernent par exemple la puissance et l'occupation en espace. Dans certains cas le HLS est amené à chercher les bons compromis qui d'une part répondrons aux exigences temporelles des applications et d'autre part satisferons au reste des contraintes (espace et puissance). Par conséquent, les optimisations prévues pour un HLS auront à être aussi liés aux techniques issues de ce qui est dit le Design Space Exploration (DSE).

2.1 Les optimisations dans un ordre général et vue d'ensemble

2.1.1 Le chainage des opérations

Le principe du chaînage d'opérations consiste à contraindre un ensemble d'opérations à être exécuté juste en un seul cycle horloge. Il est possible de l'appliquer par exemple dans le cas où l'on a deux opérations qui sont dépendantes l'une de l'autre et ils peuvent tous deux terminer l'exécution dans un temps inférieur au cycle horloge suggéré. il s'agit ici d'une optimisation qui s'opère au niveau matériel.

2.1.2 Otimisation de la longueur de représentation des données

L'optimisation de la longueur de représentation implique l'utilisation de transformations qui mèneront à réduire le nombre de bits requis par les opérateurs dans le chemin de données. Différemment aux compilateurs classiques ou conventionnels qui sont conçus pour les processeurs à usage général où les chemins de données sont de taille fixe (16, 32 ou 64 bits), un compilateur matériel peut exploiter la spécialisation pour générer des opérateurs (incluant des unités fonctionnelles et des registres) de taille personnalisée. En conséquence, nous pourrons être en mesure de sélectionner le minimum nombre de bits requis pour les opérations et/ou le stockage dans l'ensemble d'un algorithme spécifique donnée. Ceci conduira en principe à des minimisations en espace, en puissance et va rendre les chemins de données critiques moins encombrants. Cependant, cette optimisation ne peut pas être appliquée facilement. Le compilateur doit déterminer si des algorithmes spécifiques avec l'ensemble de données qui s'y attachent conviendront bien à une telle optimisation.

2.1.3 L'allocation mémoire

Les FPGA contiennent plusieurs bancs mémoire sous forme de BRAM (bloques RAM). Le concepteur pourra bien partitionner et mapper les structures de données logicielles sur ces BRAM. Ceci assurera une grande flexibilité pour les mouvements des données et les accès pourront se faire de façon assez rapide. On sera même en mesure d'établir plusieurs accès et des opérations de mémoire simultanés dans juste un seul cycle horloge lorsqu'on arrive à déterminer statiquement les opérations qui pourront s'exécuter sans contentions. L'utilisation de plusieurs BRAM fournit un support approprié pour l'implémentation du parallélisme.

2.1.4 Les optimisations au niveau des boucles

L'accélération matérielle est particulièrement importante lorsqu'il s'agit d'algorithmes qui contiennent des boucles à calculs intensifs. Le pipelining de boucle est une variante d'optimisation fréquemment adoptée. On exploite le parallélisme au niveau de la boucle en agençant en un pipeline l'exécution des itérations. En pipelining de boucle, l'intervalle d'initiation (II) est une mesure qui désigne le nombre de cycles d'horloge entre les itérations de boucle successives. Pour les débits élevés, il est souhaitable que l'intervalle d'initiation soit aussi petit que possible. Dans l'idéal il devrait valoir un, ce qui revient à faire démarrer une nouvelle itération à chaque nouveau cycle horloge.

La minimisation de l'intervalle d'initiation est obturée par deux facteurs. Le premier est dû aux contraintes de ressources alors que le deuxième est lié aux dépendances intra-boucles. La contrainte de ressources peut être illustrée en parti à travers le scénario suivant. Considérant un corps de boucle qui comprend trois opérations de chargement et une opération de rangement. Dans ce cas, l'obtention d'un intervalle d'initiation moins de deux est impossible si la mémoire a deux ports, puisque chaque itération de boucle effectue quatre opérations mémoire.

Pour cette raison, les optimisations de boucles sont fréquemment combinées avec l'architecture multi-bancs pour exploiter pleinement le parallélisme. Les dépendances intra-boucles ne permettent pas de démarrer une itération que si tous les résultats dont elle dépend dans les itérations précédentes sont déjà complètement achevées.

En raison de la grande disponibilité des ressources dans les FPGA, les difficultés liées à la question de ressources et la question de dépendances intra-boucles pourront être considérablement atténués dans le pipelining de boucle. Certains outils HLS commencent par fixer l'intervalle d'initiation à une valeur souhaitée, qui pourra permettre l'obtention des débits requis, puis ils amplifient le nombre de ressources continuellement jusqu'à ce qu'à l'acquisition de l'intervalle d'initiation voulu. Pour les boucles imbriquées sont employés

les transformations et les optimisations polyhedrales qui seront discutées à la section 2.2.

2.1.5 La librairie de ressources

Dans le processus de génération HLS, pour plus d'efficacité et pour répondre aux exigences de latence et débit tout en minimisant le nombre de ressources, il est essentiel de déterminer comment implémenter chaque opération. Tout d'abord est inspectée la spécification comportementale fournie en entrée pour identifier les caractéristiques des opérations, en l'occurrence, le type de chaque opération (arithmétique ou non arithmétiques), le types des opérandes (entiers, flottants ou doubles) et la longueur de représentation.

À ce stade, certaines opérations peuvent bénéficier d'optimisations spécifiques. Par exemple, les multiplications ou les divisions par une constante sont généralement transformées en opérations qui utilisent seulement des décalages et des additions pour minimiser l'occupation en espace et raccourcir les temps d'exécution. Toutes ces caractéristiques sont ensuite utilisées lors de la phase de l'allocation du module, où les opérations résultantes sont associées à des unités fonctionnelles contenues dans la bibliothèque de ressources.

La bibliothèque d'unités fonctionnelles peut être assez riche et peut contenir plusieurs implémentations pour chaque tache unique. D'une part, la bibliothèque comprend généralement des ressources qui sont spécifiques à différentes technologies (par exemple, plusieurs fournisseurs de FPGA). Le module d'allocation va exploiter les ressources qui ont été explicitement adaptés et optimisés pour la cible spécifique. La bibliothèque peut également contenir des ressources exprimées en tant que modèles données en tant que HDL standard (Verilog ou VHDL). Ces modèles peuvent être réadaptés et personnalisés en fonction des caractéristiques de la technologie cible. Dans ce cas, l'outil de synthèse sous-jacent peut déterminer le meilleur choix pour implémenter chaque fonction. Par exemple, les multiplicateurs peuvent être mappés soit sur des blocs DSP dédiés ou bien avec des tables (LUT).

Pour effectuer des optimisations agressives, chaque composant de la bibliothèque doit être annoté avec des informations utiles pendant tout le processus de synthèse, tel que l'occupation des ressources et la latence pour l'exécution des opérations. Il y a plusieurs approches pour la caractérisation des ressources de la bibliothèque. La première approche repose sur une synthèse logique rapide au cours de l'ordonnancement et la liaison des opérations pour déterminer l'unité la plus appropriée. On peut aisément remarquer que cette approche est assez coûteuse en termes de temps de calcul, surtout si la synthèse est effectuée à plusieurs reprises pour une même cible. Une approche alternative consiste à pré-caractériser toutes les ressources à l'avance. L'estimation de la performance est obtenue en suggérant un modèle paramétrique générique de l'unité fonctionnelle. Les paramètres sont les longueurs de représentation en bits et

le nombre d'étages du pipeline. La latence et l'occupation des ressources sont estimés en synthétisant chaque configuration puis les résultats sont finalement stockés dans la bibliothèque. Des modèles mathématiques peuvent être développés pour l'analyse des paramètres. Ces informations peuvent également être couplées avec les retards et les overheads engendrés durant la phase place-et-route pour rendre les résultats plus authentiques.

2.1.6 Spéculation et mouvement de code

La plupart des techniques d'ordonnancement HLS peuvent extraire le parallélisme uniquement dans la même région de contrôle (c'est-à-dire le même bloc CDFG de base). Cela peut limiter les performances de l'accélérateur résultant, en particulier dans les conceptions à forte intensité de contrôle. La spéculation est une technique de mouvement de code qui déplace les opérations le long des traces d'exécution, pour les anticiper par exemple avant les constructions conditionnelles. Les compilateurs pour le logiciel sont moins susceptibles d'utiliser cette technique. Dans le cas d'une compilation pour synthèse matériel, les opérations spéculées peuvent souvent être exécutées en parallèle avec le reste des opérations. Leurs résultats seront simplement conservés ou rejetés plus tard dépendamment des résultats calculés pour les branches.

2.1.7 Exploitation du parallélisme spatial

Les outils HLS s'attachent aussi à l'extraction du parallélisme au niveau des instructions en analysant les dépendances de données et le parallélisme au niveau des boucles via le pipelining de boucle. Cependant, Il est souvent difficile d'extraire automatiquement de grandes masses de parallélisme et les défis sont justement semblables à ceux rencontrés dans les compilateurs logiciels pour l'auto-parallélisation. Une alternative abondamment adoptée dans beaucoup d'outils HLS, consiste à synthétiser un accélérateur et écrire ensuite manuellement le code RTL qui invoque plusieurs instances du noyau synthétisé, et faire diriger les données depuis et vers les entrées/sorties de chacun. Cependant, cette approche est sujette aux erreurs et nécessite aussi une expertise en matériel. Une approche alternative qui semblerait mieux séduisante consiste à soutenir les paradigmes de parallélisation dans les logiciels de synthèse.

LegUp (voir section 3.2.1) est un outil HLS qui s'attache aussi au support des standards pthreads et OpenMP. Ces deux derniers sont largement utilisés pour le parallélisme des applications par les ingénieurs du logiciel. L'idée générale adoptée dans cet outil est de synthétiser les threads logiciels parallèles en un nombre égal d'unités matérielles fonctionnant en parallèle. Avec pthreads, un utilisateur peut exprimer à la fois le parallélisme spatial des tâches et celui

des données. Dans le premier cas, chaque unité matérielle peut effectuer une fonction différente, et dans le deuxième cas, plusieurs unités matérielles auront à effectuer la même fonction sur différentes parties d'un ensemble de données d'entrée. LegUp s'est attaché à prendre en charge également la synthèse de deux constructions de synchronisation pour le standard pthreads qui sont les mutexes et les barrières. Pour OpenMP, les auteurs se sont concentrés surtout sur l'aspect de la norme qui cible en particulier la parallélisme des boucles. Dans une boucle, si les itérations sont indépendantes alors elles pourront être affectées à des unités fonctionnelles matérielles qui seront exécutés en parallèle. LegUp s'attache à fournir un support pour le parallélisme imbriqué à travers ce qui est dit « threads forking threads ». Ici, il s'agit de threads qui eux même vont créer d'autres threads ou vont renfermer des constructions de parallélisation OpenMP.

2.2 La parallélisation de boucles dans les compilateurs dédiés au calcul haute performance

Dans cette section sont examinées les techniques de parallélisation de boucles envisagées pour le calcul haute performance (HPC, High Performance Computing). Elles sont employées en particulier dans les compilateurs "logiciels" (non orientés HLS) source-à-source dédiés aux applications à calculs intensifs. Elles peuvent être appliquées sans aucune connaissance et de façon indépendante de l'architecture cible.

Dans ces compilateurs, on s'intéresse à la détection du parallélisme au niveau des boucles (par transformation des boucles DO en boucles DOALL) et on cherche à discriminer les dépendances qui sont à l'origine du caractère séquentiel du code.

Les algorithmes de parallélisation de boucle emploient différentes techniques mathématiques. Ils intègrent des algorithmes de graphes, des calculs matriciels ou encore la programmation linéaire. Ils adoptent des représentations différentes pour décrire les dépendances. Il y a ceux qui utilisent une description par graphes à niveaux, graphes à vecteurs de direction ou aussi les polyèdres et les expressions affines.

2.2.1 Graph de dépendance

2.2.1.1 Le graph EDG (Expanded Dependence Graph)

Dans un ensemble de boucles imbriquées, les dépendances entre les opérations sont représentées par un graphe acyclique orienté appelé graphe de dépendance étendu ou (EDG). Pour le code exemple donné ci-dessous (Code 2.1), la représentation EDG correspondante est donnée à la figure 2.1.

L'EDG est généralement trop grand et n'est pas toujours complètement fiable pour décrire de façon exacte les dépendances au moment de la compilation. Le problème du volume est comblé par une autre variante de graphe qui est le Reduced Dependence Graph (RDG).

Code 2.1

```
for i = 1 to n
    for j = 1 to n
        A(i, j) = A(i, j − 1) + A(i − 1, j − 1)
    endfor
endfor
```

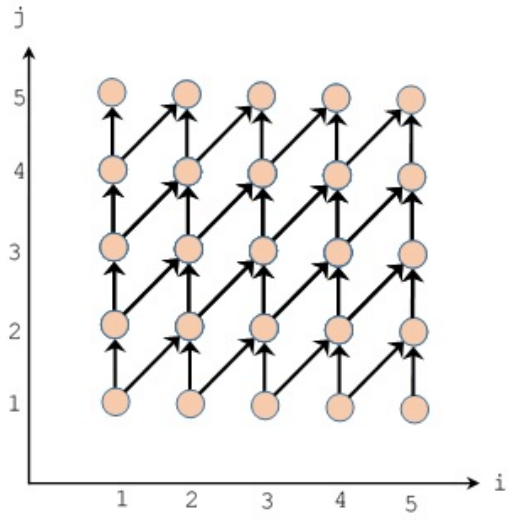

FIGURE 2.1 – EDG correspondant au code 2.1

2.2.1.2 Le graph de dépendance réduit RDG

Le Reduced Dependence Graph (RDG) attribue un sommet pour chaque instruction dans le nid de boucles mais peut utiliser différents étiquetages pour les arcs. Certaines variantes de RDG utilisent les niveaux de dépendances pour l'étiquetage, alors que d'autres variantes utilisent les vecteurs de direction. Toujours pour le code 2.1, les deux variantes de RDG correspondantes sont données à la figure 2.2. Il est à noter qu'il est possible qu'un RDG pourrait avoir plus que juste un seul EDG équivalent, ce qui engendre des problèmes d'ambigüité pour le compilateur.

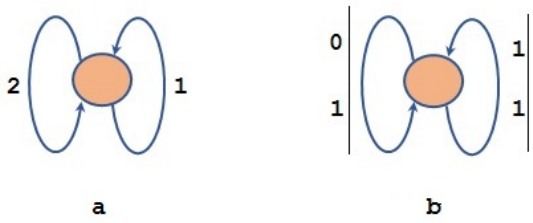

FIGURE 2.2 – RDG correspondant au code de l'exemple 2.1 - (a) Étiquetage par niveaux de dépendance - (b) Étiquetage par vecteurs de direction

2.2.2 Les transformations

Il y a au moins trois façons de définir un nouvel ordre pour les opérations d'un nid de boucle donné.

— Utiliser des transformations élémentaires comme étape de base. Utiliser par exemple les distributions de boucle comme c'est réalisé dans l'algorithme d'Allen et Kennedy (voir section 2.3.1), l'échange de boucles et l'inclinaison des boucles.

— Appliquez un changement linéaire de base sur le domaine d'itération, c'est-à-dire appliquez une transformation unimodulaire sur les vecteurs d'itération. C'est l'approche adoptée par Wolf et Lam (décrite à la section 2.3.2)

— Définir une planification pour un domaine d-dimensionnel, c'est-à-dire appliquez une transformation affine à partir de Z^n à Z^d et interpréter la transformation comme une fonction multidimensionnel. Chaque composant correspondra à une boucle séquentielle.

2.2.3 L'abstraction de dépendance

Dans un nid de boucle d'ordre 'n', 'n' est appelé la profondeur (depth) du nid. Les vecteurs de Z^n sont dit vecteurs d'itérations qui son contenu dans un polyèdre convexe fini appelé le domaine d'itération. la i^{me} composante d'un vecteur d'itération est la valeur du i^{me} compteur de boucle dans le nid de boucle en comptant à partir de la boucle la plus externe et les itérations sont donc exécutées dans l'ordre lexicographique de leur vecteurs d'itération.

On désigne par D le domaine d'itération polyédrique, par I et J les vecteurs d'itération n-dimensionnels dans D, et par S_i la i^{me} instruction dans le nid de boucle, où $1 \leq i \leq s$. on écrit $I >_l J$ si I est lexico-graphiquement plus grand que J

2.2.4 Les distances dans les graphs de dépendances

Deux opérations sont en dépendance ou l'une est dépendante de l'autre si les deux opérations accèdent au même emplacement de mémoire et si au moins l'un des accès est une écriture. La dépendance est dirigée selon l'ordre séquentiel de l'exécution c.à.d. de la première opération exécutée envers la dernière, que l'accès est ici pour une écriture ou bien pour une lecture.

On écrit $S_i(I) \Longrightarrow S_j(J)$ si l'instruction S_j à l'itération J dépend de l'instruction S_i à itération I. L'ordre partiel défini par $\Longrightarrow$ définit le graphe de dépendance étendu EDG. Il est à noter aussi que $(J - I)$ est toujours lexico-graphiquement non négatif lorsque $S_i(I) \Longrightarrow S_j(J)$.

Le calcul d'un EDG au moment de la compilation n'est pas vraiment toujours faisable. Un manque d'informations concernant certains paramètres comme la taille des variables ou si les accès mémoires ne sont pas désignés complètement de façon exacte et précise, tout cela empêchera de construire un EDG valide. Si la taille du domaine est assez grande ou complexe, les temps de calculs pour la génération d'un EDG pourront s'allonger de façon fatale. Les dépendances sont aussi capturées utilisant des graphes orientés plus réduits compacts dits RDG (Figure 2.2).

Dans le RDG, il y a dépendance entre deux instructions S_i et S_j (on écrit l'arc (Edge) $e : S_i \longrightarrow S_j$) s'il existe au moins une paire (I, J) telle que $S_i(I) \Longrightarrow S_j(J)$. De plus, l'arc e dirigé de S_i à S_j dans le RDG est étiqueté par l'ensemble $\{(I, J) \in D^2 | S_i(I) \Longrightarrow S_j(J)\}$.

Pour une certaine classe de boucles imbriquées, il est possible d'exprimer exactement cet ensemble des paires (I, J). I est donnée comme une fonction affine $f_{i,j}$ de J où J varie dans un polyèdre $P_{i,j}$ défini comme

$$\{(I, J) \in D^2 | S_i(I) \Longrightarrow S_j(J)\} = \{(f_{i,j}(J), J) | J \in P_{i,j} \subset D\} \tag{2.1}$$

Dans beaucoup d'algorithmes d'analyse de dépendance, au lieu de considérer l'ensemble des paires (I, J), on introduit plutôt l'ensemble $E_{i,j}$, pour toutes les valeurs possibles $(J - I)$. $E_{i,j}$ est appelé l'ensemble des vecteurs de distance, ou ensemble de distance :

$$E_{i,j} = \{(J - I) | S_i(I) \Longrightarrow S_j(J)\}$$

Lorsque l'analyse de dépendance exacte est réalisable, l'équation 2.1 montre que l'ensemble des vecteurs de distance est la projection des points entiers d'un polyèdre. Cet ensemble peut être approximé par sa coque convexe ou par une description plus ou moins précise d'un polyèdre plus grand (ou d'une union finie de polyèdres). La représentation par les vecteurs de distance n'est pas équivalente à la représentation par paires, car on perdra l'information qui concerne en particulier l'emplacement dans l'EDG d'un vecteur de distance donnée. Néanmoins, cette représentation reste toujours importante, surtout s'il parait que les temps de calculs lors de l'analyse de dépendance pourront être fatals. Les représentations de distances communément utilisés sont : Niveaux de dépendance, vecteurs de direction et les polyèdres de dépendance.

2.2.5 L'Algorithme d'Allen et Kennedy

L'algorithme d'Allen et Kennedy [Allen 1987] a été à l'origine conçu pour la vectorisation des boucles. Il a été étendu afin de maximiser le nombre de boucles parallèles et pour minimiser le nombre de synchronisations dans le code transformé. L'entrée de l'algorithme est un LRDG (Labeled RDG).

L'algorithme est basé sur les faits suivants :

1. Une boucle est parallèle s'il n'y a pas dépendance pour une instruction entourée par la boucle en question dont le niveau est égal à la profondeur de la boucle.

2. Toutes les itérations d'une instruction S_1 peuvent être effectuées avant toute itération d'une instruction S_2 s'il n'y a pas de dépendance dans le LRDG de S_2 vers S_1.

La propriété (1) permet de marquer une boucle ou bien comme une boucle DOALL ou une boucle DOSEQ, alors que la propriété (2) suggère que la détection du parallélisme peut être menée indépendamment dans chaque composant fortement couplé du LRDG. L'extraction du parallélisme est faite par l'effet de distribution de boucle.

2.2.5.1 L'algorithme

Pour un graphe de dépendance G, on note $G(k)$ le sous-graphe de G dans lequel toutes les dépendances au niveau strictement inférieur à k ont été supprimées.

— Si k $>$ n, s'arrêter.
— Décomposer G(k) en ses composants fortement couplés G_i et trier les topologiquement.

— Réécrire le code pour que chaque G_i appartienne à un nid de boucle différent (au niveau k) et l'ordre sur le G_i est préservé (distribution des boucles au niveau $\leq$ k).
— Pour chaque G_i, marquer la boucle au niveau k comme une boucle DOALL si G_i n'a pas d'arc au niveau k. Sinon marquez la boucle comme une boucle DOSEQ.
— Pour chaque G_i, appelez Allen-Kennedy (G_i, k + 1).

Exemple

Code 2.2

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
          S1 : A(i,j,k) = A(i-1,i+j,k) + A(i,j,k-1) + B(i,j-1,k)
          S2 : B(i,j,k) = B(i,j-1,k+j) + A(i-1,j,k)
        endfor
    endfor
endfor
```

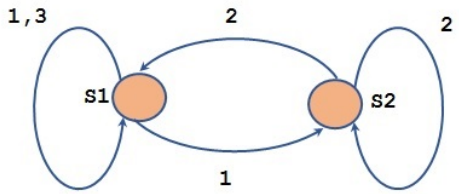

FIGURE 2.3 – RDG correspondant au code 2.2

Le graphe de dépendance G = G(1), dessiné sur la figure 2.3, n'a qu'un seul composant fortement connecté et au moins un arc au niveau 1. donc la boucle la plus externe est séquentielle. Au deuxième tour (l'arc au niveau 1 n'est plus considéré), G(2) est décomposé en deux composants fortement couplés : toutes les itérations de l'instruction S_2 peuvent être effectuées avant toutes les itérations de l'instruction S_1. Une distribution en boucle est effectuée. Le composant fortement connecté S_1, ne contient aucun arc de niveau 2 mais un

arc de niveau 3. Ainsi, la deuxième boucle entourant S_1 est marquée DOSEQ et la troisième est marquée un DOALL.

Le composant fortement connecté enfermant S_2 contient un arc de niveau 2 mais pas de bord au niveau 3. Ainsi la deuxième boucle entourant S1 est marquée DOALL alors que la troisième va être marquée DOSEQ. On obtient

Code 2.3

DOSEQ $i = 1$ *to* n
  DOSEQ $j = 1$ *to* n
    DOALL $k = 1$ *to* n
      $S_2 : B(i, j, k) = B(i, j - 1, k + j) + A(i - 1, j, k)$
    endDO
  endDO
  DOSALL $j = 1$ *to* n
  DOSEQ $k = 1$ *to* n
    $S_1 : A(i, j, k) = A(i - 1, i + j, k) + A(i, j, k - 1) + B(i, j - 1, k)$
  endDO
  endDO
endDO

2.2.5.2 Points forts

— L'algorithme est optimal par rapport à tous les algorithmes de parallélisations où les boucles et les instances de S au niveau du code transformé sont exactement les mêmes que ceux qui se trouvaient dans le code initial.
— L'algorithme Allen-Kennedy est optimal par rapport aux algorithmes de détection de parallélismes existant si l'entrée est un graphe de dépendance réduit avec étiquetage niveau (RLDG) [Darte 1996a].

2.2.6 L'algorithm de Wolf et Lam

Le code 2.4 et 2.5 contiennent du parallélisme qui ne peut pas être détecté par L'algorithme d'Allen et Kennedy. Le parallélisme dans ces deux cas ne peut pas être extrait si les dépendances sont exprimées par des étiquetage de niveaux.

Pour surmonter ces limitations, Wolf et Lam [Wolf 1991] ont proposé un algorithme qui utilise en entrée des graphes avec des vecteurs de direction. Leur travail unifie tous les précédents algorithmes basés sur des opérations matricielles élémentaires telles que l'inclinaison de boucle, l'échange et l'inversion de boucle en une unité de transformations unimodulaires.

Code 2.4

```
for i = 1 to n
    for j = 1 to n
        A(i, j) = A(i − 1, j − 1) + A(i, j − 1)
    endfor
endfor
```

Code 2.5

```
for i = 1 to n
    for j = 1 to n
        A(i, j) = A(i − 1, j) + A(i, j − 1)
    endfor
endfor
```

2.2.6.1 Objectif

Wolf et Lam s'attache à construire des ensembles de nids de boucles entièrement permutables. Les boucles pleinement permutables sont à la base de toutes les techniques de pavage [Boulet 1994, Wolf 1991]. Le pavage (tiling) est utilisé pour exposer le parallélisme à moyenne et large granularité. En outre, un ensemble de boucles « d » entièrement permutables peuvent être réécrites en une seule boucle séquentielle et « d - 1 » boucles parallèles.

L'algorithme de Wolf et Lam construit le plus grand ensemble externe de boucles permutables. Ensuite, il examine récursivement les dimensions restantes et les dépendances non satisfaites par ces boucles.

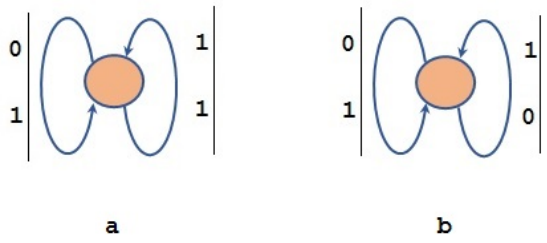

FIGURE 2.4 – RDGs : a. RDG pour le code 2.4 - b. RDG pour le code 2.5

2.2.6.2 Interprétations théoriques

Les transformations uni-modulaires vont satisfaire essentiellement deux critères : la linéarité et l'inversion. La linéarité permet de facilement vérifier si T est une transformation valide. En effet, T est valide si et seulement si $T_d >_l 0$ pour tous les vecteurs de distance non nuls d. L'inversion permet une réécriture telle que la transformation va être opérée par un simple changement de base dans Z^n.

On Note $t(1), ..., t(n)$ les rangées de T. et Soit Γ la fermeture du cône généré par tous les vecteurs de direction. Pour un vecteur de direction d :

$$T_d >_l 0 \Leftrightarrow \exists k_d, 1 \leq k_d \leq n \setminus \forall i, 1 \leq i < k_d, t(i).d = 0 \; and \; t(k_d).d > 0$$

Cela signifie que les dépendances représentées par d sont entraînées au niveau de boucle k_d. Si $k_d = 1$ pour tous les vecteurs de direction d, alors toutes les dépendances si elles sont entrainées à la première boucle, toutes les boucles internes seront des boucles DOALL. $t(1)$ est alors appelé un hyperplan de séparation. Un tel hyperplan de séparation existe si et seulement si Γ est pointé, c'est-à-dire si et seulement si Γ ne contient pas d'espace linéaire.

Ceci est également équivalent au fait que le cône Γ^+ (défini par $\Gamma^+ = \{y \mid \forall x \in \Gamma, y.x \geq 0\}$) est en pleine dimension. Construire T à partir de n vecteurs linéairement indépendants de Γ^+ permet de transformer les boucles en n boucles entièrement permutables.

Lorsque le cône Γ n'est pas pointé, Γ^+ a une dimension r, $1 < r < n$, $r = n - s$ où s est la dimension de l'espace de linéarité de Γ. Avec r vecteurs linéairement indépendants de Γ^+, on peut transformer le nid de la boucle pour que les r boucles les plus externes deviennent entièrement permutables. Ensuite, on peut appliquer récursivement la même technique pour transformer les $n - r$ boucles intérieures, en considérant les vecteurs de direction qui ne sont pas déjà entrainés par une boucle parmi les r boucles externes, c'est-à-dire en considérant les vecteurs de direction inclus dans l'espace de linéarité de Γ.

2.2.6.3 L'algorithm

Dans l'algorithme Wolf-Lam, l'entrée est un ensemble de vecteurs de direction D et une séquence de vecteurs linéairement indépendants E (initialisée à vide) qui donnerons lieu à l'élaboration de la matrice de transformation.

— Définir Γ comme fermeture du cône généré par les vecteurs de direction de D.

— Définir $\Gamma^+ = \{y \mid \forall x \in \Gamma,\ y.x \geq 0\}$ et soit r la dimension de Γ^+.
— Compléter E dans un sous-ensemble E' de r vecteurs linéairement indépendants de Γ^+ (par construction, $E \subset \Gamma^+$).
— Soit D' le sous-ensemble de D défini par
$d \in D' \Leftrightarrow \forall\ v \in E,\ v.d = 0$ c.à.d $(D' = D \cap E'^{\perp} = D \cap lin.space(\Gamma))$.
— Appelez Wolf-Lam(D', E').

La matrice unimodulaire T souhaitée peut être réalisée comme suit :

— Pour l'ensemble D des vecteurs de direction. Mettre $E = \emptyset$ et appeler Wolf-Lam (D, E).
— Construire une matrice T_1 non singulière dont les premières lignes sont les vecteurs de E (avec le même ordre). Soit $T_2 = p.T_1^{-1}$ où p est choisi pour que T_2 soit une matrice intégrale.
— Calculer la forme Hermitinne gauche de $T^2, T^2 = QH$, où H est non négatif, triangulaire inférieur et Q est unimodulaire.
— Q^{-1} est la matrice de transformation désirée (puisque $pQ^{-1}D = HT_1D$).

Exemple

Considérons le nid de boucle défini dans le code 2.6 et son graph RDG donnée par la figure 2.5

Code 2.6

```
for i = 1 to n
    for j = 1 to n
      for k = 1 to n
        A(i,j,k) = A(i-1,j+i,k) + A(i,j,k-1) + A(i,j-1,k+1)
      endfor
    endfor
endfor
```

L'ensemble des vecteurs de direction est $D = \{(1,-,0),(0,0,1),(0,1,-1)\}$. L'espace de linéarité de $\Gamma(D)$ est bidimensionnel (généré par $(0,1,0)$ *et* $(0,0,1)$). Ainsi, $\Gamma^+(D)$ est unidimensionnel et généré par $E1 = (1,0,0)$. Alors $D' = \{(0,0,1),(0,1,-1)\}$ et $\Gamma(D')$ sont pointés. Nous complétons $E1$ par deux vecteurs de $\Gamma^+(D')$, par exemple par $E2 = \{(0,1,0),(0,1,1)\}$.

Dans cet exemple particulier, la matrice de transformation dont les lignes sont $E1, E2$ est déjà unimodulaire et correspond à une simple déviation de boucle. Pour exposer les DOALL boucles, nous choisissons le premier vecteur

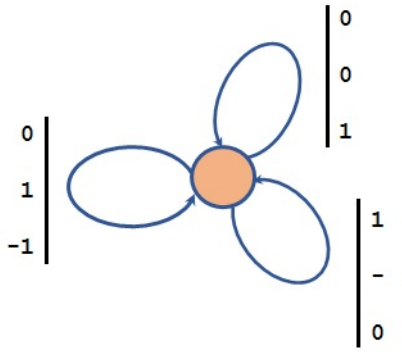

FIGURE 2.5 – RDGs : RDG pour le code 2.6

de $E2$ dans l'intérieur relatif de Γ^+, par exemple $E2 = \{(0,2,1),(0,1,0)\}$. En termes de transformations de boucles, cela revient à dévier la boucle k par le facteur 2 puis à intervertir les boucles j et k. On obtient

Code 2.7

$DoSEQ\ i = 1\ to\ n$
  $DoSEQ\ k = 3\ to\ 3 \times n$
    $DoALL\ j = Max(1, \lceil \frac{k-n}{2} \rceil)\ to\ min(n, \lfloor \frac{k-1}{2} \rfloor)$
      $A(i, j, k - 2 \times j) = A(i-1, j+i, k-2 \times j) + A(i, j, k-2 \times j-1)$
      $+ A(i, j-1, k-2 \times j+1)$
    $endDo$
  $endDo$
$endDo$

2.2.7 Algorithm de Darte et Vivien

Dans cet algorithme [Darte 1994, Darte 1996b] l'entrée est un graphe réduit polyédrique de dépendance. Avec les deux algorithmes précédent on peut se trouvé dans des situations où l'on a une portion de parallélisme qui est détectable par un algorithme, au moment où la sortie reste toujours séquentielle utilisant le deuxième algorithme et vice versa. Cela motive la recherche d'un nouveau algorithme qui dans l'éventuel pourra produire les résultats des deux algorithmes tous deux à la fois. Si les deux algorithmes sont combinés, il y a une possibilité d'exposer du parallélisme, mais les résultats ne seront pas aussi optimaux. Et comme c'est montré à travers l'exemple 2.2, les résultats sont presque juste similaires à ceux produits par l'algorithme de Allen-Kennedy.

Code 2.8

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
          S1 : A(i, j, k) = B(i − 1, i + j, k) + B(i, j − 1, k + 2)
          S2 : B(i, j, k) = A(i, j − 1, k + j) + A(i, j, k − 1)
          endfor
    endfor
endfor
```

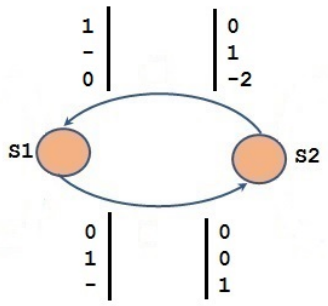

FIGURE 2.6 – RDGs : RDG pour le code 2.8

Il faut remarquer aussi que les deux algorithmes utilisent des représentations différentes pour les entrées. Dans l'algorithme de Darte et Vivien on essaye de combiner les deux effets des deux algorithmes et on applique une variante de la transformation uni-modulaire combinée avec une distribution de boucles toutes deux à la fois et utilisant une nouvelle forme de représentation adaptée pour les entrées.

Le nid de boucle présenté dans l'exemple suivant contient une quantité de parallélisme qui ne peut pas être détecté si les dépendances sont représentées par un graphe de dépendance à vecteurs de directions ou vecteurs de niveaux. L'algorithme de Darte et Vivien utilise un graphe de dépendance polyédrique.

Avec l'algorithme d'Allen-Kennedy, on trouve dans le graphe (a) de la figure 2.7 des dépendances au niveau 1 et au niveau 2. Ainsi aucun parallélisme ne pourra être détecté. Avec l'algorithme Wolf-Lam, on ne peut trouver des boucles qui pourront être complètement permutables. Et donc, le parallélisme n'est pas encore une fois ici détectable.

Code 2.9

```
for i = 1 to n
    for j = 1 to n
      S : A(i, j) = A(j, i) + A(i, j − 1)
    endfor
endfor
```

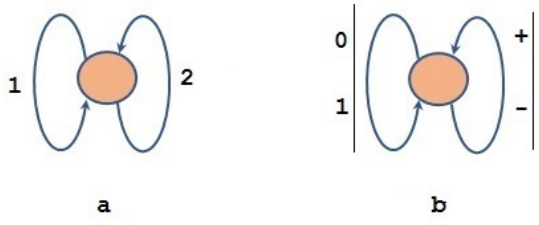

FIGURE 2.7 – RDGs : RDG pour le code 2.9

Dans l'exemple précédent, on peut remarquer qu'il y a une dépendance uniforme $u = (0, 1)$ et un ensemble de vecteurs de distance $\{(j - i, i - j) = (j - i)(1, -1)|1 \leq j - i \leq n - 1\}$ qui peut être approximé aussi par $P = \{(1, -1) + \lambda(1, -1)|\lambda \geq 0\}$. P est un polyèdre avec un sommet $v = (1, -1)$ et un rayon $r = (1, -1)$.

On cherche une transformation linéaire $T(i, j)$, tel que $X = (x1, x2)$ et $T(i, j) = x1.i + x2.j$. Pour que T soit valide nous cherchons$X$ tel que $X.d \geq 1$ pour tout vecteur de dépendance d.
Ainsi, $X(0, 1) \geq 1$ et $X.p \geq 1$ pour tout $p \in P$. Cette dernière inégalité est équivalente à :
$X(1, -1) + \lambda.X(1, -1) \geq 1$ *avec* $\lambda \geq 0$, ce qui équivaut à :
$X(1, -1) \geq 1$ *et* $X(1, -1) \geq 0$, c'est-à-dire $X.v \geq 1$ et $X.r \geq 0$. Par conséquent, il suffit de résoudre les trois inégalités suivantes :

$X.u \geq 1, \quad X.v \geq 1 \quad$ et $\quad X.r \geq 0$
exprimées aussi comme

$$X.\begin{pmatrix}1\\0\end{pmatrix} \geq 1, \quad X.\begin{pmatrix}1\\-1\end{pmatrix} \geq 1 \quad \text{et} \quad X.\begin{pmatrix}1\\-1\end{pmatrix} \geq 0$$

Ce qui donne $X = (2, 1)$. Avec une approximation des dépendances par des

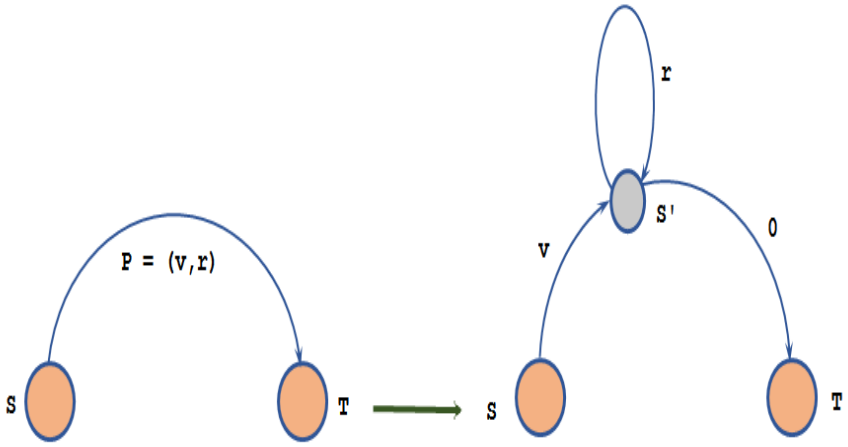

FIGURE 2.8 – RDGs : RDG pour le code 2.9

polyèdres, on retrouve le parallélisme non détecté par l'algorithme d'Allen-Kennedy et le parallélisme non détecté aussi par Wolf-Lam en résolvant juste un ensemble simple d'inégalités.

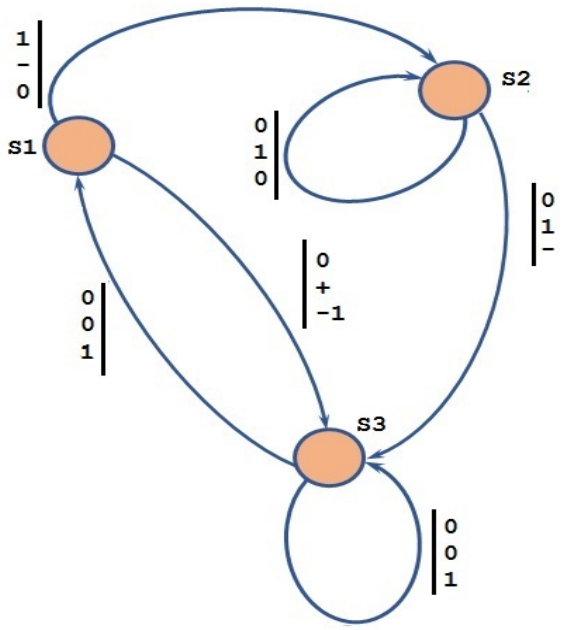

FIGURE 2.9 – RDGs : RDG pour le code 2.10

Ici on opère une « uniformisation » sur l'ensemble P qui permettra d'obtenir des inégalités uniformes sur v et r. Grâce à cette uniformisation, les contraintes affines disparaissent et nous n'avons pas à utiliser le lemme de Farkas comme c'est le cas avec l'algorithme de Feautrier [Feautrier 1992].

Code 2.10

$for\ i = 1\ to\ n$
  $for\ j = 1\ to\ n$
    $for\ k = 1\ to\ n$
      $S_1 : A(i,j,k) = C(i,j,k-1) + 1$
      $S_2 : B(i,j,k) = A(i-1,j+i,k) + B(i,j-1,k)$
      $S_3 : C(i,j,k+1) = C(i,j,k) + B(i,j-1,k+i) + A(i,j-k,k+1)$
      $endfor$
  $endfor$
$endfor$

Code 2.11

$DoSEQ\ i = 1\ to\ n$
  $DoSEQ\ j = 1\ to\ n$
    $DoPAR\ k = 1\ to\ j$
      $B(i,j,k) = A(i-1,j+i,k) + B(i,j-1,k)$
    $endDo$
  $endDo$
  $DoSEQ\ k = 1\ to\ n+1$
    $if(k \leq n)then$
    $DoPAR\ j = k$ to n
      $A(i,j,k) = C(i,j,k-1) + 1$
    $endDo$
    $endif$
    $if(k \geq 2)then$
    $DoPAR\ j = k-1$ to n
      $C(i,j,k) = C(i,j,k-1) + B(i,j-1,k+i-1) + A(i,j-k+1,k)$
    $endDo$
    $endif$
  $endDo$
$endDo$

Pour mieux comprendre le principe de "l'uniformisation", on a à considérer le chemin de dépendance. L'arc e, orienté de l'instruction S envers l'instruction T et marqué comme vecteur de distance $p = v + \lambda.r$ va représenter un chemin noté Φ qui utilise une seule fois le vecteur de dépendance "uniforme" v et λ fois le vecteur de dépendance "uniforme" r. Un nouveau nœud S' est

introduit et le chemin Φ est simulé de la façon décrite par la figure 2.8. On fait remarquer que l'on a un arc de poids nul qui est dirigé de S' vers le nœud initial T.

Le code 2.10 illustre un autre exemple où ni l'algorithme de d'Allen-Kennedy ni l'algorithme de Wolf-Lam ne pourront mener à une détection du paralllisme.

Dans le graphe réduit de dépendance donné à la figure 2.9, les arcs sont étiquetés par des vecteurs de direction. La troisième instruction semble être purement séquentielle si on applique l'algorithme de d'Allen-Kennedy. l'algorithme proposé ici permettra de construire la transformation multi-dimensionnel : $(2i+1, 2k)$ pour la première instruction, $(2i, j)$ pour la deuxième et $(2i+1, 2k+3)$ pour la troisième instruction.
Le code 2.11 est le résultat généré par cette transformation.

2.2.7.1 Uniformisation

Les PRDG (graphes polyédriques à dépendance réduite) peuvent être capturé dans des structures équivalentes plus simples à manipuler. Il s'agit de graphes de dépendance uniformes. Ce sont des graphes avec des arrêtes étiquetés par des vecteurs à dépendance à valeurs constantes.

Les sommets dans les graphes polyédriques sont appelés nœuds pour qu'ils ne soient pas confondus avec les sommets des graphes réduits dans l'algorithme d'uniformisation. Et le PRDG initial qui décrit les dépendances dans le code à paralléliser est appelé le graphe d'origine et est noté $G_o = (V, E)$. Le RDG uniforme équivalent à G_o est construit utilisant l'algorithme de translation et est noté $G_u = (W, F)$.

L'algorithme de translation construit G_u en scrutant tous les arrêtes de G_o. On commence en posant $G_u = (W, F) = (V, \phi)$, ensuite pour chaque arête e de E, on ajoute à G_u de nouveaux nœuds et de nouveaux arrêtes en fonction du polyèdre $P(e)$. On appelle nœuds virtuels les nœuds qui sont nouvellement créés, par opposition aux nœuds réels qui correspondent aux nœuds de G_o. Soit e un arrête de E. on dénote x_e et y_e respectivement les nœuds que e quitte et atteint : $x_e \longrightarrow^e y_e$. Cette définition est aussi généralisée et appliqués aux chemins. Nous suivons les notations suivantes : ω, ρ *et* λ désignent respectivement le nombre de sommets v_i, nombre de rayons r_i, et nombre de segments de droites l_i du polyèdre $P(e)$.

2.2.7.2 L'Algorithme de translation.

Soit $W = V$ et $F = \phi$
Pour $e : x_e \longrightarrow y_e \in E$ faire
- Ajouter àW un nouveau nœud virtuel n_e
- Ajouter à F ω arrêtes de poids $v_1, v_2, ..., v_?$ dirigé de x_e à n_e
- Ajouter à F ρ auto-boucles autour de n_e de poids $r_1, r_2, ..., r_\rho$
- Ajouter à F λ auto-boucles autour de n_e de poids $l_1, l_2, ..., l_\lambda$
- Ajouter à F λ auto-boucles autour de n_e de poids $-l_1, -l_2, ..., -l_\lambda$
- Ajouter à F un front de poids nul dirigé de n_e à y_e

On applique la translation en considérant comme entrée l'exemple précédent, et son PRDG illustré à la figure 2.10, on obtient le graphe de dépendance uniforme associé décrit par la figure 2.11. Il possède trois nouveaux nœuds marqués en gris (c'est-à-dire des nœuds virtuels) qui correspondent au symbole "+" et les deux symboles "-" dans les vecteurs de direction initiaux.

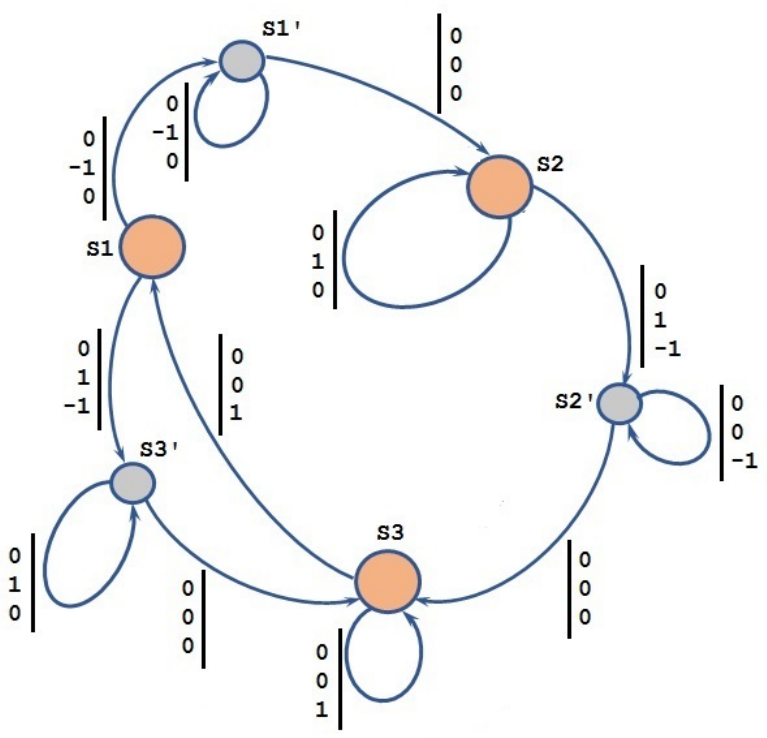

FIGURE 2.10 – RDGs : RDG pour le code 2.11

2.2.7.3 L'étape d'ordonnancement

On agit sur le graphe de dépendance translaté ou uniformisé G_u pour générer une transformée multidimensionnelle pour chaque nœud réel. G_u est supposé être fortement connecté.

A Chaque étape de la récurrence on construit un sous-graphe G' particulier du graphe G en cours de traitement. Une fois G' construit, un ensemble de de contraintes linéaires sont définit et une transformation valide qui satisfait à toutes les dépendances des arêtes non appartenant à G' pourra bien être calculée.

G' est défini comme un sous-graphe de G généré par tous les arêtes de G qui appartiennent à au moins un multi-cycle de poids nul. Un multi-cycle est une union de cycles, pas nécessairement connecté, et le poids d'une union de cycles est la somme des poids de ses cycles constitutifs. G est construit par la résolution d'un programme linéaire. L'étape d'ordonnancement peut être résumée par l'algorithme récursif donné ci-dessous.
L'appel initial est Darte-Vivien $(G_u, 1)$. L'algorithme construit, pour chaque nœud réel S de G_u, une séquence de vecteurs $X_S^1, ..., X_S^{ds}$ et une séquence des constantes $\rho_S^1, ..., \rho_S^{dS}$ qui définit une transformation multidimensionnel valide.

DARTE-VIVIEN (G, k)

1. Construire G' le sous-graphe de G généré par tous les arêtes appartenant au moins à un multi-cycle à poids nul de G.
2. Ajouter en G' tous les arêtes de x_e à y_e et toutes les auto-boucles sur y_e si $e = (x_e, y_e)$ est déjà un arête de G' allant d'un nœud réel x_e à un nœud virtuel y_e.
3. Sélectionnez un vecteur X et une constante ρ_S pour chaque nœud S dans G, tel que :
   $e = \{x_e, y_e\} \in G'$ *or* x_e *is a virtual node* $\Longrightarrow Xw(e) + \rho y_e - \rho x_e \geq 0$
   $e = \{x_e, y_e\} \notin G'$ *or* x_e *is a virtual node* $\Longrightarrow Xw(e) + \rho y_e - \rho x_e \geq 1$
   Pour tous les nœuds réels S de G, soit $\rho_S^k = \rho S$ et $X_S^k = X$.
4. Si G' est vide ou n'a que des nœuds virtuels, alors faire return.
5. Si G' est fortement connecté et a au moins un nœud réel, alors G n'est pas calculable (et le PRDG G_o initial n'est pas cohérent), faire return.
6. Sinon, décomposer G' en ses composantes fortement couplées G_i et appelez $DARTE - VIVIEN(G_i, k+1)$ pour chaque sous-graphe G_i qui a au moins un nœud réel.

Dans la figure 2.10 Il y a deux cycles élémentaires de poids $(1, 0, 1)$ et $(0, 1, 1)$, et cinq auto-boucles de poids $(0, 0, 1)$, $(0, 0, -1)$, $(0, 1, 0)$ (deux fois)

et $(0, -1, 0)$. Donc, tous les arêtes (sauf les arêtes qui appartiennent seulement au cycle de poids $(1, 0, 1)$) appartiennent à un multi-cycle de poids nul. Le sous-graphe G' résultant est donné à la figure 6.11.

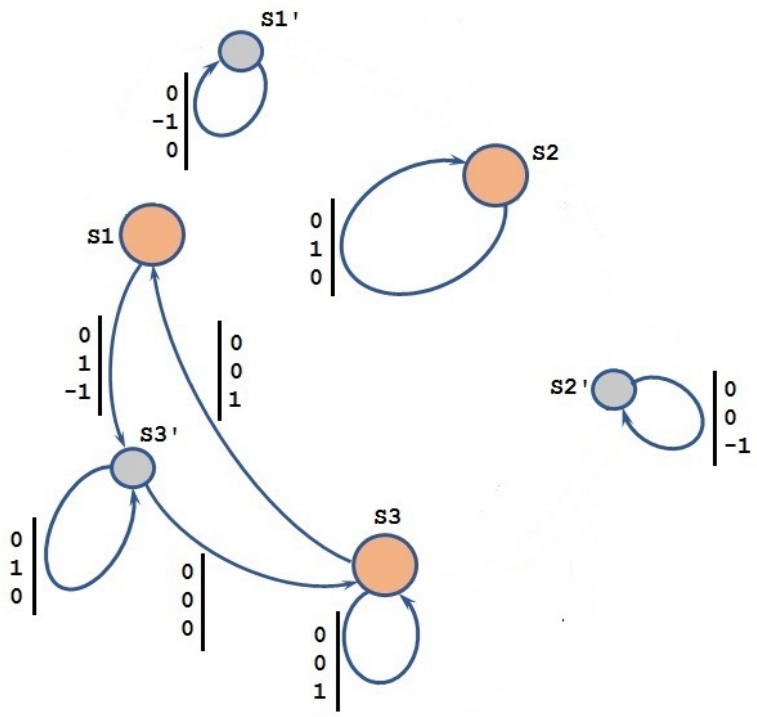

FIGURE 2.11 – RDGs : RDG pour le code 2.11

Les contraintes sur les arêtes dans G' imposent que $X = (x, y, z)$ doit être orthogonal au poids de tous les cycles de G'. Par conséquent, $y = z = 0$. Enfin, en considérant les autres contraintes, on trouve la solution $X = (2, 0, 0)$, $\rho_{S1} = \rho_{S3} = 1$ *et* $\rho_{S2} = 0$. En G', il reste quatre composantes fortement connectés et deux d'entre eux ne sont pas pris en compte car ils ont seulement des nœuds virtuels.

Les deux autres composantes n'ont pas de multi-cycles de poids nul. Le composant fortement couplé avec le nœud unique $S2$ peut être considéré avec le vecteur$X = (0, 1, 0)$, alors qu'on en déduit pour les autres composantes fortement connectée entre autres la solution : $X = (0, 0, 2)$, $\rho_{S1} = 0$ et $\rho_{S3} = 3$. En résumé, les résultats, seront ceux déjà indiqués à la section 6.3 et la transformée bidimensionnels est : $(2i, j)$ pour $S2, (2i + 1, 2k)$pour $S1$ et $(2i + 1, 2k + 3)$ pour $S3$.

Chapitre 3

Le framework de perfectionnement

Sommaire

Préambule

Dans ce chapitre on décrit un ensemble de concepts et d'heuristiques qui conduiront à un renfoncement net de l'efficacité du HLS. Notre objectif consiste en particulier à trouver les stratégies fiables qui mèneront à des perfectionnements en latence. On prévoit l'obtention des taux d'accélérations qui pourront atteindre en générale des rapports de 10× à 50× et excéderont un rapport de cinq (5×) dans quelques cas défavorables. Tout d'abord le parallélisme permettra de faire des améliorations ou bien en latence ou bien en cadence (le débit). L'implémentation du parallélisme est réalisée dans certains cas par l'agencement des ressources d'exécution en un pipeline. Dans ce cas, c'est particulièrement la cadence et non pas la latence qui est améliorée si on raisonne sur un flux de données continu.

Cette première distinction est déjà fondamentale parce que dans la majorité des outils HLS, c'est bien ce parallélisme avec pipelining qui est adopté. On peut dans ce cas faire accroître la cadence à la mesure de dispenser autant de ressources d'exécution, c.-à-d. si l'on augmente le nombre de ressources, on pourra toujours faire grimper la cadence. Cependant et si on raisonne sur

des flux de données continus, la latence quand-t-à elle, est limitée par le potentiel parallèle intrinsèque de l'algorithme en considération conformément à la loi d'Amdahl. Et cette forme de parallélisme par pipelining ne conduit pas réellement à l'amélioration de la latence pour les applications.

Dans ce contexte, les sections qui suivent discuteront en premier lieu les options qui pourront mener spécifiquement dans le HLS aux améliorations nettes des latences dans les applications. Seront analysées surtout les limites théoriques d'accélération engendrées par les potentiels parallèles algorithmiques conformément à la loi d'Amdahl et seront examinées des solutions suggérées pour maintenir des taux d'utilisations raisonnables en ressources. En deuxième lieu seront étudiés les choix de faire des adéquations des solutions déjà proposées pour des architectures reconfigurables. En d'autre terme on aura à voir comment les propositions précédentes pourront être adaptées pour des cibles reconfigurables.

3.1 Le perfectionnement du HLS

Le développement d'un outil HLS fiable nous amène à chercher les meilleures conduites de perfectionnement à appliquer au niveau de toutes les principales facettes. On aura à définir par exemple les bons choix pour ce qui concerne le langage de spécification du code d'entrée. On devra trouver les meilleurs solutions pour l'élaboration de la partie interface du système cible. On serait contraint de définir les optimisations adéquates à appliquer pour permettre les accroissements nettes d'accélérations. On aura aussi à analyser soigneusement les options valides qui permettront de dispenser les plus valables méthodes de tests, de validations et de co-simulations. On fait noter toutefois, qu'on s'intéresse ici en particulier au perfectionnement de la partie centrale du HLS qui est en générale le noyau assurant la génération de la spécification matérielle à travers ce qu'on peut nommer d'une façon globale une instrumentation de code. On n'est pas vraiment attachés aux aspects liés à la définition explicite de l'interface, aux questions qui concernent par exemple les méthodes de test et de validation ou encore les aspects liés à la technologie cible.

La plus part du temps c'est le langage C ou les dérivés du C qui sont adoptés beaucoup plus pour les spécifications comportementales. Il y a assez d'arguments qui ont favorisé ce choix. Ce langage a déjà une bonne réputation dans toutes les différents domaines de l'ingénierie. Les industriels accordent beaucoup d'intérêts à ce qui est dit « courbes d'apprentissages » ou learning curves. Autant que le personnel chargé du développement est déjà familier avec un outil ou une technique ou il peut le devenir aussitôt, autant qu'il y a plus d'opportunités pour en faire plus de productivité. Le niveau d'abstraction du langage semble aussi être bien convenable. En milieu académique, beaucoup de travaux de recherche livrent des suggestions de solutions qui sont

données en des spécifications algorithmiques même s'il le champ d'étude ne concerne même pas de près ou de loin la conception électronique ou même pas le domaine numérique. Ce qui signifie que les spécifications algorithmiques sont jusqu'à présent adéquates pour en faire des abstractions des descriptions comportementales.

On a vu au premier chapitre qu'on a fait déjà recours au SystemC dans quelques outils. Comme c'était déjà mentionnée précédemment, la raison principale était de permettre des représentations avec des précisions à l'échelle de bit et des détails de temporisation et de synchronisme. Ces détails peuvent être utiles particulièrement lorsqu'on atteint la phase de synthèse des interfaces et les ports du système.
On n'est pas préoccupé au cours de la présente contribution par la définition des parties interfaces. On a choisi de considérer le langage C pour la spécification comportementale parce qu'on désire maintenir l'aspect de l'abstraction haut niveau. Et il y aura toujours des options qui permettront d'aménager faiblement les interfaces. L'exemple simple est de considérer des modèles d'interfaces régit par contraintes.
Pour obtenir un outil fiable, il va falloir satisfaire les exigences de performances (incluant latence et cadence) tout en maintenant aussi réduit que possible les occupations en espace et les consommations de puissances. Donc, les solutions proposées ici ont été élaborées en se fixant surtout comme objectif primaire, faire les améliorations les plus ultimes en latence avec le minimum possible de ressources.

3.2 Liens avec les solutions contemporaines

Le HLS maintenant est incontestablement considéré une nécessité pour le renforcement de la productivité. Mais malgré ça, il reste encore à prouver qu'on pourra bien être en mesure aussi d'élaborer avec du HLS des solutions les plus nettement sophistiquées. Dans un papier [Debbi 2016] qu'on a déjà publié, on avait examiné d'une façon succinctes les variantes méthodes employées dans l'ensemble des outils HLS contemporains pour perfectionner les solutions générées. En l'occurrence on a mené une analyse dans les outils HLS : LegUp [Canis 2011b], DWARV [Razfan 2012] et Vivado [Xilinx a].

On considère que le degré de sophistication d'un outil HLS est lié tout d'abord à la logique originale de base autour de laquelle l'outil a été développé. Dans le papier cité précédemment on s'est concentré surtout sur cet aspect sans aborder abondamment les détails d'optimisations. LegUp par exemple se distinguait par sa forte aptitude à supporter et à prendre en charge le plus large jeu de syntaxe et de constructions C. Ce qui signifie qu'à l'origine on s'est bien fixé ce but. On peut aussi estimer que le concept original prévu dans LegUp pour faire des accélérations reposait simplement sur un fait de

partitionnement où les parties dites critiques étaient cédés à une exécution matérielle, alors que les parties non critiques auront à être exécutées sur un soft processeur. Et en conséquence, les taux d'accélérations restaient assez réduits.

Avec l'outil ROCCC [Najjar 2003, Guo 2005, Villarreal 2010] aussi la synthèse fondamentalement reposait sur l'approche dite "bottom-up resuse" qui consistait à faire des systèmes larges à base de blocs élémentaires dupliqués plusieurs fois. Le plus souvent c'était des implémentations en pipeline qui en résultaient. Si ça avait occasionné des améliorations, ces perfections concernaient seulement le débit et non pas la latence.

3.2.1 LegUp

LegUp [Canis 2011a, Canis 2013, Canis 2011b] est une infrastructure de synthèse haut niveau développée et maintenue activement à l'Université de Toronto depuis 2010. Les auteurs projetaient rendre la programmation FPGA plus facile pour les développeurs de logiciels. LegUp est un outil HLS open source ciblant principalement des systèmes hybrides à base de FPGA. Ces systèmes rassemblent principalement un soft processeur et des accélérateurs personnalisés qui sont censés exécuter les sections critiques d'un programme cible. Le flot de synthèse dans LegUp est décrit par la figure 3.1.

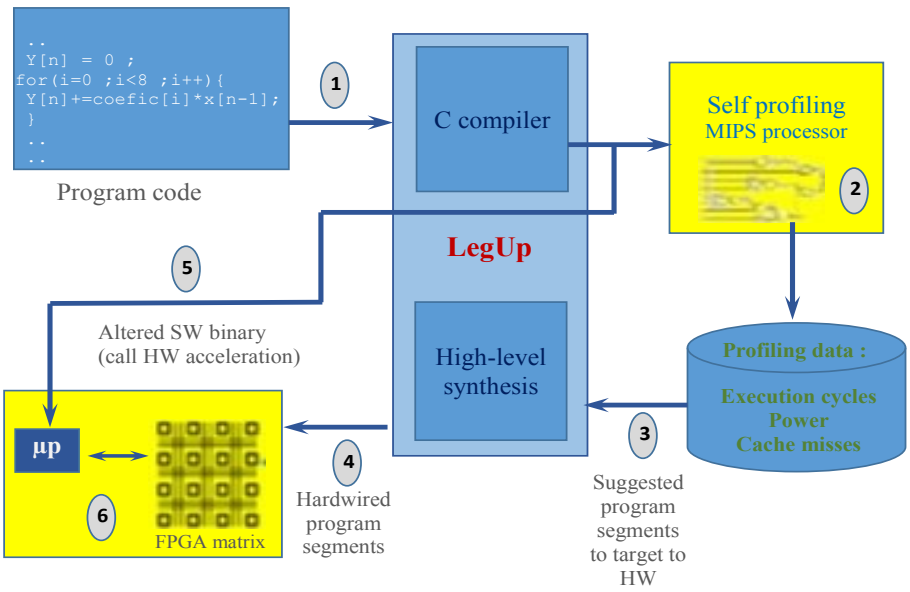

FIGURE 3.1 – Flot de synthèse dans Legup (adapté de [Canis 2011a, Canis 2011b])

LegUp utilise le Framework LLVM (Low Level Virtual Machine) [LLVM] pour les passes de compilation et d'optimisations. A l'étape 1 le programme d'entrée donné en C est compilé en un exécutable binaire ciblant le soft processeur TigerMIPS. Un profileur matériel à l'étape 2 identifie les sections critiques du programme qui pourront bénéficier de l'accélération matérielle. A l'étape 3, l'utilisateur, en fonction des données de profilage, marque les fonctions qui auront à être synthétisées autant qu'accélérateurs matériels. A l'étape 4, Le moteur de synthèse de haut niveau de LegUp est invoqué pour synthétiser ces fonctions comme accélérateurs. A l'étape 5, la source C est recompilée encore une fois en remplaçant les corps des fonctions à accélérer par ce qui est dit des wrappers qui sont utilisés pour appeler des accélérateurs matériels. Ce qui résulte enfin en un système hybride formé d'un processeur et d'accélérateurs. Enfin, le processeur hybride/système-d'accélération s'exécute sur le FPGA.
LegUp prévoit l'utilisation de Pthreads, OPenMP et le pipelining des boucles pour exploiter le parallélisme. Chaque thread devra être implémenté comme accélérateur. Et donc c'est presque le même procédé qui avait été appliqué aux fonctions, il est repris encore avec les threads. La spécification du parallélisme avec les threads est faite évidement au niveau du code. Cette approche ne semble pas permettre d'exploiter le parallélisme à grains fin. Le pipelining comme on l'avait déjà mentionné pourra occasionner des perfectionnements en cadence mais pas en latence.

TABLE 3.1 – Résultats de synthèse avec LegUp et DWARV [Razfan 2012]

Kernel	Slices	Cycles	Max freq	ETAMF	Speedup
sra-legup	370	70	202	0.35	0.82
sra-dwarv	338	64	225	0.28	1.22
loop-legup	122	292	251	1.16	1.30
loop-dwarv	122	380	252	1.51	0.77
fft-legup	1980	7377	98	75.28	0.71
fft-dwarv	3198	8053	150	53.69	1.40
fir-legup	320	223	80	2.79	0.23
fir-dwarv	1063	127	201	0.63	4.41
idct-legup	N/A	24004	N/A	N/A	N/A
idct-dwarv	9519	41338	75	551.17	N/A

En conclusion LegUp ne prévoit pas de moyens pour traiter le parallélisme à grain fin ce qui fait limiter sérieusement son potentiel à raccourcir significativement les latences. Dans la table 3.1 qui reprend partiellement l'ensemble de résultats reportés dans [Razfan 2012] on peut bien parvenir à constater que toutes les accélérations faites avec LegUp ne sont au maximum que dans les rang 3× à 4×.

3.2.2 DWARV

Delft workbench automated reconfigurable VHDL generator (DWARV) est un outil HLS academic aussi, conçu utilisant le framework de compilation Cosy [ACE]. Semblablement à LegUp, il cible aussi spécifiquement les plateformes basées sur FPGA. DWARV 2.0 intègre un sous-ensemble de modules Cosy pour conduire une génération VHDL. Les modules principaux inclus dans ce sous-ensemble sont : Le module *CFront* (frontal ANSI/ISO C) qui crée la représentation intermédiaire IR (intermediat representation). Les modules *cse* et *ssa* qui assurent l'élimination de sous-expressions communes. Le module *psrequiv* qui fait l'annotation des registres/variables à définir dans le VHDL généré. Le module *Fplib* chargé de rechercher et instancier les modèles HW trouvés dans la bibliothèque, et aussi le module *emit* qui va émettre les VHDL synthétisables.

Contrairement à DWARV 2.0 et Legup, le Framework qu'on propose ne fait intervenir aucune plateforme logicielle tierce. Il s'agit d'une production radicalement propre incluant un compilateur C compatible orienté envers l'instrumentation de code. Les accélérations atteintes avec ce Framework qu'on a baptisé "HSCoT" dans un autre papier [Debbi 2017] sont de loin beaucoup meilleures que celles produites avec DWARV 2.0 et LegUp. Ceci est principalement dû au fait que "HSCoT" incorpore prématurément dans les étapes de transformation une approche fiable pour l'exploitation du parallélisme. Cela rend "HSCoT" considérablement distingué parmi les autres offres.

3.2.3 Vivado HLS

L'outil Vivado HLS [Xilinx a] est typiquement voué à la génération de blocs IP correspondants à des fonctions écrites en C. Ces blocs IP pourront être intégrés dans un flot de conception de projets plus larges. Il est à noter que Vivado HLS intègre principalement l'outil AutoESL [Cong 2011] (à l'origine Autopilot [Cong 2006]) que Xilinx a acquis en 2011.

AutoESL à son tour intègre fondamentalement la plateforme de compilation LLVM. Les optimisations de la conception peuvent être effectuées par l'utilisateur final au cours du développement, au moyen de directives particulières que le développeur pourrait insérer dans le programme. Principalement, les optimisations autorisées dans Vivado HLS sont les dés-enroulements de boucles, pipelining des boucles, aplatissement de boucles et autres optimisations relatives aux tableaux. Cependant, ces optimisations ne sont pas réalisables dans toutes les conditions, ils y a plusieurs limitations pour leur applications [Xilinx d]. Les boucles qui n'ont pas de bords déterminés, qui ne sont pas régulières ou qui sont fortement imbriquées ne peuvent être assujetties à des optimisations dans Vivado HLS. Les optimisations telles que

le dés-enroulement de boucles ne peuvent pas être invoquées s'il existe des interdépendances. Ces contraintes ont limité le potentiel de Vivado HLS à atteindre les bons rapport d'accélérations.

3.2.4 Altera OpenCL

Le compilateur Altera OpenCL [Altera a] développé à l'origine par Khronous Group [Khronos a] permet l'utilisation de la programmation basée sur le langage C pour cibler une variété de plateformes, y compris les systèmes à processeurs, à GPU, à DSP, à FPGA et aussi les systèmes hétérogènes. Il est enrichi avec une extension de spécifications C dédiée à l'exploitation du parallélisme et une librairie d'API qui fait l'abstraction des communications entre l'hôte et les accélérateurs.

OpenCL suggère un modèle de programmation global dans lequel sont spécifiées des recommandations pour les plates-formes, pour le flux d'exécution, pour la mémoire, et pour le style de programmation [Khronos b].

La plate-forme recommandée consiste en un hôte connecté à des modules nommés «OpenCL devices». Chaque module est une collection d'unités de calcul (UC) et chaque UC à son tour est formée d'une collection d'éléments de traitement (PE Processing Element). Le moniteur d'exécution d'OpenCL (OpenCL run time) s'exécutant sur l'hôte partage l'ensemble des tâches parallèles pour les affecter aux modules «OpenCL devices».

A l'instar de l'outil ROCCC, OpenCL adopte la logique de faire le pipelining des structures et considère particulièrement des architectures et des structures en pipeline pour le streaming de données. Même si, que OpenCL supporte des optimisations pour le dés-enroulement boucles, comme c'est le cas avec Vivado, il y a plusieurs limitations pour leurs applications avec efficacité.

3.3 Le Framework

Les outils HLS devront permettre la génération de solutions matériels qui pourront assurer des améliorations assez nettes en performances. On estime que ces solutions devront apporter des taux d'accélérations qui excèderont toujours le facteur 5x. Le plus souvent les solutions générées par HLS n'offrent pas réellement des minimisations en occupation de ressources comme celles réalisées avec les synthèses manuelles. Les facteurs d'accélérations qui dépassent l'ordre déjà indiqués sont nécessaires pour compenser l'imperfection en occupation de ressources mais surtout aussi pour promouvoir la réputation du HLS.

On propose ici un ensemble de stratégies qui en major certitude vont assurer les facteurs de gains requis en accélérations. L'idée de base repose sur

l'exploitation du parallélisme inhérent à des niveaux de granularité fine et large à la fois. On propose deux procédures de parallélisme. Chaque procédure va cibler une section de code particulière.
La première visera les sections de code comportant des boucles imbriquées et non imbriquées de toute sorte caractérisées par une intensité d'itérations restreinte. Un nid de boucles (une imbrication de boucle) est qualifié comme étant à intensité d'itération restreinte si le dés-enroulement intégral du nid de boucle pourrait être fait dans un temps de compilation raisonnable. La deuxième procédure visera les régions de code comportant des boucles ou nid de boucles à forte intensité d'itérations. Généralement, ce genre de boucles est surtout fréquent dans les applications orientées aux calculs intensifs. Les variantes de techniques prévues pour le parallélisme de ce type de boucles ont été déjà présentées au chapitre 2. Ces techniques pourront bien être adaptées pour faire le parallélisme des applications ciblées par le HLS. Et bien sûr l'application de ces variantes personnalisées de techniques n'est menée que dans certains scenarios rares (parce que les boucles à forte intensité d'itérations ne sont pas aussi fréquentes dans les applications ciblées par le HLS).

Pour chaque procédure de parallélisme on proposera aussi un modèle d'architecture spécifique qui s'accommodera fermement avec le partitionnement des calculs dicté par la procédure en considération. Pour la première procédure par exemple on prévoit l'adoption de processeurs VLIW processeurs personnalisés. Et pour la deuxième procédure de parallélisme on prévoit l'adoption des vectorisations et des variantes GPU.

3.3.1 Parallélisme des régions de code à forte dépendance de données

3.3.1.1 Considérations théoriques

Tout d'abord les accélérations qui peuvent être achevées en faisant un parallélisme d'un code séquentiel sont dépendantes du potentiel parallèle intrinsèque de l'algorithme en considération. La loi d'Amdahl stipule que l'accélération maximum réalisable par un parallélisme donné ne peut excéder l'inverse de la fraction séquentielle de l'algorithme de sorte qu'on a toujours :

$$S_{max} \leq \frac{1}{f_s} \tag{3.1}$$

S_{max} est l'accélération ou le speedup max qui pourrait être atteint et f_s est la fraction séquentielle inhérente de l'algorithme en considération.

Il est intéressant de garder toujours en esprit cette loi comme repère lorsqu'on apporte des propositions de méthodes pour faire du parallélisme d'applications. Il est intéressant de savoir ou du moins estimer l'ordre des rapports d'accélérations théoriques qui peuvent être réalisés par parallélisme d'un al-

gorithme donné. Est-ce qu'on peut trouver de grandes disparités en potentiel parallèle entre les applications. Est-ce que tous les algorithmes méritent un parallélisme si l'on considère les rapports coûts/performances.

3.3.1.2 Les dépendances de données

Le degré de dépendance de données dans l'algorithme est le paramètre déterminant qui va définir son potentiel parallèle. Les algorithmes dans lesquelles il y a des fortes concentrations de dépendances de données sont sans doute à potentiel parallèle limité. Les applications à calculs intensifs où les boucles possèdent de fortes intensités d'itérations comme ceux considérées dans le chapitre 2, pourront avoir un potentiel parallèle intrinsèque élevé même s'il parait qu'il y a de forte dépendance de données dans le corps interne du nid boucle. S'il y a une indépendance vis-à-vis d'un seul niveau de boucle et si le nombre d'itération est de 100 pour ce niveau de boucles on peut atteindre un rapport d'accélération de 100× même si le corps interne dans le nid de boucle est totalement séquentiel.

Les applications qui pourront faire l'objet d'une implémentation HLS ne possèdent pas en générale des boucles avec ces caractéristiques pareilles qui permettront des vectorisations. Elles possèdent plutôt des boucles à nombre d'itération relativement réduit avec des schémas d'interdépendances très compliqués. Les techniques discutées au chapitre 2 deviennent inappropriées pour les applications qui pourront être assujetti à des implémentations HLS si elles ne subissent pas des adaptations.

3.3.1.3 Plan de parallélisme

On propose un schéma qui permettra de discriminer le parallélisme à des niveaux de granularités fines dans des structures algorithmiques bien complexes même si le degré de parallélisme intrinsèque est réellement limité. On identifie explicitement toutes les dépendances engendrés dans le code puis on établit un nouveau plan de partitionnement. On rassemblera tous les calculs qui sont indépendants dans des groupes qui auront à être exécutés de façon instantanée. Pour distinguer les dépendances, on établit un graphe de dépendance de données DFG Data Flow Graph.

Dans ce graphe, on serait en mesure de distinguer les rassemblements de calculs indépendants possibles. L'identification des dépendances est une tâche qui n'est pas aussi évidente. Beaucoup de travaux [Zhang 2009, Chen 2004, Kim 2010, Li 2015, Sato 2012] ont été consacrés entièrement pour la discrimination des dépendances. Beaucoup de travaux n'avaient pas réussi à donner des résultats satisfaisants. Certaines propositions n'ont pas apporté de solutions appropriées lorsque les dépendances interviennent pour des données et des structures dynamiques à travers des fonctions et des chemins critiques.

Dans la plus part des cas, il s'agissait de programmes de caractérisation de dépendances dynamiques dits aussi « dynamique dependence profilers » [Kim 2010, Li 2015, Sato 2012] qui n'étaient pas réellement adéquats à des réalisations de compilateurs optimiseurs.

On s'est proposé de déterminer les schémas de dépendances à travers une instrumentation rigoureuse de code. A la différence des autres outils HLS comme ceux précédemment discutés et qui intégraient des compilateurs et des plates formes tierces comme LLVM [LLVM], Cosy [ACE] et gcc, la partie frontale du Framework qu'on propose ici est constituée d'un compilateur compatible C propre qu'on a développé nous-même. Donc, on n'est en aucun cas contraints à des dues de licences. Ce premier module assure la génération de la représentation syntaxique abstraite (ASR : Abstract Syntax Representation) du code cible introduit. En appliquant principalement le dés-enroulement des boucles on établit un schéma de dépendances explicites donné sous forme d'un DFG.

La figure 3.2 illustre une allure d'un DFG généré pour le fragment du Code 3.1 qui est une variante similaire à un exemple considéré dans notre papier [Debbi 2016]. Cette forme de représentation va permettre de distinguer les calculs qui sont totalement indépendants et qui peuvent être effectués en parallèle.

Code 3.1

```
int i, j, k, d, H, S;
d = k + 5;
S = 9%3 + k;
k = 5|3/d + 6;
j = d^2 + 1 * S − 4;
for(i = 1; i < 3; i + +) {
        H = j/d|25;
        S = h − 3 + i&9;
}
```

Cette forme de représentation permet aussi d'évaluer la fraction séquentielle de l'algorithme en considération et donc estimer de façon presque exact le potentiel parallèle. Les algorithmes contiennent généralement des interdépendances de données assez complexes de sorte qu'il n'est pas possible le plus souvent d'estimer le potentiel parallèle de façon intuitive.

Le code 3.2 a été aussi introduit dans un papier [Debbi 2017] qu'on publié déjà et il est évoqué ici comme un deuxième exemple où on pourra obtenir un schéma de dépendance relativement complexe. Il n'est pas tout à fait à portée d'estimer même d'une façon approchée le potentiel parallèle dans ce

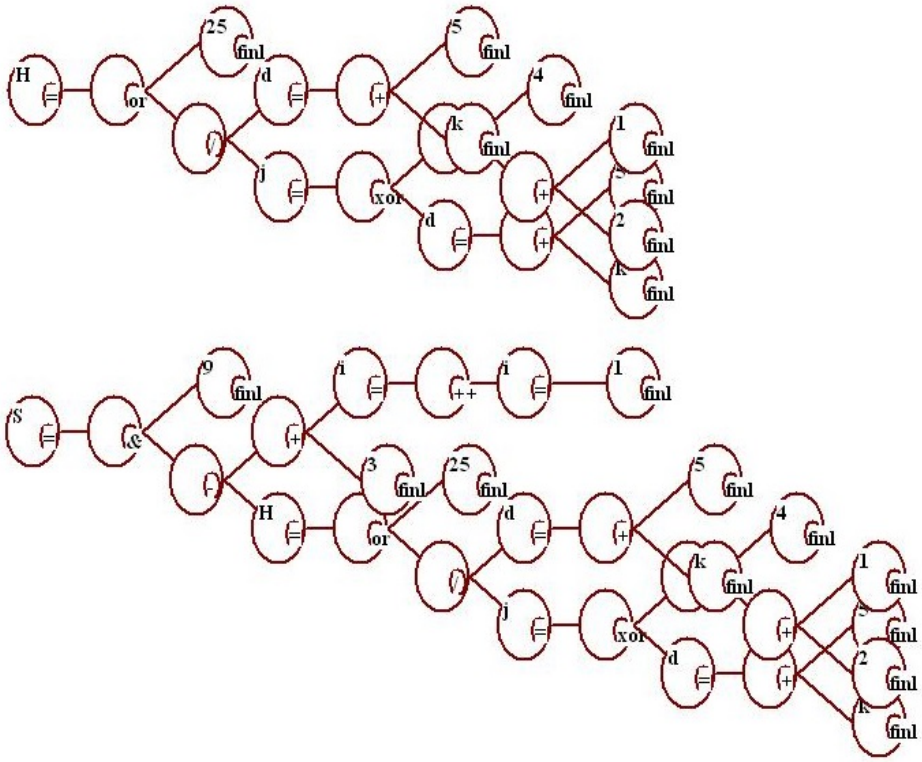

FIGURE 3.2 – DFG correspondant au code 3.1 [Debbi 2016]

fragment de code. La figure 3.3 montre le DFG correspondant généré au sein du compilateur qu'on propose pour l'instrumentation du code.

L'allure globale du DFG nous permet d'en déduire la proportion du potentiel parallèle dans l'algorithme cible. Le graphe révèle si l'algorithme est intrinsèquement à fort parallèle potentiel ou à l'encontre, il est intrinsèquement à forte fraction séquentielle.

Si les arbres du graphe se prolongent en profondeur c'est-à-dire dans le sens horizontal, cela indique que l'algorithme tend à être beaucoup plus à nature séquentiel. Si au contraire les arbres du graphe ont tendance à avoir un élargissement dans le sens orthogonal avec un accroissement en nombre de branches et sans étalement en profondeur, ceci donne un signe que l'algorithme renferme comme même un certain taux de potentiel parallèle.

On peut à partir de ce graphe calculer la fraction séquentielle intrinsèque de l'algorithme. La fraction séquentielle f_s est donnée par le rapport :

$$fs = \frac{temps\ d'excution\ des\ calculs\ lies\ par\ dpendances}{temps\ d'excution\ total} \tag{3.2}$$

si l'on admet que le temps d'exécution est pareil pour toute les opérations, f_s

Code 3.2
Filtrage Adaptatif comme exemple renfermant des interdépendances dans des bouclés imbriquées

a) Pseudo code

```
1 :   for k = k0 : k0 + N
2 :       Yestimated = W^T * Xinput
3 :       err = Yestimated − Ywanted
4 :       W = W + μm * err * Xinput
5 :   end for
```

b) Variante du filtrage adaptatif explicitement en C

```
1 :  for(k = k0 + filter_order; k < k0 + N + filter_order; k++)
2 :      {
3 :      for(i = 0; i < filter_order; i++)
4 :          Yestimated[k] = Yestimated[k] + W[i] * Xinput[i + k − filter_order]
5 :      for(i = 0; i < filter_order; i++)
6 :          err[i] = Yestimated[i + k − filter_order] − Ywanted[i]
7 :      for(i = 0; i < filter_order; i++)
8 :          W[i] = W[i] + μm * err[i] * Xinput[i + k − filter_order]
9 :  }
```

est peut être aussi exprimée comme :

$$fs = \frac{nombre\ de\ niveaux\ de\ profondeurs\ max}{nombre\ d'oprations\ totales} \tag{3.3}$$

La figure 3.4 [Debbi 2018] illustre l'architecture globale du Framework. Jusqu'ici est mentionnée juste la partie frontale. Le code cible subit une première transformation à travers les parseurs lexicaux et syntaxiques pour donner la représentation syntaxique abstraite (ASR Abstract Syntax Representation) considérée ici comme première forme de représentation et dénotée IR^1 pour Intermediat Representation. La structure IR^2 dénote la deuxième forme de représentation générée à partir de IR^1 à travers une série de transformations incluant essentiellement des chainages « use-def ». on arrive à construire des DFGs en appliquant aussi entre autre le dés-enroulement des boucles et les « inlinning » de fonctions. Cette nouvelle structure donnée sous forme de graphes a été dénommé aussi « Map » dans nos papiers [Debbi 2017, Debbi 2018].

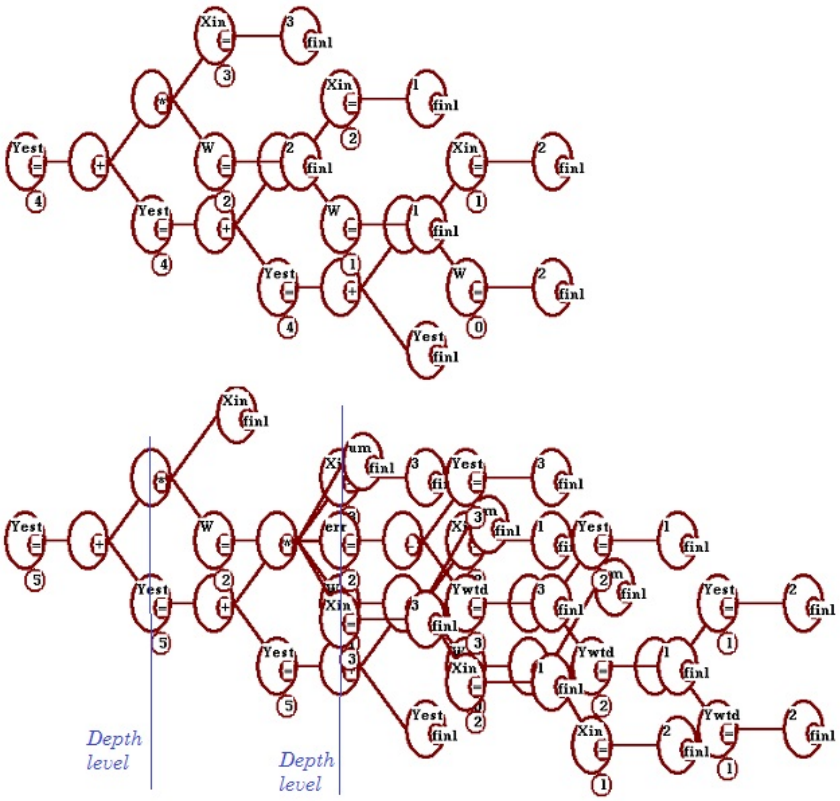

FIGURE 3.3 – DFG généré au sein du Framework correspondant au code 3.2 [Debbi 2018]

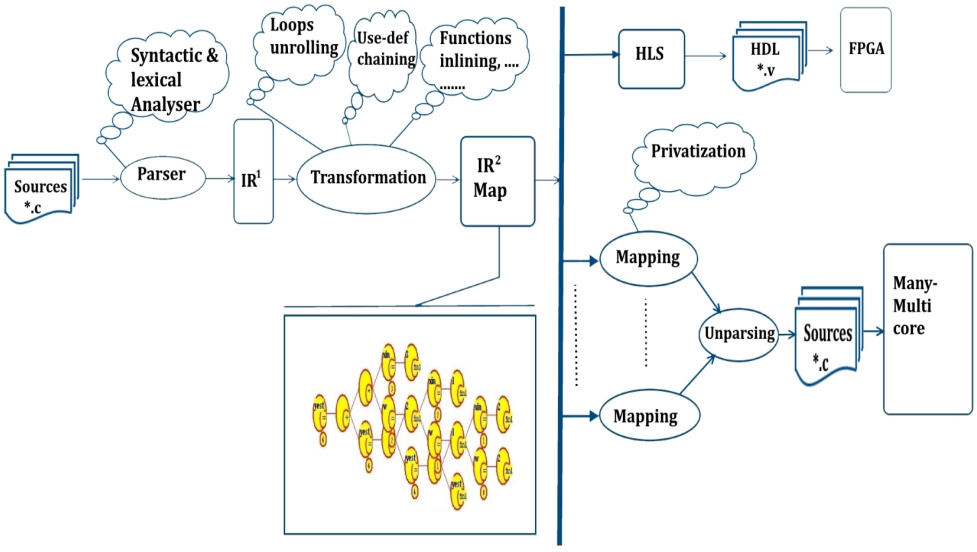

FIGURE 3.4 – Architecture du Framework

Le partitionnement des calculs est la deuxième tache fondamentale impliquée dans le processus de synthèse. Il s'agit de repérer dans la «Map» les calculs qui sont indépendants et de les rassembler pour les exécuter en un cycle unique. Il s'agit des calculs qui pourront être repérés sur un même niveau de profondeur dans la « Map ». Au niveau de la figure 3.3 ces ensembles sont désignés par des lignes perpendiculaires en bleu.

On prévoit deux scénarios au sein desquels on appliquera ce partitionnement. Le premier scénario est envisagé lorsqu'on désire faire la synthèse de modules qui devront avoir des performances suprêmes en latence et en débit et qui auront à assurer des traitements élémentaires comme le filtrage et les transformations de Fourier ou en cosinus DCT. Tous les traitements qui ont des utilisations fréquentes et qui auront à être implémenté comme cœurs fondamentaux dans des applications larges pourront avoir cette forme de synthèse. Dans ce cas on réalise ce qui est dit distribution arithmétique totale ou (TDA Total Distribution Arithmetic). L'occupation en ressources serait probablement considérable mais les performances en latences et en cadence seront optimales.

Cette forme de synthèse a été appliquée pour une variété de filtres numériques et des transformés de Fourier et la transformés en cosinus DCT. Les résultats ont été publiés dans [Debbi 2016] et ils sont aussi repris encore ici et données par les Tables 3.2, 3.3 et 3.4. Au niveau de la table 3.3, on constate que les latences et les accélérations amenées par nos solutions sont de loin meilleures que ceux apportées par les autres outils HLS. La raison est simple et due justement au fait que l'exploitation du parallélisme intrinsèque a été faite façon plus rigoureuse.

TABLE 3.2 – Résultats de synthèse avec "HSCoT" (filtrage et transformées) [Debbi 2016]

Kernel		Occupation en ressources				perfectionnement
Filtrage	Dimensions du filtre N,M	Nombre de registres	nombre de de LUT	Nombre BoundedIOB	Nombre de DSP48E	Latence (cycle H)
IIR	N=16, M=8	96	53	17	16	3
	N=24, M=8	80	36	17	22	3
DFT	8 points	82	99	65	4	8
	16 points	1141	2099	129	9	8
DCT	4 points	66	84	65	19	8
	8 points	34	36	65	73	10

Ces traitements même sont proposés et fourni sous forme de cœurs IP (IP cores) par une variété de constructeurs et d'auteurs. Les performances de ces blocs IP sont récapitulées dans la table 3.4. On peut bien constater cette

TABLE 3.3 – Résultats de synthèse avec "HSCoT" comparés aux solutions générées par LegUp et DWARV'[Debbi 2016]

Outil	Kernel							
HSCoT	fir		loop		fft		dct	
	Cycles 3	Speedup > 30	Cycles 3	Speedup > 10	Cycles 10	Speedup > 200	Cycles 10	Speedup > 300
Legup [Razfan 2012]	223	0.23	292	1.30	7377	0.71	24004	N/A
DWARV [Razfan 2012]	127	4.41	380	0.77	8053	1.40	41338	N/A

fois ci, que les solutions sont bien optimisées. Il y a déjà une similitude en performances avec nos solutions. Mais il faut noter bien que ces données mentionnées déjà dans la table 3.4 ne s'agissent pas de réalisations et de solutions générés par outils quelconques mais ce sont des synthèses manuelles, alors que nos solutions sont des résultats générées par le Framework proposé. Les données reportées dans ces tables montrent que la faisabilité de systèmes élémentaires avec des performances extrêmes approchant les ordres théoriques est toujours possible. On peut bien être convaincu que ces réalisations sont aussi un résultat des exploitations du parallélisme intrinsèque mais qui sont faites manuellement. En particulier pour ce petit sous ensemble de traitements, la détermination de façon intuitive des ensembles de calculs qui peuvent être menés en parallèle est ici possible. Mais il serait presque impossible de mener intuitivement et manuellement un parallélisme si les schémas d'interdépendances étaient assez complexes. Il n'est pas vraiment possible de déterminer intuitivement un partitionnement de calculs parallèles pour le filtrage adaptatif par exemple comme celui décrit déjà par la variante de code 3.2.

L'exploitation ultime du parallélisme et le partitionnement optimal des calculs parallèles pour atteindre des ordres théoriques d'accélération pour les traitements pareils (analogues à l'exemple du filtrage adaptatif) ne peuvent être rendus possible sauf si on utilise un outil de parallélisme pareil au Framework qu'on a proposé.

Cette première solution serait préférée lorsqu'il s'agit de cœurs (cores) à volume relativement restreint et qui sont régulièrement invoqués dans des systèmes plus larges comme les filtres et les transformées. Lorsqu'il s'agit de faire une synthèse de systèmes plus larges comme ceux qui sont données dans le Benchmark CHStone [Hara 2008] à savoir : ADPCM, Blowfish, AES, GSM, ... etc. Dans ce cas il ne semble pas être approprié d'opter toujours à une TDA. Il serait plus convenable de trouver une solution qui assure toujours la prise en charge du partitionnement parallèle mais qui soit moins gourmande en ressources. Il va s'agir donc d'un chemin données et d'une structures de contrôle orientées lots (batchs). Les réalisations existantes et qui peuvent s'accommo-

TABLE 3.4 – Résultats de synthèse avec "HSCoT" comparés à des réalisations IP tierces [Debbi 2016]

Core	Auteur	Spécification	Perfermance	
			Débit	Latence
fir	HSCoT	FIR 128 tap	200 Msps	3 cycle H
	Altera [Altera b]	FIR 97 tap	290 Msps	x
	Xilinx [Xilinx b]	FIR 2-2048 tap	1 sample/clock	x
D/FFT	HSCoT	DFT 128 point à 62 MHz	1 sample/clock	10 cycle H
	Huan [Huan 2008]	FFT 128 point	x	582 cycle H
	Yousri [Yousri 2011]	FFT	1 sample/clock	x
	Xilinx [Xilinx c]	DFT 1200point à 245 MHz	x	26μ s
	OpenCore [OpenCores b]	FFT 128 point	x	310 cycle horloge
	Actel [Actel]	FFT 256 point	x	11 μ s
DCT	HSCoT	block 8x8 à 61 MHz	16 sample/clock	10 cycle H
	Enas & Thomas [Enas 2010]	block 8x8 à 84MHz	x	123 cycle H
	Fakhari & Fathy [Fakhari 2010]	block 8x8 à 67 MHz	1 sample/clock	67 cycle H
	Visengi [visengi]	block 8x8 à 229 MHz	x	95 cycle H
	OpenCores [OpenCores a]	block 8x8 à 300MHz	x	132 cycle H

der avec les structures obtenues déjà au sein du Framework sont principalement les VLIW processeurs. Ces processeurs ne se sont pas répondu largement dans l'industrie. L'exploitation de ces processeurs avec les compilateurs classiques ne permettait pas de garantir les efficacités escomptées.

Comme la réalisation physique de ce type processeur peut compromettre beaucoup de contraintes, on a prévu de faire des émulations de processeur. Les émulations eux même peuvent se faire à différents niveaux d'abstraction. Une émulation à haut niveau d'abstraction pourra avoir lieu et va concerner juste le code. Une autre variante d'émulation qui est à un niveau d'abstraction moins élevé va concerner le code et quelques composantes matérielles limitées comme les « file register » et les UALs.

Pour assurer les adéquations nécessaires avec les structures parallèle générées par la partie frontale du Framework et qui permettront donc de garantir ou du moins se rapprocher des accélérations trouvés au niveau des DFGs, on avait en déduit qu'il faudra trouver une architecture cible pour le chemin de données qui doit satisfaire un nombre d'exigences déterminées. Les recommandations données juste ci-dessous récapitulent l'ensemble des exigences qui paraissent être les plus importantes.

- ⋆ Le processeur doit être capable d'exécuter plusieurs opérations simultanées. Un minimum de quatre opérations peut paraitre un ordre raisonnable. En admettant que le système peut contenir plusieurs instances de ce processeur, il y a de fortes chances de réaliser des ordres d'accélérations qui excèdent 10×. une longueur du mot processeur qui tient dans un gap allant de 64 à 128 bits suffit largement pour supporter plus que quatre opérations avec leurs opérandes. Au-delà de cet ordre, il faudra réexaminer encore le problème d'encombrement au niveau du chemin de données.

- ⋆ les structures générées par le compilateur créent des opportunités pour la localité des données. Les accès aux données se font conformément à un schéma particulier. Chaque donnée résultante d'une opération est immédiatement impliquée comme opérande dans l'opération qui devra suivre. Les accès mémoires vont être ainsi considérablement réduits car les résultats intermédiaires des opérations sont maintenus au niveau du « file register » ou dans le cache pour être consommées immédiatement par l'opération suivante. Le processeur dans ce cas réalise de façon indirecte une privatisation. La privatisation [Li 1992, Tu 1993] est une technique utilisée abondamment dans les compilateurs [Bae 2013, Campanoni 2012, Blume 1996] dédiés au parallélisme.

- ⋆ Le chemin de données doit assurer la souplesse des mouvements de données entre les parties opératives, le « file registre » et le cache d'une façon qui s'accommode absolument avec ce schéma d'exécution.

- ⋆ Les exécutions spéculatives et les prédictions de branche deviennent dans ce nouveau schéma d'exécution inutiles. Le partitionnement qui devra être appliqué aux structures générées impose un ordre strict sur les opérations. Le compilateur d'une certaine façon a anticipé tous les effets qui peuvent être apportés par les techniques de spéculation.

L'outil proposé ici est en mesure d'assurer l'évaluation du potentiel parallèle globale au niveau d'une application cible donnée toute entière, d'assurer un recensement sélectif de ce potentiel parallèle au niveau des fonctions et des fragments de code particuliers et la caractérisation des dépendances intra-boucles et inter-fonctions de façon entièrement fiable. Tout ça confirme bien qu'on détient par ce Framework un potentiel d'instrumentions considérable. Nos premières instrumentations apportées à la suite des programmes incluse dans le Benchmark CHStone ont conduit aux résultats énoncés par les Tables 3.5, 3.6, 3.7 et 3.8.

TABLE 3.5 – Mesures des accélérations pouvant être achevées dans un sous-ensemble du Benchmark CHStone [Debbi 2018]

Program	ADPCM	AES	GSM	BLOWFISH
Number of functions	15	11	12	6
Test vector length	100	16	160	120
Workload order	Θ(35500)	Θ(6500)	Θ(32300)	Θ(79700)
Cost(Dependence-enchained-workload)	256	206	440	2245
Speedup	138×	31×	73×	35×

TABLE 3.6 – Recensement du potentiel parallèle au niveau des fonctions de l'application GSM [Debbi 2018]

Program	GSM		
Function	Autocorrelation()	Reflection_ coefficients()	Quantization_ and_coding()
Input vector length	160	8(LARc)	8(LARc)
Workload order	Θ(23900)	Θ(6300)	Θ(1600)
Cost(Enchained workload)	351	86	41
Speedup	68×	73×	39×

3.3.2 Optimisations de l'occupation de ressources dans les cibles configurables

Avec les cibles configurables, on arrive à détenir certaines options pour faire des minimisations en ressources. Des zones spécifiques programmables dans les FPGA par exemple peuvent être exploitées dynamiquement en continu pour supporter les différents modules optimisés de l'application. Il serait possible de faire entrelacer temporellement plusieurs modules sur une zone unique. Les outils HLS peuvent confier de bonnes opportunités pour en faire profite de cette option s'ils incluent des modules d'instrumentations efficaces. Le degré de fiabilité des résultats va dépendre étroitement du degré de perfectionnement des modules d'instrumentations. Dans l'outil LegUp par exemple est comme c'est fait déjà avancé (voir section 3.2.1), un nombre de fonctions qui sont considérés comme cœurs critiques (critical kernels) étaient sélectionnés pour faire l'objet d'une projection dans le FPGA au moment où le reste du code est exécuté par le soft-processeur. Avec LegUp on se limitait juste à déterminer

TABLE 3.7 – Recensement du potentiel parallèle au niveau des fonctions de l'application AES [Debbi 2018]

Program	AES	
Function	KeySchedule()	MixColumn_AddRoundKey()
Input vector length	16	16
Workload order	$\Theta(3100)$	$\Theta(800)$
Cost(Enchained workload)	202	16
Speedup	15.3×	50×

TABLE 3.8 – Recensement du potentiel parallèle au niveau de la fonction de BF_set_key() de l'application BLOWFISH [Debbi 2018]

Program	BLOWFISH
Function	BF_set_key()
Input vector length	18
Workload order	$O(11700)$
Cost(Enchained workload)	350
Speedup	33×

les corps de fonctions et sélectionner celles qui sont volumineuses pour les considérer comme parties critiques devront faire l'objet d'une implémentation matérielle sur FPGA. Les fonctions volumineuses ne sont pas forcément à potentiel parallèle élevé. Les auteurs de LegUp ne mentionnent pas de conduite particulière vis-à-vis de la mesure du potentiel parallèle intrinsèque et vis-à-vis du profilage des dépendances.

La minimisation dans les cibles configurables va être conduite convenablement seulement si on arrive à faire une instrumentation rigoureuse du code. Le Framework qu'on a proposé est assez sophistiqué et permet de mener des exploitations rigoureuses des cibles configurables.

L'instrumentation faite au sein du Framework nous confie un moyen pour la détermination du potentiel parallèle intrinsèque de n'importe quelle fonction faisant partie de l'application cible comme c'est déjà illustré dans les tables 3.6.,3.7 et la table 3.8. Par cette instrumentation on arrive aussi à identifier de façon suffisamment précise le schéma de dépendances à un niveau gras de granularité (coarse grain level).

On est parvenu précédemment à prévoir trois schéma pour établir un parallélisme. Dans Le premier on avait ciblé des traitements élémentaires possédant des potentiels parallèles considérables comme les transformés FFT et DCT. On a vu qu'il fallait faire intervenir ce qu'on a appelé « la distribution arithmétique total ». Elle est applicable à tout module qui peut être considéré comme macro-opérateur à fort potentiel parallèle intrinsèque. Dans ce cas c'est un codage en dur qui est réalisé. La deuxième technique avait ciblé les portions de code à potentiel parallèle moyen. On a proposé de faire un mappage du code transformé envers des instances de VLWI processeurs optimisés pour le parallélisme. La troisième forme de parallélisme est discutée succinctement dans la section suivante et concerne les boucles à forte intensité d'itérations. Donc, on doit être en mesure aussi à travers l'instrumentation de discriminer le type de région afin de la faire entrelacer dans la zone configurable adéquate.

3.3.3 Parallélisme des régions à forte concentration de calculs

Les zones ciblées ici sont en particulier les boucles à nombre d'itérations élevé. Ce type de boucles n'est pas aussi fréquent dans les applications qui pourront être assujetties à une synthèse HLS, mais leur présence occasionnelle dans un programme cible pourra rendre un facteur d'accélération considérable si le parallélisme à ce niveau est mené convenablement car en générale ces boucles renferment un grand potentiel parallèle.

Dans ces régions on pourra envisager les deux scénarios majeurs suivant. Dans le premier scénario on pourra avoir des boucles "régulières" à forte intensité d'itérations qui sont analogues à ceux discutés dans le chapitre 2. Dans le deuxième scénario on aura à avoir des régions à schéma d'interdépendances assez complexes qui sont à l'intérieur de boucles "régulières" à forte intensité d'itérations. Dans ce cas, les régions internes sont traitées avec l'approche évoquée précédemment pour les rendre parallèles. Alors que les boucles externes vont être rendu parallèles utilisant un nouveau ensemble d'algorithmes basés sur les méthodes d'analyse analogues à ceux discutés dans le chapitre 2.

La solution proposée ici est dans beaucoup de cas beaucoup plus fiable que tous les autres algorithmes discutés au chapitre 2. Elle fait intervenir une procédure qui n'est pas aussi compliquée au moment où elle permet de détecter le parallélisme non détecté par l'algorithme de Kenedy, et elle donne des résultats meilleurs que ceux aquis avec l'algorithme de Wolf et Lam tant sur le plan de taux d'accélération que sur le plan de la simplicité d'implémentation.

Le principe de l'algorithme est énoncé ici succinctement pour être redonné sous forme d'un schémas de transformations qui mèneront à la version parallèle du nid de boucles considéré. Les résultats vont être démontrés en considérant tous les exemples qu étaient exactement vus au chapitre 2 et adoptés dans les différents algorithmes.

Code 3.3

```
for i = 1 to n
    for j = 1 to n
        A(i,j) = A(i,j-1) + A(i-1,j-1)
    endfor
endfor
```

On adopte une représentation par vecteurs de directions pour la description des dépendances. On fait intervenir deux concepts : L'échelonnement et la transposition. Les boucles sont réécrites séparément pour chaque vecteur de direction. Le code 3.3 pris comme premier exemple est réécrit en appliquant la fragmentation pour obtenir la variante du code 3.4.

Code 3.4 obtenu par échelonnement du code 3.3

```
for i = 1 to n
    for j = 1 to n
        A(i,j) = A(i,j-1)
    endfor
endfor
for i = 1 to n
    for j = 1 to n
        A(i,j) = A(i,j) + A(i-1,j-1)
    endfor
endfor
```

Si le vecteur est colinéaire avec un vecteur de base, on forme les boucles DOALL correspondantes aux autres vecteurs de bases non colinéaires. Si non si le vecteur de direction est non colinéaire avec aucun vecteur de base alors on doit faire un changement de base orthonormée où le vecteur en considération est en colinéarité avec l'un de ces vecteurs de base. La matrice définit pour le changement de base va elle-même constituer les paramètres du système linéaire dont les solutions seront les vecteurs de la transformations recherchés.

3.3.3.1 L'algorithme

1. S'il y a plusieurs vecteurs de directions alors faire une fragmentation (échelonnement)
2. Pour chaque vecteur de direction faire
   Si le vecteur est colinéaire faire les DOALL correspondants Si non, faire la transformation décrite juste à la suite de l'algorithme et appliquer les DOALL possibles
3. Faire les agrégations.

3.3.3.2 La transformation

Pour tout vecteur de dépendance $V(c_1, c_2, ...c_k)$ non colinéaire avec les vecteurs de la base orthonormée initiale, former la matrice M de passage utilisée pour le changement de la base orthonormée B qui inclue ce vecteur. B est formée par les vecteurs $(V \perp V_2 \perp V_3 \perp ....V_k))$.
Cette matrice aura la forme suivante

$$M \begin{pmatrix} c_1 & -c_1 & c_1.. & . & . & . & . & .c_1 \\ c_2 & c_2 & -c_2.. & . & . & . & . & .c_2 \\ c_3 & c_3 & c_3.. & . & . & . & . & .c_3 \\ .. & .. & .. & . & & & & \\ .. & .. & .. & . & & & & \\ c_k & c_k & c_k.. & . & . & . & . & -c_k \end{pmatrix}$$

La transformation qui conduit au nid de boucles parallèle équivalent est obtenue en calculant les racines du système d'équations M.R = V formé par cette matrice.

$$M \begin{pmatrix} c_1 & -c_1 & c_1.. & . & . & . & . & .c_1 \\ c_2 & c_2 & -c_2.. & . & . & . & . & .c_2 \\ c_3 & c_3 & c_3.. & . & . & . & . & .c_3 \\ .. & .. & .. & . & & & & \\ .. & .. & .. & . & & & & \\ c_k & c_k & c_k.. & . & . & . & . & -c_k \end{pmatrix} \times \begin{pmatrix} r_1 \\ r_2 \\ .. \\ .. \\ .. \\ r_k \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ .. \\ .. \\ .. \\ v_k \end{pmatrix}$$

$R(r_1 r_2. . . . . .r_k)$ est le vecteur racine.

$$r_1 = \frac{\Delta_1}{\Delta}, r_2 = \frac{\Delta_2}{\Delta}, ...r_k = \frac{\Delta_k}{\Delta}$$

L'application de cet algorithme à l'exemple précédent donne les boucles parallèles décrites par le code 3.5.

Code 3.5 résultats du parallélisme du code 3.3

$$
\begin{array}{l}
DoALL\ i = 1\ to\ n \\
\quad DoSEQ\ j = 1\ to\ n \\
\quad\quad A(i,j) = A(i,j-1) \\
\quad endfor \\
endfor \\
\\
DoALLfor\ x = -\frac{n}{2}\ to\ \frac{n}{2} \\
\quad DoSEQ\ y = 1\ to\ n \\
\quad\quad if((x+y>0)\ and\ (x+y<n)\ and\ (y-x>0)\ and\ (y-x<n)) \\
\quad\quad\quad a(x+y,y-x) = a(x+y,y-x) + a(x+y-1,y-x-1); \\
\quad endfor \\
endfor \\
\\
DoALLfor\ x = -\frac{n}{2}\ to\ \frac{n}{2} \\
\quad DoSEQ\ y = 1\ to\ n \\
\quad\quad if((x+y>0)\ and\ (x+y<n)\ and\ (y-x>0)\ and\ (y-x<n)) \\
\quad\quad\quad a(x+y+1,y-x) = a(x+y+1,y-x) + a(x+y-1,y-x-1); \\
\quad endfor \\
endfor
\end{array}
$$

Conclusion

Dans cette thèse on s'est attaché à déterminer les conduites à adopter lors du développement d'un outil HLS pour rendre beaucoup plus efficaces les solutions générées. Plus spécifiquement on a proposé des stratégies pour l'exploitation du parallélisme intrinsèque des algorithmes en vue de faire des améliorations nettes en latence et en cadence. Cette approche ne peut convenir dans les cas classiques où les synthèses sont menées de façon manuelle et intuitive. Cette approche n'est applicable que dans le cadre d'un HLS, car elle implique des phases d'instrumentations et d'analyses avancées du code d'entrée. Le parallélisme intrinsèque dans les algorithmes n'est pas explicitement exposé et ne peut être détecté intuitivement. A travers différentes analyses et des séries d'instrumentation faites sur des exemples du code on avait parvenu à conclure qu'il fallait prévoir plusieurs schémas pour établir un parallélisme. Chaque schéma est censé être appliqué au niveau d'une région de code particulière. Les régions qui consistaient en des traitements élémentaires et possédant des potentiels parallèles considérables comme les transformés FFT et DCT, on a vu qu'il fallait faire intervenir ce qu'on a appelé « la distribution arithmétique total ». Cette forme de parallélisme est applicable à tout module qui peut être considéré comme macro-opérateur à fort potentiel parallèle intrinsèque. Dans ce cas c'est un codage en dur qui est réalisé. Une deuxième forme de parallélisme était prévue pour cibler les portions de code à potentiel parallèle moyen. Ces régions sont les plus fréquentes dans les applications pouvant faire l'objet d'une implémentation HLS. On a proposé de faire un mappage du code transformé envers des instances de VLWI processeurs optimisés pour le parallélisme. La troisième forme de parallélisme a été prévue pour les régions de boucles à forte intensité d'itérations.

Annexe

Sous ensemble de specifications du langage HDL Verilog

Cette annexe reporte une compilation résumant quelques parties essentielles de la référence au langage de synthèse matériel Verilog.

1. Déclarations des modules

```
module M (p1, p2, p3, p4);
          input p1, p2;
          output [7:0] P3;
          inout P4;
          reg [7:0] R1, M1[1:1024];
          wire W1, W2, W3, W4;
          parameter C1 = "This is a string";
      initial
          begin : BlockName
              Statements
          end
      always
          begin : BlockName
              Statements
          end
      // Continuous assignments...
          assign W1 = Expression;
          wire (Strong1, Weak0) [3:0] # (2,3) W2 = Expression;
      // Module instances...
          COMP U1 (W3, W4);
          COMP U2 (.P1(W3), .P2(W4));
      task T1;
          input A1;
          inout A2;
          output A3;
          begin
              // Statements
          end
      endtask
      function [7:0] F1;
          input A1;
          begin
              // Statements
              F1 = Expression;
          end
      endfunction
endmodule
```

2. Les expressions

```
#delay
wait (Expression)
@(A or B or C)
@(posedge Clk)
Reg = Expression;
Reg <= Expression;
VectorReg[Bit] = Expression;
VectorReg[MSB:LSB] = Expression;
Memory[Address] = Expression;
assign Reg = Expression
deassign Reg;
TaskEnable(...);
disable TaskOrBlock;
-> EventName;
if (Condition)
...
else if (Condition)
...
else
...
case (Selection)
Choice1 :
...
Choice2, Choice3 :
...
default :
...
endcase
for (I=0; I<MAX; I=I+1)
...
repeat (8)
...
while (Condition)
...
forever
...
```

3. Éléments de la syntaxe Verilog

Always

Contient une ou plusieurs instructions (affectations procédurales, tâche activée, des constructions if, case et des boucles), qui sont exécutées à plusieurs reprises tout au long d'une exécution de simulation.

Syntaxe

```
Always
   statement
```

Exemple

```
always @(posedge Clock or negedge Reset)
   begin
      if (!Reset) // Asynchronous reset
         Count <= 0;
      else
         if (!Load) // Synchronous load
            Count <= Data;
         else
            Count <= Count + 1;
   end
```

begin

Utilisé pour regrouper les instructions, afin qu'elles s'exécutent en séquence.

Syntaxe

```
begin [: Label [ Declarations...]]
   Statements...
end
Declaration = either Register Parameter Event
```

Exemple

```
initial
   begin : GenerateInputs
      integer I;
      for (I = 0; I < 8; I = I + 1)
          #Period {A, B, C} = I;
   end
```

Case

Exécute conditionnellement au plus une branche, en fonction de la valeur de l'expression.

Syntaxe

```
CaseKeyword (Expression)
   Expression,... : Statement
   Expression,... : Statement
   ... ...
   [default [:] Statement]
endcase
CaseKeyword = either case casex casez
```

Exemple1

```
case (Address)
   0 : A <= 1;   // Select a single Address value
   1 : begin     // Execute more than one statement
         A <= 1;
         B <= 1;
       end
   2, 3, 4 : C <= 1; // Pick out several Address values
   default :
       $display("Illegal Address value ");
endcase
```

Exemple2

```
casex (Instruction)
   8'b000xxxxx : Valid <= 1;
   8'b1xxxxxxx : Neg <= 1;
   default
       begin
         Valid <= 0;
         Neg <= 0;
       end
endcase
```

affectation continue

crée des événements sur un ou plusieurs lignes chaque fois que l'expression associée à la ligne ou au registre change de valeur.

Syntaxe

```
assign [Strength] [Delay] NetLValue = Expression,
           NetLValue = Expression,
           ...;
```

```
NetType [Expansion] [Strength] [Range] [Delay]
       NetName = Expression,
       NetName = Expression,
```

Exemple

```
wire cout, cin;
wire [31:0] sum, a, b;
assign cout, sum = a + b + cin;
wire enable;
reg [7:0] data;
wire [7:0] #(3,4) f = enable ? data : 8'bz;
```

Delay

Les retards modélisent le délai de propagation des composants et des connexions.

Syntaxe

```
{either}
#DelayValue
#(DelayValue[,DelayValue[,DelayValue]])   {Rise,Fall,Turn-Off}
```

Disable

Force l'arrêt d'exécution d'une tâche active ou d'un bloc

Syntaxe

```
disable BlockOrTaskName;
```

Exemple

```
begin : Break
   forever
   begin : Continue
        ...
        disable Continue;    // Continue with next iteration
        ...
        disable Break;       // Exit the forever loop
        ...
   end   // Continue
end      // Break
```

Event

Les événements peuvent être utilisés pour décrire les synchronisation dans les communications et les modèles comportementaux

Syntaxe

```
event Name,...;     {Declare the event}
-> EventName;     {Trigger the event}
```

Exemple

```
event StartClock, StopClock;
always
fork
   begin : ClockGenerator
        Clock = 0;
        @StartClock
        forever
           #HalfPeriod Clock = !Clock;
   end
   @StopClock disable ClockGenerator;
join
```

Expression

Une expression calcule une valeur à partir d'un ensemble d'opérateurs, de noms, de valeurs de littéraux et de sous-expressions

Syntaxe

```
Expression = either
   Primary
   Operator Primary              {unary operator}
   Expression Operator Expression    {binary operator}
   Expression ? Expression : Expression
   String

Primary = either
   Number
   Name              {of parameter, net, or register}
   Name[Expression]     {bit select}
   Name[Expression:Expression]     {part select}
   MemoryName[Expression]
   Expression,...      {concatenation}
   ExpressionExpression,...      {replication}
   FunctionCall
  (MinTypMaxExpression)  {MinTypMax expressions are used for delays}
```

Exemple

```
A + B
(A && B) || C
A[7:0]
-4'd12/3      // A large positive number
$realtobits(r);       // System function call
{A, B, C[1:6]}       // Concatenation (8 bits)
1:2:3      // MinTypMax
```

For

Déclaration de boucle à usage général. Permet à une ou plusieurs instructions d'être exécuté itérativement.

Syntaxe

```
for (RegAssignment;      {initial assignment}
       Expression;       {loop condition}
       RegAssignment)       {iteration assignment}
   Statement

RegAssignment = RegisterLValue = Expression
RegisterLValue = {either}
   RegisterName
   RegisterName[Expression]
   RegisterName[ConstantExpression:ConstantExpression]
   Memory[Expression]
   {RegisterLValue,...}
```

Exemple

```
V = 0;
for ( I = 0; I < 4; I = I + 1 )
begin
   F[I] = A[I] & B[3-I];     // 4 separate and gates
   V = V ^ A[I];      // 4 cascaded xor gates
end
```

Force

Semblable à une affectation continue procédurale, cause un changement d'état pour les fils et les registres. Cette clause est utilisée pour faciliter le débogage.

Syntaxe

```
{either}
force NetLValue = Expression ;
force RegisterLValue = Expression ;

{either}
release NetLValue ;
release RegisterLValue ;
```

Exemple

```
force f = a && b;
........
release f
```

Forever

Provoque l'exécution d'une ou de plusieurs instructions dans une boucle infinie.

Syntaxe

```
forever
       Statement
```

Exemple

```
initial
begin : Clocking
   Clock = 0;
   forever
       #10 Clock = !Clock;
end

initial
begin : Stimulus
   ......
   disable Clocking;      // Stops the clock
end
```

Fork

Regroupe les instructions dans un bloc pour une exécution parallèle.

Syntaxe

```
fork [ : Label
[Declarations...]]
       Statement .... .
join

Declaration = {either} Register Parameter Event
```

Exemple

```
initial
fork : stimulus
   #20 Data = 8'hae;
   #40 Data = 8'hxx; // This is executed last
   Reset = 0; // This is executed first
   #10 Reset = 1;
join // Completes at time 40
```

Fork

Regroupe les instructions dans un bloc pour une exécution parallèle.

Syntaxe

```
fork [ : Label
[Declarations...]]
```

```
        Statement .... .
    join

    Declaration = {either} Register Parameter Event
```

Exemple

```
    initial
    fork : stimulus
       #20 Data = 8'hae;
       #40 Data = 8'hxx; // This is executed last
       Reset = 0; // This is executed first
       #10 Reset = 1;
    join // Completes at time 40
```

Function

Une fonction est déclarée à l'intérieur d'un module, et est généralement appelée seulement à partir de ce module.

Syntaxe

```
    function [RangeOrType] FunctionName;
        Declarations...
        Statement .... .
    endfunction

    RangeOrType = either Range integer time real realtime
    Range = [ConstantExpression:ConstantExpression]
    Declaration = either
        input [Range] Name,...;
        Register
        Parameter
        Event

    Declaration = {either} Register Parameter Event
```

Exemple

```
    function [7:0] ReverseBits;
       input [7:0] Byte;
       integer i;
       begin
           for (i = 0; i < 8; i = i + 1)
              ReverseBits[7-i] = Byte[i];
       end
    endfunction
```

Function Call

Appelle une fonction qui renvoie une valeur à utiliser dans une expression.

Syntaxe

```
    FunctionName ( Expression,... );
```

Exemple

```
    Byte = ReverseBits(Byte);
```

if

Une déclaration qui exécute l'une des deux déclarations ou blocs de déclarations dépend d'une expression de condition.

Syntaxe

```
    if (Expression)
        Statement
    [else
        Statement]
```

```
Declaration = {either} Register Parameter Event
```

Exemple

```
if (C1 && C2)
begin
   V = !V;
   W = 0;
   if (!C3)
       X = A;
   else if (!C4)
       X = B;
   else
       X = C;
end
```

Initial

Contient une instruction ou un bloc d'instructions qui n'est exécuté qu'une seule fois, à partir du début de la simulation.

Syntaxe

```
initial
    Statement
```

Exemple

```
reg Clock, Enable, Load, Reset;
reg [7:0] Data;
parameter HalfPeriod = 5;
initial
begin : ClockGenerator
       Clock = 0;
   forever
       #(HalfPeriod) Clock = !Clock;
end
initial
begin : ClockGenerator
       Load = 0;
       Enable = 0;
       Reset = 0;
   #20 Reset = 1;
   #100 Enable = 1;
   #100 Data = 8'haa;
       Load = 1;
   #10 Load = 0;
   #500 disable ClockGenerator;     // Stops clock generator
end
```

Module

Le module est l'unité de base de la hiérarchie dans Verilog. Les modules contiennent des déclarations et des descriptions fonctionnelles et représentent des composants matériels.

Syntaxe

```
{either}
module ModuleName [(Port,...)];
    ModuleItems...
endmodule
macromodule ModuleName [(Port,...)];
    ModuleItems...
endmodule

ModuleItem = {either}
    Declaration
```

```
        Defparam
        ContinuousAssignment
        Instance
        Specify
        Initial
        Always
        Instance

    Declaration = {either}
        Port
        Net
        Register
        Parameter
        Event
        Task
        Function
```

Exemple

```
 macromodule nand2 (f, a, b);
    output f;
    input a, b;
    nand (f, a, b);
endmodule

module PYTHAGORAS (X, Y, Z);
    input [63:0] X, Y;
    output [63:0] Z;
    parameter Epsilon = 1.0E-6;
    real RX, RY, X2Y2, A, B;
    always @(X or Y)
    begin
       RX = $bitstoreal(X);
       RY = $bitstoreal(Y);
       X2Y2 = (RX * RX) + (RY * RY);
       B = X2Y2;
       A = 0.0;
       while ((A - B) > Epsilon || (A - B) < -Epsilon)
       begin
           A = B;
           B = (A + X2Y2 / A) / 2.0;
       end
    end
    assign Z = $realtobits(A);
endmodule
```

Net

sont utilisés pour modéliser les connexions (fils et bus). La valeur d'un net est déterminée par les valeurs des pilotes du net. Les pilotes peuvent être des sorties d'instances gate, module, ou affectations continues.

Syntaxe

```
 {either}
NetType [Expansion] [Range] [Delay] NetName,...;
trireg [Expansion] [Strength] [Range] [Delay]
    NetName,...;

Net declaration with continuous assignment
NetType [Expansion] [Strength] [Range] [Delay]
    NetAssign,...;

NetAssign = NetName = Expression
NetType = {either}
    wire tri
    wor trior
```

```
        wand triand
        tri0
        tri1
        supply0
        supply1
    Expansion = {either} vectored scalared
        Range = [ConstantExpression:ConstantExpression]
```

Exemple

```
 wire Clock;
wire [7:0] Address;
tri1 [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
    g = a || b;      // Continuous assignments
```

Number

Un nombre entier ou réel. Les nombres entiers dans Verilog sont représentés par des bits, dont certains peuvent être inconnus (X) ou à haute impédance (Z).

Syntaxe

```
 {either} BinaryNumber OctalNumber DecimalNumber HexNumber RealNumber
BinaryNumber = [Size] BinaryBase BinaryDigit...
OctalNumber = [Size] OctalBase OctalDigit...
DecimalNumber = {either}
    [Sign] Digit...
    [Size] DecimalBase Digit...
HexNumber = [Size] HexBase HexDigit...
RealNumber = {either}
    [Sign] Digit... .Digit...
    [Sign] Digit...[.Digit...]e[Sign]Digit...
    [Sign] Digit...[.Digit...]E[Sign]Digit...
BinaryBase = {either} 'b 'B
OctalBase = {either} 'o 'O
DecimalBase = {either} 'd 'D
HexBase = {either} 'h 'H
Size = Digit...
Digit = {either} _ 0 1 2 3 4 5 6 7 8 9
BinaryDigit = {either} _ x X z Z ? 0 1
OctalDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7
HexDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7 8 9 a A
b B c C d D e E f F
UnsignedNumber = Digit...
```

Exemple

```
'Haf        // An unsized hex number
6'o67        // A 6 bit octal number
8'bx        // An 8 bit unknown number (8'bxxxx_xxxx)
4'bz1        // All but the lsb are Z (4'bzzz1)
```

Number

Un nombre entier ou réel. Les nombres entiers dans Verilog sont représentés par des bits, dont certains peuvent être inconnus (X) ou à haute impédance (Z).

Syntaxe

```
 {either} BinaryNumber OctalNumber DecimalNumber HexNumber RealNumber
BinaryNumber = [Size] BinaryBase BinaryDigit...
OctalNumber = [Size] OctalBase OctalDigit...
    NetName,...;
```

```
Net declaration with continuous assignment
NetType [Expansion] [Strength] [Range] [Delay]
    NetAssign,...;

NetAssign = NetName = Expression
NetType = either
    wire tri
    wor trior
    wand triand
    tri0
    tri1
    supply0
    supply1
Expansion = either vectored scalared
    Range = [ConstantExpression:ConstantExpression]
```

Exemple

```
wire Clock;
wire [7:0] Address;
tri1 [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
    g = a || b;      // Continuous assignments
```

Parameter

Les paramètres sont un moyen de donner des noms à des valeurs constantes. Les valeurs des paramètres peuvent être remplacés lors de la compilation mais pas pendant la simulation.

Syntaxe

```
parameter Name = ConstantExpression,
    Name = ConstantExpression,
    ......;

or
parameter [Range] Name = ConstantExpression,
    Name = ConstantExpression,
    ......;
Range = [ConstantExpression:ConstantExpression]
```

Exemple

```
module Shifter (Clock, In, Out, Load, Data);
    parameter NBits = 8;
    input Clock, In, Load;
    input [NBits-1:0] Data;
    output Out;
    always @(posedge Clock)
        if (Load)
           ShiftReg <= Data;
        else
           ShiftReg <= ShiftReg[NBits-2:0], In
    assign Out = ShiftReg[NBits-1];
endmodule
```

Port

Les ports de modules modélisent les broches ou les connecteurs de bord du composant matériel.

Syntaxe

```
{definition}

{either}
PortExpression          {ordered list}
```

```
.PortName([PortExpression])          {named list}

PortExpression = {either}
    PortReference
    {PortReference,...}

PortReference = {either}
    Name
    Name[ConstantExpression]
    Name[ConstantExpression:ConstantExpression]

{declaration}
{either}
input [Range] Name,...;         of port reference
output [Range] Name,...;         of port reference
inout [Range] Name,...;         of port reference
Range = [ConstantExpression:ConstantExpression]
```

Exemple

```
module (A, B[1], C[1:2]);
    input A;
    input [1:1] B;
    output [1:2] C;

module (.A(X), .B(Y[1]), .C(Z[1:2]);
    input X;
    input [1:1] Y;
    output [1:2] Z;
```

Procedural Assignment

Modifie la valeur des registres ou planifie un futur changement.

Syntaxe

```
{Blocking assignment}
RegisterLValue = [TimingControl] Expression ;

{Non-blocking assignment}
RegisterLValue <= [TimingControl] Expression ;

RegisterLValue = {either}
    RegisterName
    RegisterName[Expression]
    RegisterName[ConstantExpression:ConstantExpression]
    Memory[Expression]
    {RegisterLValue,...}
```

Exemple

```
always @Swap
fork           // Swap the values of a and b
    a = #5 b;
    b = #5 a;
join           // Completes after a delay of 5

always @(posedge Clock)
begin
    a <= b;
    b <= a;           // Uses the 'old' value of 'b'
end

always @(posedge Clock)
    Count <= #1 Count + 1;


initial
begin
```

```
        Reset = repeat(5) @(negedge Clock) 1;
        Reset = @(negedge Clock) 0;
    end
```

Procedural Continuous Assignment

Une affectation continue procédurale, lorsqu'elle est active, attribue des valeurs à un ou plusieurs registres, et empêche les affectations procédurales ordinaires d'affecter les valeurs du registe assigné.

Syntaxe

```
 assign RegisterLValue = Expression ;
deassign RegisterLValue ; ;

{Non-blocking assignment}
RegisterLValue <= [TimingControl] Expression ;

RegisterLValue = {either}
    RegisterName
    RegisterName[Expression]
    RegisterName[ConstantExpression:ConstantExpression]
    Memory[Expression]
    {RegisterLValue,...}
```

Exemple

```
 always @(posedge Clock)
    Count = Count + 1;

always @(Reset)           // Asynchronous Reset
    if (Reset)
      assign Count = 0;  // Prevents counting, until Reset goes low
    else
      deassign Count;   // Resume counting on next posedge Clock
```

Bibliographie

[ACE] ACE. *Associated Compiler Experts ACE : CoSy compiler platform. Disponible en ligne.* www.ace.nl. (Cité en pages 59 et 63.)

[Actel] Actel. *CoreFFT Fast Fourier Transform.* Disponible en ligne : http ://www.actel.com. (Cité en page 69.)

[Aditya 2008] S. Aditya et V. Kathail. *Algorithmic Synthesis Using PICO. In : P. Coussy and A. Morawiec, editors. High-Level Synthesis From Algorithm to Digital Circuit.* Springer Science + Business Media B.V., pages 53–12, 2008. (Cité en pages vii, 21 et 25.)

[Allen 1987] J.R. Allen et K. Kennedy. *Automatic translations of Fortran programs to vector form.* ACM Toplas, no. 9, pages 491–542, 1987. (Cité en page 39.)

[Altera a] Altera. *Altera SDK for OpenCL : Programming Guide.* Disponible en ligne : www.Altera.com. (Cité en page 60.)

[Altera b] Altera. *FIR compiler user guide.* Disponible en ligne : www.Altera.com. (Cité en page 69.)

[Bae 2013] H. Bae, D. Mustafa J.W. Lee, Aurangzeb et al. *The Cetus Source-to-Source Compiler infrastructure : Overview and Evaluation.* International journal of parallel programming, Springer, vol. 41, pages 753–767, 2013. (Cité en page 70.)

[Blume 1996] W. Blume, R. Doallo, R. Eigenmann, J Grout J et al. *Parallel programming with Polaris.* Computer, IEEE, vol. 29, pages 78–82, 1996. (Cité en page 70.)

[Boulet 1994] P. Boulet, A. Darte, T. Risset et Y. Robert. *(pen)-ultimate tiling ?* Integration, the VLSI Journal, vol. 17, no. 1, pages 33–51, 1994. (Cité en page 42.)

[Campanoni 2012] S. Campanoni, T.M. Jones, G. Holloway, G.Y. Wei et al. *Helix : Making the extraction of thread-level parallelism mainstream.* IEEE Micro, vol. 32, pages 8–18, 2012. (Cité en page 70.)

[Canis 2011a] A. Canis, J. Choi, M. Aldham, V. Zhang et al. *LegUp : An open-source high-level synthesis tool for FPGA-based processor/accelerator systems.* ACM Transactions on Embedded Computing Systems, vol. 13, no. 2, pages 1–25, 2011. (Cité en pages vii et 57.)

[Canis 2011b] A. Canis, J. Choi, M. Aldham, V. Zhang et al. *LegUp : High-Level Synthesis for FPGA-Based Processor/Accelerator Systems.* In FPGA' ll Proceedings of the 19th ACM/SIGDA

international symposium on Field programmable gate arrays, pages 33–36. ACM, 2011. (Cité en pages vii, 56 et 57.)

[Canis 2013] A. Canis, J. Choi, B. Fort, R. Lianand et al. *From software to accelerators with LegUp high-level synthesis.* In proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) 2013, pages 1–9. IEEE, 2013. (Cité en page 57.)

[Chen 2004] T. Chen, J. Lin, X. Dai, W.C. Hsu et al. *Data dependence profiling for speculative optimizations.* In proceedings of the 24 Conference on Compiler Construction, CC 2004, pages 57–72. Springer, 2004. (Cité en page 62.)

[Cong 2006] J. Cong, Y. Fan, G. Han, W. Jiang et al. *Platform-Based Behavior-Level and System-Level Synthesis.* In Proceedings of 2006 IEEE International SoC conference l, pages 199–202. IEEE, 2006. (Cité en pages 15 et 59.)

[Cong 2011] J. Cong, B. Liu, S. Neuendorffer, J. Noguera et al. *High-Level Synthesis for FPGAs : From Prototyping to Deployment.* IEEE Transactions on computer-aided design of integrated circuits and systems, vol. 30, no. 4, pages 473–491, 2011. (Cité en pages 15 et 59.)

[Cong 2015] J. Cong et B. Reagen. *Tutorial : Rapid Exploration of Accelerator-rich Architectures, Automation from Concept to Prototyping.* In The 42nd International Symposium on Computer Architecture (ISCA), 2015. (Cité en pages vii et 16.)

[Coussy 2010] P. Coussy, G. Lhairech-Lebreton, D. Heller et E. Martin. *GAUT : A Free and Open Source High-Level Synthesis Tool.* 2010. (Cité en pages vii et 28.)

[Coussy 2011] P. Coussy, D. Heller et C. Chavet. *High-Level Synthesis : On the path to ESL design.* In Proceedings of ASIC (ASICON), 2011 IEEE 9th International Conference, pages 25–28. IEEE, 2011. (Cité en page 26.)

[Darte 1994] A. Darte et Y. Robert. *Mapping uniform loop nests onto distributed memory architectures.* Parallel Computing, vol. 20, no. 5, pages 679–710, 1994. (Cité en page 45.)

[Darte 1996a] A. Darte et F. Vivien. *On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops.* Journal of Parallel Algorithms and Applications, vol. 12, no. 3, pages 83–112, 1996. (Cité en page 41.)

[Darte 1996b] A. Darte et F. Vivien. *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs.* In Proceedings of PACT 96, Boston, MA, October 1996. IEEE Computer Society Press, 1996. (Cité en page 45.)

[Debbi 2016] A. E. Debbi. *Dependencies data flow graph based approach for speeding-up application.* In Industrial Informatics and Computer Systems (CIICS), 2016 International Conference, pages 1–6. IEEE, 2016. (Cité en pages vii, viii, 56, 63, 64, 67, 68 et 69.)

[Debbi 2017] A. E. Debbi. *Data-Flow Programming Paradigm for High Level Synthesis Improvement.* Journal of computers, vol. 13, no. 6, pages 622–637, 2017. (Cité en pages 59, 63 et 65.)

[Debbi 2018] A. E. Debbi. *Incremental Banerjee test conditions committing for robust parallelization frmework.* Turkish journal of electrical engineering and computer sciences, vol. 13, no. 4, pages 1–10, 2018. (Cité en pages vii, viii, 65, 66, 71 et 72.)

[Enas 2010] D. Enas et S. Thomas. *FPGA Implementation of Pipelined 2D-DCT and Quantization Architecture for JPEG Image Compression.* In proceedings of the IEEE Information technology Symposium (ITSIM),), pages 1–6. IEEE, 2010. (Cité en page 69.)

[Fakhari 2010] A. Fakhari et M. Fathy. *A Two Level Architecture for High Throughput DCT-Processor and Implementing on FPGA.* In proceedings of the international conference on Reconfigurable Computing and FPGAs (ReConFig), pages 115–120. IEEE, 2010. (Cité en page 69.)

[Feautrier 1992] P. Feautrier. *Some efficient solutions to the affine scheduling problem.* International journal of parallel programming, vol. 21, no. 6, pages 389–420, 1992. (Cité en page 49.)

[Guo 2005] Z. Guo, B. Buyukkurt, W. Najjar, K. Vissers et al. *Optimized Generation of Data-path from C Codes for FPGAs.* In proceedings of Design, Automation and Test in Europe, 2005, pages 112–117. IEEE, 2005. (Cité en page 57.)

[Gupta 2008] R. Gupta et F. Brewer. *High-Level Synthesis : A Retrospective. In : P. Coussy and A. Morawiec, editors. High-Level Synthesis From Algorithm to Digital Circuit.* Springer Science + Business Media B.V., pages 13–28, 2008. (Cité en pages vii et 8.)

[Hara 2008] Y. Hara, H. Tomiyama, S. Honda, H. Takada et al. *CHStone : A Benchmark program suite for practical C-based high-level synthesis.* In proceedings of the International Symposium on Circuits and Systems (ISCAS 2008), pages 1192–1195. IEEE, 2008. (Cité en page 68.)

[Huan 2008] L. Huan, P. Wei et L. Shui-sheng. *A new fabric of reconfigurable FFT processor for high-speed and low-cost system.* In proceedings of the International conference on Machine Learning and Cybernetics, pages 3525–3529. IEEE, 2008. (Cité en page 69.)

[Kathail 2002] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau et al. *PICO : automatically designing custom computers.* Computer, vol. 35, no. 9, pages 39–47, 2002. (Cité en pages 18 et 24.)

[Khronos a] Khronos. *OpenCL Reference Pages.* Disponible en ligne : www.khronos.org. (Cité en page 60.)

[Khronos b] Khronos. *The OpenCL Specification.* Disponible en ligne : www.khronos.org. (Cité en page 60.)

[Kim 2010] M. Kim, H. Kim et C. Luk. *SD3 : A scalable approach to dynamic data-dependence profiling.* In proceedings of 43rd Annual 26 IEEE/ACM International symposium on micro-architecture (MICRO), pages 535–546. IEEE/ACM, 2010. (Cité en pages 62 et 63.)

[Li 1992] Z. Li. *Array privatization for parallel execution of loops.* In proceedings of the 6th International Conference on Supercomputing ;, pages 313–322. ACM, 1992. (Cité en page 70.)

[Li 2015] Z. Li, A. Jannesari et F. Wolf. *An efficient data-dependence profiler for sequential and parallel programs.* In proceedings of Parallel and Distributed Processing Symposium, (IPDPS)), pages 484–493. IEEE, 2015. (Cité en pages 62 et 63.)

[LLVM] LLVM. *The LLVM Compiler Infrastructure Project. Disponible en ligne.* https ://llvm.org. (Cité en pages 58 et 63.)

[Martin 1993] E. Martin, O. Sentieys, H. Dubois et J. L. Philippe. *GAUT : An architectural synthesis tool for dedicated signal processors.* In Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference, pages 14–19. IEEE, 1993. (Cité en page 26.)

[Mentor] Mentor. *Disponible en ligne.* www.mentor.com. (Cité en pages 8 et 9.)

[Najjar 2003] W. Najjar, W. Bohm, B. Draper, J. Hammes et al. *From Algorithms to Hardware - A High-Level Language Abstraction for Reconfigurable Computing.* Computer, vol. 36, no. 8, pages 63–69, 2003. (Cité en page 57.)

[OpenCores a] OpenCores. *dct_ _idct.* Disponible en ligne : http ://opencores.org. (Cité en page 69.)

[OpenCores b] OpenCores. *pipelined_fft_128.* Disponible en ligne : http ://opencores.org. (Cité en page 69.)

[Oruklu 2012] E. Oruklu, R. Hanley, S. Aslan, C. Desmouliers et al. *System-on-Chip Design Using High-Level Synthesis Tools.* Circuits and systems, vol. 3, no. 1, pages 1–9, 2012. (Cité en pages vii et 23.)

[Razfan 2012] N. Razfan, S. Vlad-Mihai, O. Bryan, M. Roel et al. *DWARV 2.0 : A CoSy-based C-to-VHDL Hardware Compiler*. In proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL 2012), pages 29–31. IEEE, 2012. (Cité en pages viii, 56, 58 et 68.)

[Sato 2012] Y. Sato, Y. Inoguchi et T. Nakamura. *Whole program data dependence profiling to unveil parallel regions in the dynamic execution*. In proceedings of the IEEE International Symposium on Workload Characterization, (IISWC), pages 69–80. IEEE, 2012. (Cité en pages 62 et 63.)

[Schreiber 2002] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail et al. *PICO-NPA : High-Level Synthesis of Nonprogrammable Hardware Accelerators*. Journal of VLSI signal processing systems for signal, image and video technology, vol. 31, no. 2, pages 127–142, 2002. (Cité en pages 18 et 24.)

[Tu 1993] P. Tu et D. Padua. *Automatic array privatization*. In International Workshop on Languages and Compilers for Parallel Computing, pages 500–521. Springer, 1993. (Cité en page 70.)

[Urard 2008] P. Urard, J Yi, H Kwon et A. Gouraud. *User Needs. In : P. Coussy and A. Morawiec, editors. High-Level Synthesis From Algorithm to Digital Circuit*. Springer Science + Business Media B.V., pages 1–12, 2008. (Cité en pages vii et 7.)

[Villarreal 2010] J. Villarreal, A. Park, R. Halstead W. Najjar et al. *Designing Modular Hardware Accelerators in C With ROCCC 2.0*. In proceedings of 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010, pages 127–134. IEEE, 2010. (Cité en page 57.)

[visengi] visengi. Disponible en ligne : http ://www.visengi.com. (Cité en page 69.)

[Wolf 1991] M. E. Wolf et M. S. Lam. *A loop transformation theory and an algorithm to maximize parallelism*. IEEE Trans. Parallel Distributed Systems, vol. 2, no. 4, pages 452–471, 1991. (Cité en pages 41 et 42.)

[Xilinx a] Xilinx. *Disponible en ligne*. www.xilinx.com. (Cité en pages 15, 56 et 59.)

[Xilinx b] Xilinx. *FIR compiler V7.2 logicCORE IP Product Guide*. Disponible en ligne : www.xilinx.com. (Cité en page 69.)

[Xilinx c] Xilinx. *logicCORE IP DFT Transform , V4. Product Guide*. Disponible en ligne : www.xilinx.com. (Cité en page 69.)

[Xilinx d] Xilinx. *Vivado Design Suite User Guide : High Level Synthesis*. Disponible en ligne : www.xilinx.com. (Cité en page 59.)

[Yousri 2011] O. Yousri, J. Maher et A. Alfalou. *Implementation techniques of high-order FFT into low-cost FPGA*. In proceedings of the IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS), pages 1–4. IEEE, 2011. (Cité en page 69.)

[Zhang 2009] X. Zhang, A. Navabi et S. Jagannathan. *Alchemist : A transparent dependence distance pro*

filing infrastructure. In proceedings of the 7th annual IEEE/ACM International Symposium on CGO'09 Code Generation and Optimization CGO, pages 47–58. IEEE/ACM, 2009. (Cité en page 62.)