Titre: Title:	Intégration dans un flot de conception système d'un outil de traduction assistée de code C pour la création de coprocesseurs matériels
Auteur: Author:	Arnaud Desaulty
Date:	2016
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Desaulty, A. (2016). Intégration dans un flot de conception système d'un outil de traduction assistée de code C pour la création de coprocesseurs matériels [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2285/

UNIVERSITÉ DE MONTRÉAL

INTÉGRATION DANS UN FLOT DE CONCEPTION SYSTÈME D'UN OUTIL DE TRADUCTION ASSISTÉE DE CODE C POUR LA CRÉATION DE COPROCESSEURS MATÉRIELS

ARNAUD DESAULTY

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

JUILLET 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

INTÉGRATION DANS UN FLOT DE CONCEPTION SYSTÈME D'UN OUTIL DE TRADUCTION ASSISTÉE DE CODE C POUR LA CRÉATION DE COPROCESSEURS MATÉRIELS

présenté par : DESAULTY Arnaud

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. BELTRAME Giovanni, Ph. D., président

M. BOIS Guy, Ph. D., membre et directeur de recherche

M. LANGLOIS J. M. Pierre, Ph. D., membre

REMERCIEMENTS

Je tiens tout d'abord à remercier ma famille et mes amis, qui malgré l'éloignement géographique, ont su me donner le courage et la motivation nécessaires.

Je remercie aussi tout particulièrement mon directeur de recherche M. Bois et son équipe dans les locaux de SpaceCodesign, pour la patience dont ils ont fait preuve et les conseils qu'ils m'ont prodigués.

Je remercie également mes collègues de laboratoire, Étienne et Mathieu, qui étaient toujours là pour me soutenir et pour d'incroyables discussions autour d'un café.

Enfin, je tiens à remercier l'intégralité des personnes que j'ai rencontrées ici au Québec, expatriés ou natifs. Votre gentillesse et votre style de vie resteront toujours ancrés dans ma mémoire. Merci tout particulièrement à mes colocataires, Gavin et Sebastian pour les bons moments passés ensemble.

RÉSUMÉ

Depuis les débuts de la conception de systèmes électroniques, un des buts de la recherche est de fournir des outils et méthodes de travail de plus en plus puissants et rapides. Ceci est illustré très explicitement dans le fossé qui sépare la croissance du nombre de transistors par puce de la croissance du nombre de transistors intégrés par ingénieur par mois. Au cours des années se sont alors développées des méthodes permettant à un concepteur d'abstraire de plus en plus son travail afin de faciliter l'intégration et la distribution de composantes matérielles. À ces abstractions de description viennent s'ajouter des outils automatisant certaines parties du travail d'un concepteur. C'est par exemple le cas de la synthèse haut-niveau, qui permet de synthétiser des algorithmes de manière guidée. Enfin, de nombreux flots de travail ont aussi vu le jour afin de fournir une méthodologie dans la conception de systèmes électroniques. Le flot que nous retiendrons est le flot niveau système électronique (ESL) qui décrit une méthode de développement incrémentale partant d'une description la plus abstraite possible. Les abstractions sont itérativement relâchées pour finalement arriver à une description matérielle.

Toutefois, il n'existe pas d'outil complet intégrant toutes les étapes du flot ESL. Nous proposons donc un outil de traduction assistée de code C afin de compléter un flot de conception ESL dans le cadre de la création de coprocesseurs matériels. L'outil proposé, C2Space, permet la transition d'un code C séquentiel vers un code organisé en modules SpaceStudio de manière rapide et configurable. Cet outil de traduction, basé sur le transpilateur Clang, procède à l'analyse statique du code pour effectuer les choix de traduction appropriés. C2Space permet l'intégration en amont du flot d'une solution d'analyse dynamique de code C, appelée Pareon Profile. Pareon permet une première analyse du code séquentiel afin de repérer rapidement les portions du code susceptibles de profiter d'un passage sur coprocesseur (de par leur potentiel de parallélisme et leur poids dans l'exécution séquentielle). Une fois la traduction effectuée, le reste du flot est supporté par l'environnement de codéveloppement SpaceStudio, qui permet alors de procéder à l'exploration architecturale du code traduit. Il est alors possible de tester différents schémas d'allocation des modules aux ressources matérielles (processeurs, coprocesseurs). Les modules alloués en matériel sont alors synthétisés à l'aide d'un logiciel de synthèse haut-niveau. Le système complet peut ensuite être exporté vers une des plateformes matérielles supportées (par exemple, pour FPGA Xilinx).

Nous avons confronté ce flot à un algorithme de détection de contours (filtre de Canny) afin d'en tester l'efficacité. Les résultats montrent que l'approche proposée permet un raffinement rapide de l'algorithme, de sa définition logique jusqu'à son implémentation. Nous n'avons pu accélérer, comme envisagé initialement, la vitesse d'exécution de l'algorithme, mais nous avançons plusieurs pistes pour expliquer ces résultats. Nous proposons une série de recommandations visant à améliorer à la fois C2Space et le flot proposé. Nous décrivons notamment l'utilisation d'une nouvelle métrique très tôt dans le flot de conception mettant en relation (1) le potentiel de parallélisme d'un segment de code, (2) la portion du temps d'exécution global de ce segment, (3) le coût de communications processeur/coprocesseur pour ce segment et (4) le temps d'exécution logiciel de cette même portion de code.

ABSTRACT

Since the beginning of Electronic System Design, one of the main goals pursued in research has been to provide smarter and faster tools and workflows. This is clearly shown in the gap that separates the number of transistor per chip and the number of transistor that can be integrated by an engineer in a month. Over the years, many methods allowing a designer to abstract more and more his design came to fruition. In addition to those methods, automatization of certain tedious and repetitive tasks in the design process appeared. For instance, the tools of High Level Synthesis allow a high level specification to be automatically translated into a working Register Transfer level design. Finally, numerous workflows arose to provide more definite framework in Electronic System Design. The workflow that we will tackle is the Electronic System Level Flow (ESL), which describe an incremental method that starts from the most abstract specification possible. Abstraction is then released in small increments until a fully hardware specification is obtained.

However, there is no comprehensive tool incorporating all stages of the ESL flow. We therefore propose an assisted translation tool for C code in order to complete an ESL design flow as part of creating hardware coprocessors. The proposed tool, C2Space, allows the transition of a sequential C code to a code organized in SpaceStudio modules in a fast and configurable way. This translation tool, based on the transpiler Clang, performs static analysis of the code to perform the appropriate translation choices. C2Space allows the integration, upstream of the flow, of a dynamic code analysis solution for C, called Pareon Profile. Pareon allows a first analysis of the sequential code to quickly identify sections of code that could benefit from a passage on coprocessor (by their potential parallelism and their weight in the sequential execution). Once the translation is done, the rest of the workflow is supported by the co-design environment SpaceStudio, which allows for architectural exploration of the translated code. It is then possible to test different mappings of the modules to the hardware resources (processors, coprocessors). Modules that are allocated material are then synthesized using a high-level synthesis software. The complete system can then be exported to one of the supported hardware platforms (e.g., for Xilinx FPGAs).

We compared this flow to an edge detection algorithm (Canny filter) to test its effectiveness. The results show that the proposed approach enables rapid refinement of the algorithm, from the logic definition to its implementation. We have not been able to accelerate, as originally envisioned, the execution speed of the algorithm but we are offering several possible explanations for these results.

We provide a series of recommendations to improve both C2Space and the proposed workflow. We describe in particular the use of a new metric early in the design flow linking (1) the potential parallelism of a code segment, (2) the portion of the total execution time of this segment, (3) the cost of communications between processor and coprocessors for this segment and (4) the software runtime of that same piece of code).

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES EXTRAITS DE CODE

LISTE DES SIGLES ET ABRÉVIATIONS

AADL	Architectural Analysis Description Langage
ALAP	As Late As Possible
ASAP	As Soon As Possible
AST	Abstract Syntax Tree
BRAM	Block Random Access Memory
BSP	Board Support Package
CFDG	Control-Flow Data Graph
CMU-DA	Carnegie Melon University Design Automation
DAM	Destinée À être Modularisée
DMA	Direct Memory Access
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EDK	Embedded Development Kit
ELF	Executable and Linkable Format
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
IP	Intellectual Property
JSON	JavaScript Object Notation
MCSE	Méthodologie de Conception des Systèmes Électroniques
PBD	Platform Based Design

RTL	Register Transfer Level
RTM	Register Transfer Modules
SoC	System on Chip
TLM	Transaction Level Model
VLSI	Very Large Scale Integration
XPS	Xilinx Platform Studio

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Le domaine du design de puces destinées à différents systèmes informatiques, comme les circuits intégrés pour application spécifique, les processeurs d'utilisation générale ou encore les processeurs de traitement de signal numérique, comporte de nombreux défis, tel qu'énoncé par le carnet de route international de la technologie des semi-conducteurs. Parmi ceux-ci, le problème des fossés entre la croissance de la productivité, la vérification des ingénieurs système et la quantité de composantes à intégrer dans une architecture reste un des grands efforts de recherche de ce champ d'applications.

La loi de Moore [1] énonce que la quantité de transistors intégrés dans les puces double tous les ans. Elle découle d'une observation de cette tendance entre les années 59 et 65. Cette prédiction s'est avérée correcte pour les dix années suivantes. Moore révisa sa loi lors des années 80, en

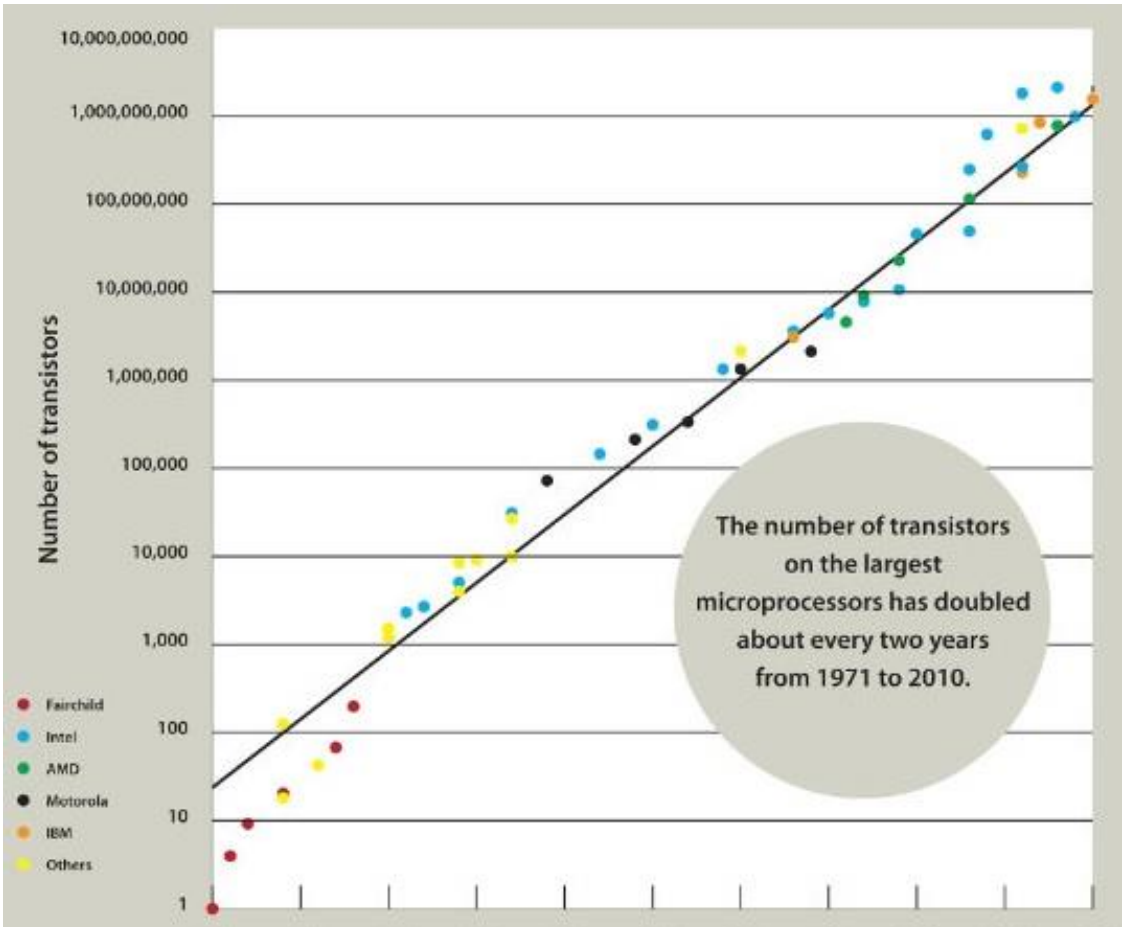


Figure 1.1-1 : Évolution de l'intégration de transistors sur puces de 1960 à 2010.
Courtoisie de Computer History Museum : Revolution Exhibition [1]

énonçant cette fois une multiplication par deux de la complexité des circuits tous les deux ans [2]. Depuis 1975, cette tendance s'est maintenue (fig. 1.1-1) et plusieurs experts soutiennent que la loi de Moore restera pertinente pour la prochaine décennie.

Toutefois, un des aspects non développés dans cette croissance de la complexité des circuits est la capacité dans un temps raisonnable de concevoir un système tirant parti de ces capacités grandissantes. Ce problème, qui émerge en 1997 dans un rapport de SEMATECH, stipule que la croissance énoncée dans la loi de Moore surpasse largement la croissance de la capacité d'intégration de transistors par ingénieur par mois et que, si ce fossé n'est pas comblé, l'industrie sera sévèrement bridée dans sa capacité d'intégration et dans ses temps de mise sur le marché.

Certaines recherches plus récentes estiment que ce fossé n'a jamais eu lieu grâce à la création d'outils de plus en plus efficaces mais commencent à s'inquiéter d'un autre problème de productivité lié à la vérification [3]. En effet, la croissance de la complexité des puces électroniques implique une augmentation des efforts de vérification pour mener un projet à terme. Ainsi en 2012, le temps alloué à la vérification d'un système représentait en moyenne 56 % du temps total d'un projet, alors qu'en 2007, ce temps n'était que de 49 %. La figure 1.1-2 résume ces constats.

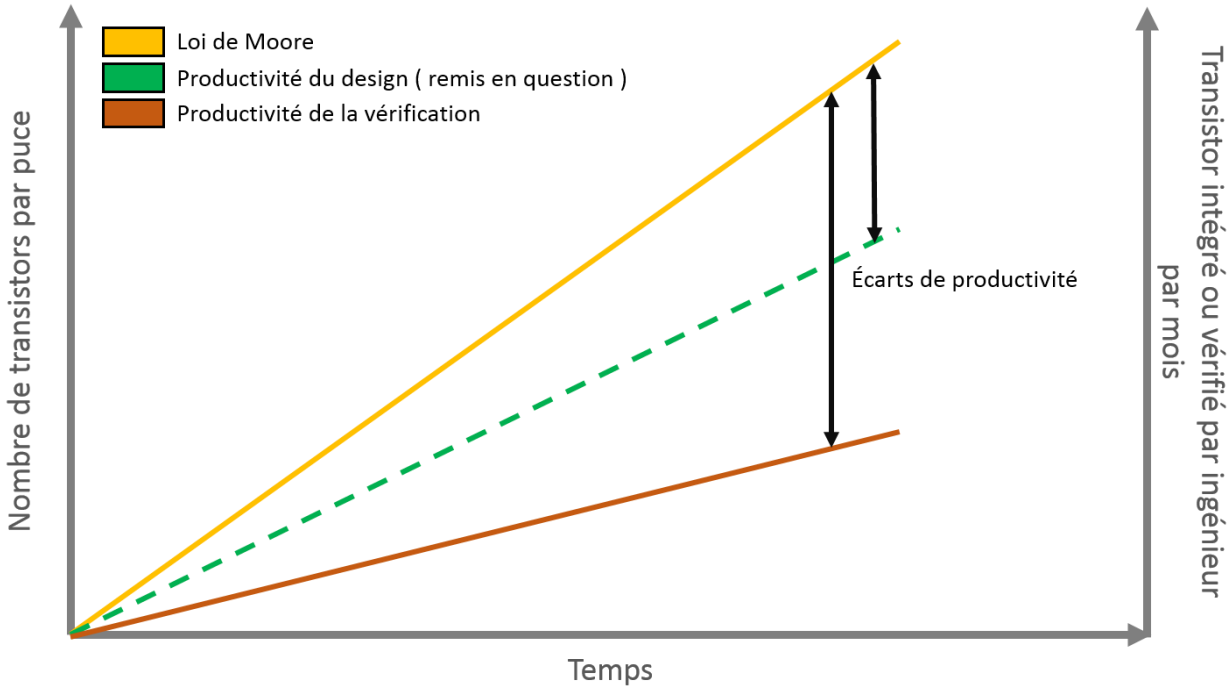


Figure 1.1-2 : Illustration des écarts de productivité avec la croissance du nombre de composants à disposition

Pour résoudre cette problématique, l'effort de recherche a abouti au cours du temps à plusieurs améliorations incrémentales du flot de développement des systèmes matériels. La suite de cette revue va résumer et classifier ces différentes améliorations.

1.1 Les langages de description de matériel (HDL)

Dès les années 80, avec la montée croissante de l'adoption de l'intégration à très grande échelle (VLSI) au détriment des puces spécifiques, ce n'est plus quelques centaines de composants qui doivent être interconnectés, mais bien plusieurs centaines de milliers. Le besoin d'une vue à plus haut niveau et de réutilisabilité dans de tels systèmes se fait alors sentir [4]. Les langages de description de matériel (HDL) commencent alors à apparaitre. Ces langages ont un but principal : la création d'un standard de communication dans le domaine de la description de systèmes matériels. Un tel standard permet la communication entre les différentes équipes, mais aussi la diffusion de modèles standards d'une entreprise à une autre. Ainsi, un langage de description doit être capable de capturer plusieurs aspects d'un design : sa fonctionnalité (aspect procédural), son architecture (aspect structurel) et la façon dont les données transitent (aspect flot de données) [5]. L'autre avantage d'une standardisation des représentations est de permettre le développement des outils d'automatisation du design électronique (EDA) basés sur ces standards. Aujourd'hui, deux HDLs dominent le marché : VHDL et Verilog HDL. Ces deux HDL possèdent à quelques différences près les mêmes capabilités techniques, comme la description haut-niveau de l'algorithme modélisé (procédural), la liste des composants bas-niveau nécessaire (structurel) ainsi que les éléments de synchronisation (flot de données). Ce qui motive le choix pour l'un ou l'autre de ces outils sera plutôt la préférence personnelle du développeur, l'environnement EDA qui est à sa disposition ou encore différentes raisons économiques [6].

1.2 La synthèse haut-niveau (HLS)

Durant la même période, un nouveau paradigme de conception de systèmes sur puce (SoC) commence à apparaitre. Les travaux préliminaires de Barbacci et Sieworek montrent qu'il est possible de convertir une description de niveau transfert de registre (RTL) comportementale en une implémentation structurelle composée de blocs spécifiques appelés modules niveau transfert registre (RTM) [7]. Ces premiers résultats donnent naissance à l'outil Carnegie-Mellon University Design Automation (CMU-DA) [8]. Celui-ci est décrit comme un flot de conception assistée par

ordinateur permettant d'accompagner la transformation d'une spécification haut-niveau de système complexe jusqu'à la création de masques qui seront utilisés pour la production de celui-ci. Ce flot prend en entrée un langage décrivant le comportement du système et fournit en sortie une description du point de vue de la RTL. Ces premiers résultats déclenchent un fort intérêt de la recherche dans le domaine de l'EDA et on voit apparaitre une première génération d'outils HLS, autant académiques qu'industriels. Toutefois, cette première vague d'outils a du mal à s'imposer et à devenir une norme dans le milieu de la conception de SoC, car les techniques de synthèse RTL sur lesquelles reposent les outils HLS de l'époque ne sont pas encore fonctionnelles et adoptées. Ces premiers outils sont en plus encore considérés comme trop simplistes et conduisent à une qualité de résultats peu encourageante [9].

C'est avec l'utilisation croissante des standards HDLs VHDL et Verilog que la nouvelle génération d'HLS trouve un second essor dans les années 90 [10]. De plus, la recherche permet aux outils de synthèse RTL de devenir plus performants, conduisant ainsi à l'adoption de ceux-ci par l'industrie. Dans [11], Camposano présente les aspects principaux de la synthèse haut niveau. Le flot présenté prend comme point de départ une description comportementale VHDL puis applique dans l'ordre trois transformations sur celle-ci. La première passe appelée « compilation » a pour but de transformer le code en une représentation mieux adaptée à la passe suivante, en l'occurrence, une paire Graphe du flot de contrôle/Graphe du flot de données. En ceci, cette première passe ressemble beaucoup à l'action qu'aurait un compilateur logiciel non-optimisant. La seconde passe de l'outil consiste à appliquer des transformations haut niveau sur la sortie de la première passe. Certaines de ces optimisations sont communes à celles qu'un compilateur aurait pu effectuer tandis que d'autres sont spécifiques au domaine matériel (par exemple le déroulement de boucle ou la création de concurrence). Ces modifications affectent la forme des deux structures précédemment créées (contrôle et données). Enfin, la troisième phase du flot consiste en l'ordonnancement et l'allocation des ressources aux éléments des graphes. Plusieurs algorithmes permettent d'arriver à différents résultats. On peut citer pour l'ordonnancement les algorithmes aussitôt que possible (ASAP) et aussi tard que possible (ALAP). Une fois un ordonnancement choisi, on passe à l'allocation des ressources matérielles. Cette étape est intimement liée à l'étape d'ordonnancement et peut être résolue avec, par exemple, des algorithmes de colorations de graphes. On notera que le résultat final de ce flot peut être très variable et nécessite encore l'intervention d'un designer pour effectuer des décisions. En effet, certains de ces algorithmes étant NP-complets, l'exploration de l'espace

entier de solution n'est pas possible et le designer est sollicité pour diriger l'exploration avec des contraintes.

Ces nouveaux outils HLS demandent toutefois une connaissance des langages HDL pour créer la première description comportementale à haut niveau, ce qui freine l'adoption de ceux-ci auprès de la communauté des ingénieurs systèmes et des ingénieurs logiciels, qui préfèrent décrire leurs algorithmes à l'aide de langages haut niveau comme C++ ou C.

C'est pourquoi, depuis les deux dernières décennies, une nouvelle génération d'outils voit le jour. Ces outils permettent le passage d'une description fonctionnelle dans un langage haut niveau comme C/C++, SystemC ou encore OpenCL vers une description RTL à l'aide d'un corpus de contraintes. On peut identifier plusieurs raisons à cette évolution du paradigme. La première est, comme cité plus haut, le besoin d'un langage facile à apprendre et peu différent des langages utilisés par les ingénieurs logiciels afin de permettre un gain de productivité ainsi que de faciliter l'adoption d'un tel outil par une entreprise. La seconde, et peut-être la plus importante, est l'apparition de processeurs de plus en plus puissants, créant un nouveau domaine d'applications dans la création de systèmes informatiques : le codéveloppement logiciel/matériel [12] (*Hardware/Software codesign*). L'apparition de nouveaux processeurs permettant d'exporter une grande partie des fonctionnalités d'un système du domaine matériel au domaine logiciel crée alors deux chemins de développement parallèles, la partition matérielle et la partition logicielle. Un besoin d'uniformité dans la manière de décrire les parties logicielles et matérielles se fait donc sentir et la synthèse haut niveau parait alors la solution de choix afin de partir d'une description fonctionnelle semblable pour tous les éléments (qu'ils soient matériels ou logiciels) pour ensuite développer automatiquement les descriptions matérielles des éléments estimés critiques.

1.3 Le niveau « système électronique » (ESL)

le terme ESL apparait aux alentours des années 2000 [13] et est défini de nombreuses manières, mais chacune de ces définitions gravite autour de la même idée : une méthodologie de développement de systèmes électroniques à haut niveau d'abstraction. Les auteurs proposent une définition :

> L'utilisation appropriée d'abstraction dans le but d'augmenter la compréhension d'un système, et d'améliorer les chances d'une implémentation réussie des fonctionnalités de manière rentable, tout en respectant les contraintes nécessaires. (Martin, Bailey et Piziali, 2010, p.3)

Ainsi, l'ESL est une méthodologie qui formalise la création de systèmes électroniques en prenant en compte les différents enjeux du projet, comme représenté sur la figure 1-3.1. De plus, cette définition implique que le système en question pourra être composé d'éléments logiciels comme matériels (par le biais de l'utilisation du terme « fonctionnalité » qui ne fait pas de postulats sur le type d'implémentation). Enfin, cette définition met en évidence le besoin d'abstraction pour parvenir à ces objectifs.

Figure 1.3-1 : Les enjeux d'un projet de création de systèmes électroniques

Le but de cette approche est donc de permettre d'accélérer le processus de développement d'un système électronique en travaillant à haut niveau d'abstraction afin de pouvoir valider les étapes de développement plus rapidement et de procéder à des vérifications incrémentales des fonctionnalités. Le flot ESL peut être décomposé en six grandes étapes résumées dans la figure 1.3-2. La première étape traite de la création des spécifications nécessaires à la création du système

électronique. Durant cette étape, les spécifications pourront être écrites dans un langage naturel (description, cahier des charges, etc.) ou dans un langage exécutable sous forme de boîtes opaques. Pour cette étape, une des motivations principales est l'abstraction. En effet, il est important de maintenir un niveau d'abstraction très haut afin de ne pas supposer l'implémentation, d'où l'utilisation d'un modèle descriptif ou boîte noire. Il faut bien comprendre qu'une segmentation du système en sous-systèmes est déjà à l'œuvre dans cette première phase. Il est important de garder à l'esprit ce constat, car si l'équipe de développement souhaite par la suite réutiliser des blocs déjà construits ou bien des Propriétés Intellectuelles (IP), il doit prévoir d'en prendre compte dans sa phase de spécifications.

La seconde phase, l'analyse de prépartitionnement, consiste en une première exploration de l'espace de solutions. Cette exploration est purement algorithmique et va permettre de dresser un tableau préliminaire des compromis possibles entre les différents domaines d'exploration de la conception de systèmes embarqués : temps, taille, puissance, complexité et temps de mise sur le marché. En effet, certains algorithmes sont par nature plus parallèles que d'autres, tandis que certains consomment plus de mémoire pour mener à bien leurs opérations. Afin d'analyser ces compromis, plusieurs outils peuvent être utilisés, notamment des outils d'analyse dynamique comme la bibliothèque Valgrind [14], qui permet d'obtenir des informations sur l'exécution de la spécification comme le nombre d'invocations de chaque fonction, le nombre de fils d'exécution, les dépendances ainsi que les accès en lecture/écriture et le nombre de fautes de caches. Ces informations sont ensuite utilisées pour profiler l'algorithme sélectionné. Des outils d'analyse statique peuvent aussi être utilisés comme l'analyseur ASTRÉE [15], qui permet d'assurer la validité d'un algorithme dans tous les cas d'utilisation à la compilation, et ce, pour des programmes complexes. À l'issue de cette phase, un premier niveau d'abstraction de l'implémentation est levé par le choix des algorithmes.

La troisième phase, le partitionnement, est une des phases les plus critiques du flot ESL. En effet, c'est lors de cette phase que nous décidons comment allouer les ressources matérielles disponibles (processeur, coprocesseur, FPGA) aux différentes parties de la spécification. Cette phase peut être décomposée en sous-phases :

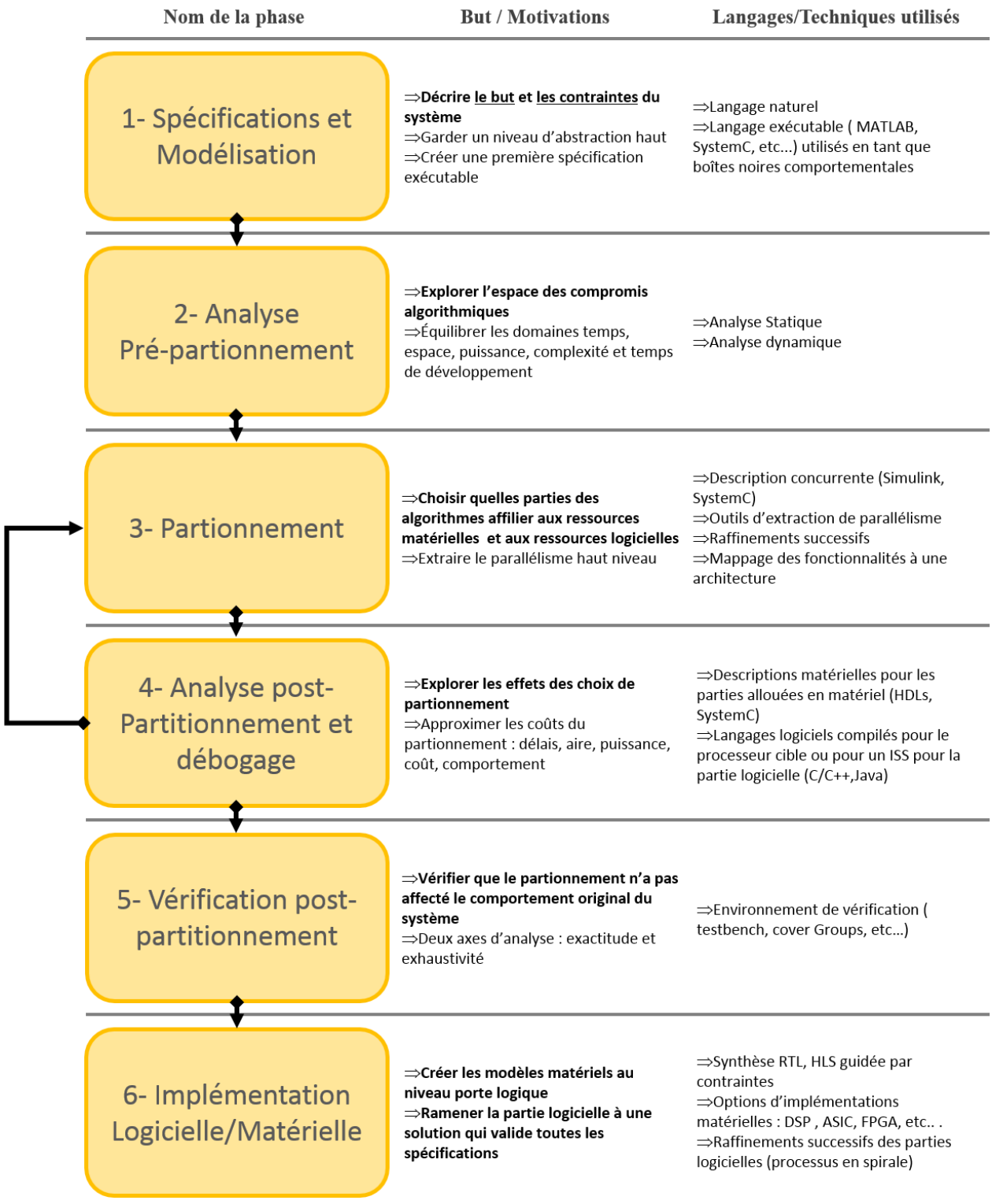


Figure 1.3-2 : Flot ESL

- Décomposition des fonctions : Cette première étape vise à décomposer les fonctionnalités en blocs afin de faire apparaitre un parallélisme d'application. Une des méthodes pour

parvenir à ce résultat est l'utilisation d'outils automatiques d'extraction de parallélisme à partir d'un code séquentiel. Ces outils doivent être en mesure d'exposer les dépendances de données afin de proposer des choix de parallélisation cohérents. C'est le cas de Pareon [16], qui est un outil d'analyse dynamique permettant d'exposer le parallélisme de boucle dans un contexte d'applications multitâches (*multithreaded*).

- Description de l'architecture : Afin de pouvoir effectuer une allocation à haut niveau d'abstraction, il est nécessaire d'avoir une description de l'architecture ciblée sous forme de blocs. Ces blocs représentent des éléments haut niveau du système (FPGA, GPU, DSP, bus, mémoire). Le langage de description et d'analyse architecturale (AADL) permet la modélisation de ce genre de système à plus ou moins haut niveau d'abstraction et peut être utilisé en tant que cadre de travail pour des méthodologies du haut vers le bas (*top-down*) [17].

- Partitionnement : Une fois les fonctions décomposées et l'architecture définie à haut niveau, on peut passer à l'allocation des fonctions à des éléments de l'architecture. Cette étape peut être appréhendée de deux manières. La première est d'utiliser un langage permettant un raffinement itératif de la solution fonctionnelle vers la solution finale. Le raffinement aura pour but de préciser les minutages de communication et de calcul. Le langage SystemC illustre parfaitement cette catégorie de solutions avec son système de modules et d'interfaces de communications permettant de créer un modèle fonctionnel sans traces de minutage, puis de le raffiner élément par élément tout en surveillant le respect de la fonctionnalité. La deuxième manière est d'utiliser un langage ou une plateforme permettant le mappage des fonctionnalités aux éléments architecturaux de manière explicite. Par exemple, Space Studio [18] est une plateforme utilisant un sous-ensemble de SystemC afin de créer une description des fonctionnalités sous forme de modules. Ces modules communiquent ensuite à l'aide de simples fonctions d'envois et de réceptions et peuvent être associés à une architecture matérielle définie par l'utilisateur (comprenant FPGA et GPU). La gestion des communications est entièrement prise en charge par l'environnement et les minutages peuvent être raffinés à l'aide de fonctions permettant d'ajouter des délais d'exécution aux modules matériels.

Au cours de la phase d'Analyse Post-Partitionnement, les modèles développés au cours de l'étape précédente sont raffinés pour coïncider avec les choix de partitionnement. Ainsi, les éléments associés à la partie logicielle sont compilés pour l'architecture du processeur ou bien sont exécutés sur des machines virtuelles émulant le comportement du processeur cible, par exemple QEMU [19], une machine virtuelle qui permet de faire fonctionner différents systèmes d'exploitation depuis un autre système d'exploitation (Windows, Linux). De plus, QEMU permet d'émuler sur sa machine virtuelle plusieurs architectures de processeurs différentes (x86, ARM, SPARC). La partie matérielle est décrite dans un langage de description matérielle (comme SystemC, VHDL ou encore System Verilog). Cette description peut être effectuée à la main par des ingénieurs ou bien automatisée dans une certaine mesure par des outils HLS. Une fois le raffinement effectué, il est possible de procéder à l'analyse proprement dite. On va chercher à préciser les approximations faites lors de la phase deux grâce aux informations supplémentaires apportées par le partitionnement. Des outils d'analyse du système modelé doivent être utilisés pour déterminer les performances des modules matériels, des modules logiciels, et des interfaces de communications les liant. Comme dans la seconde phase, les préoccupations majeures de cette analyse sont les aspects temps, taille, puissance, complexité. Cette phase peut être répétée conjointement avec la phase trois jusqu'à obtenir une solution acceptable. Ce cycle est appelé Exploration de l'espace de design ou exploration architecturale.

Le but de la cinquième phase, la vérification post-partitionnement, est de s'assurer que les spécifications comportementales créées lors de la première phase sont encore respectées à la suite du partitionnement de la solution. Afin de pouvoir créer une routine de vérification qui sera valide du plus haut niveau d'abstraction jusqu'au plus bas, il est nécessaire de planifier celle-ci sous la forme d'un plan de vérification. Ce plan doit permettre, au cours de raffinements successifs, de passer d'un ensemble de contraintes données par la spécification à un ensemble de scénarios de test de couverture. Une fois le plan de vérification effectué, on utilise un environnement de test (formel ou dynamique). System Verilog [20] est un bon exemple d'environnement de test dynamique. Il permet la création de bancs d'essai qui fournissent des stimuli au design testé. La réponse à ces stimuli est enregistrée dans un tableau des scores. Il est de plus possible de définir des assertions, conditions qui doivent rester valides tout au long de l'exécution du code. Ces résultats sont ensuite analysés par les ingénieurs pour s'assurer que le code a été testé de la manière la plus complète possible ainsi que pour s'assurer qu'aucun cas ne cause d'erreur.

Lors de la dernière phase, l'implémentation matérielle/logicielle, les éléments matériels doivent être définis jusqu'au niveau porte logique afin de pouvoir réaliser les masques pour la gravure des puces dans le cas d'un circuit intégré pour application spécifique (ASIC). Pour les autres supports matériels, d'autres types de description peuvent être nécessaires (comme un *bitstream*, dans le cas d'une implémentation sur FPGA). Les processus d'automatisation du design électronique permettent de s'exempter de la définition au niveau porte logique en utilisant des outils de synthèse niveau transfert registre (RTL synthesis) ou bien en synthétisant directement des modèles comportementaux à l'aide des techniques de synthèse haut niveau. La phase d'implémentation logicielle quant à elle, a pour but de générer tous les artefacts logiciels nécessaires au bon fonctionnement du système. Dans le cadre du flot ESL complet, une grande partie de ce travail a déjà été effectuée avec les raffinements successifs de la partie algorithmique. Il ne reste qu'à créer la logique d'interfaçage avec le reste des éléments du système et l'utilisateur. Une démarche classique de développement logiciel peut être utilisée (flot en cascade ou en spirale).

Il est important de prendre en considération que le flot présenté ici représente le flot idéal du développement ESL. Dans la réalité, les solutions qui proposent un flot ESL peuvent n'implémenter qu'une partie du flot discuté ici. De plus, l'ordre du flot peut être altéré par plusieurs facteurs, comme, par exemple, l'utilisation d'une plateforme prédéfinie par les spécifications ou bien l'obligation d'utiliser une IP particulière. Enfin, dans ce flot, chacune des étapes est censée se succéder (excepté pour les étapes 3 et 4 qui peuvent s'alterner), mais dans la réalité, il est parfois nécessaire de reprendre une étape précédente.

Par exemple, l'outil présenté dans [21] remplit entièrement les fonctionnalités attendues de l'étape quatre de la méthodologie ESL présentée ci-dessus. Cet outil basé sur une approche de design axé plateforme (PBD), permet de faire des estimations de minutages et de puissance à haut niveau d'abstraction d'un système hétérogène complet. Les éléments fonctionnels sont spécifiés sous forme de code SystemC et interconnectés par le standard TLM2 (modèle au niveau transactionnel). Une fois un partitionnement de la spécification effectué, chaque élément, selon qu'il soit matériel, logiciel ou bien privé (IP) fait l'objet d'une analyse dynamique qui permet une rétroannotation dans le code fonctionnel des informations de minutage et de puissance. Le code destiné au logiciel est compilé pour le processeur cible puis fait l'objet d'une analyse bas niveau des délais et performance. Chacune des instructions assembleur est ensuite associée aux lignes du code source et une annotation est ajoutée dans le code SystemC afin de simuler avec précision ces données dans

le cadre du système complet. Le code matériel personnalisé est quant à lui analysé à l'aide de techniques de synthèse haut niveau. Le code source est transformé en graphe de flot-contrôle de données (CFDG), puis subit les transformations habituelles d'un flot HLS (ordonnancement, allocation et mappage) afin de donner une description RTL. Puisque la phase HLS a modifié en profondeur l'agencement du code source, il n'est pas possible de rétroannoter le code source, mais une version C++ du code RTL avec les informations de minutage et de puissance est générée et vient remplacer le code source lors de la simulation. Enfin, les blocs privés étant des boîtes noires, il n'est pas possible d'accéder directement aux informations de puissance (les informations de minutage étant généralement fournies par le constructeur). Pour remédier à cela, les auteurs créent un moniteur surveillant les transactions de l'IP avec le reste du système et génèrent à partir de sa documentation une machine de puissance à états. Chaque état correspond à un scénario d'utilisation de l'IP dont la consommation est connue. Les changements d'état sont dictés par le moniteur et il devient ainsi possible de tracer un graphe de puissance au cours du temps de simulation. Une fois toutes ces analyses terminées, l'outil se montre capable d'exécuter le système virtuel annoté pour obtenir des approximations de puissance et de minutage avec une erreur de 4 % par rapport à la simulation au niveau porte logique avec un facteur de simulation de moins de 200 (simulation 200 fois plus lente que l'exécution du même système réel).

Dans l'article [22], il est présenté une méthodologie qui couvre les phases trois à six du flot ESL. Cette méthodologie tire parti des solutions HLS proposées par Xilinx et permet un prototypage rapide de système embarqué afin de répondre aux contraintes de productivité et de performance. En utilisant les outils Vivado HLS et l'environnement de développement de systèmes embarqués Xilinx EDK, le flot proposé permet d'amener une solution basée sur le langage C jusqu'à une implémentation sur FPGA. Pour faire un parallèle avec le flot ESL, les phases trois et quatre sont assurées par association des éléments partitionnés sur une architecture connue, grâce à l'interface graphique Xilinx Platform Studio, qui permet de visualiser une description matérielle de la plateforme choisie. Les éléments logiciels sont développés en C dans l'environnement de développement logiciel de Xilinx, EDK, et sont ensuite importés sous forme de fichiers ELF. Les éléments matériels sont générés à partir d'un code comportemental C qui est traité par le logiciel de synthèse haut-niveau Vivado HLS. Ce traitement est guidé par des directives exprimant la volonté de l'ingénieur (pipeliner une boucle, dérouler, etc.). Les descriptions matérielles générées sont elles aussi importées dans XPS afin de générer le fichier de configuration final (*bitstream*)

dédié à la programmation du FPGA. Les coûts du partitionnement peuvent alors être calculés à bas niveau en exécutant le système. Une fois un partitionnement satisfaisant atteint, on peut alors passer aux vérifications de post-partitionnement et à l'implémentation finale. Les auteurs ont testé leur méthodologie sur deux algorithmes et obtiennent des accélérations répondant aux contraintes tout en assurant une productivité et une réutilisabilité importante grâce aux techniques de synthèse haut-niveau.

Dans [23], les auteurs présentent une méthodologie ESL basée sur leur propre suite logicielle SpaceStudio. Cet outil permet la spécification du système à haut niveau d'abstraction (ce qui correspond à la phase 1 du flot ESL). La bibliothèque basée sur SystemC et le modèle de communication TLM-2 permettent la description fonctionnelle d'un système sous forme de blocs appelés modules. Lors de la phase de partitionnement (phase 3), il est possible d'associer ces modules à des ressources matérielles, auquel cas ces blocs sont alors passés dans un outil de synthèse haut niveau, HLS Vivado, permettant de recibler le code SystemC vers une description matérielle pour FPGA (phase 6). Les modules logiciels sont quant à eux compilés pour le processeur choisi lors de la phase de description de l'architecture (processeur dur ou présynthétisé, avec ou sans système d'exploitation). Le système complet peut alors être exporté vers une plateforme d'architecture matérielle (XPS) à partir de laquelle le fichier de configuration menant à l'implémentation finale sur FPGA peut être généré. Des outils de vérification intermédiaires sont mis à disposition dans la suite SpaceStudio permettant la simulation fonctionnelle (phase 1), puis la cosimulation matérielle/logicielle (phase 4) du système complet afin d'obtenir les premières estimations de taille, performance et puissance à différents moments du processus de développement.

La suite logicielle Intel Cofluent Studio [24] propose elle aussi une méthodologie d'exploration haut niveau de la fonctionnalité et de l'architecture d'un système électronique. L'outil permet de décrire le système grâce à une interface graphique. Les modules, qu'ils représentent la plateforme ou les fonctionnalités, sont représentés sous la forme de diagrammes blocs interconnectés par des canaux et synchronisés par évènements. Chacun de ces blocs peut alors posséder des propriétés comme le temps d'exécution ou la puissance consommée. De plus, les fonctionnalités peuvent être implémentées sous la forme de code C/C++ afin de pouvoir observer le fonctionnement du système lors de la simulation. Cofluent est de plus capable d'effectuer des simulations du système à deux niveaux d'abstraction. Le premier niveau est l'abstraction fonctionnelle pure, où l'architecture

cible n'est pas prise en compte et où il est possible de s'assurer du bon fonctionnement du système. Le second est la simulation niveau plateforme, où la vue plateforme est mappée à la vue fonctionnelle du système. Les estimations de cette simulation peuvent être recalculées de nombreuses fois en faisant varier les blocs fonctionnels et architecturaux afin de procéder à une véritable exploration architecturale avant de s'aventurer plus loin dans la conception du système et ainsi éviter les écueils lors de l'implémentation de fonctionnalités. Enfin, le logiciel permet de générer une description du système à haut niveau d'abstraction (TLM-2) en SystemC. La méthodologie Cofluent est basée sur un flot intitulé MCSE [25] qui se déroule en 4 étapes : Spécification, Description fonctionnelle, Spécification de l'implémentation et Réalisation. De ces 4 étapes, Cofluent en implémente deux, la description fonctionnelle et la spécification de l'implémentation. Il est facile de rapporter ces deux étapes aux étapes deux, trois et quatre de la méthodologie ESL présentée plus haut. L'outil ne permet toutefois pas de passer à l'implémentation.

1.4 Problématique

Cette brève revue de littérature nous a permis de dégager plusieurs évolutions importantes du domaine de la création de systèmes embarqués.

Tout d'abord, la complexité croissante des systèmes électroniques, que ce soit de par le nombre de plus en plus important de composants à intégrer ou bien de par l'hétérogénéité grandissante des systèmes, qui peuvent maintenant être constitués de plusieurs processeurs, coprocesseurs, puces dédiées ou encore FPGA, a créé un besoin d'abstraction et d'automatisation dans le processus de développement. Ce besoin s'est traduit par des créations successives de représentations de plus en plus abstraites : RTL, VHDL, SystemC, etc. L'adoption de ces représentations a engendré des outils d'automatisation du développement afin de raffiner ces représentations haut niveau : synthèse RTL et HLS. Ces langages et techniques ont permis l'amélioration de la productivité des équipes de développement. Toutefois, d'autres facteurs sont à prendre en compte.

En effet, afin d'améliorer l'efficacité du développement des systèmes embarqués, un besoin de standardisation s'est aussi beaucoup fait ressentir. Dans un premier temps, la standardisation des représentations a permis un essor de la réutilisabilité des descriptions matérielles (grâce aux deux normes HDL : VHDL et Verilog) sous la forme de composants protégés appelés IP. Depuis,

certaines entreprises se sont spécialisées dans la vente d'IP et l'utilisation de tels blocs dans un design est monnaie courante, ce qui permet aux équipes de se concentrer sur les quelques éléments du design qui seront spécialement développés pour l'application, réduisant les coûts et les temps de mise sur le marché. Dans un second temps, le domaine du développement de systèmes électroniques a aussi été marqué par le manque de méthodes claires afin d'amener un projet du cahier des charges jusqu'à l'implémentation finale. Un autre segment de la recherche dans le domaine a donc été dirigé vers la création de méthodologies visant à créer un cadre de travail. On a alors vu l'apparition de méthodes « du haut vers le bas », « du bas vers le haut », orientées plateforme, etc. Parmi ces méthodes, le flot ESL semble l'un des plus complets et englobe les concepts énoncés dans la plupart des autres flots. Cela est dû au fait que l'ESL représente un cadre assez « relâché » et constitue plus un guide de bonnes pratiques qu'une méthode à suivre à la lettre. C'est pour cela que beaucoup d'outils privés actuels peuvent être comparés au flot ESL. Comme cité dans le paragraphe précédent, le besoin d'abstraction croissant a aussi eu un impact sur les méthodologies développées, qui se définissent maintenant comme une série de raffinements visant à réduire de manière incrémentale le niveau d'abstraction d'un modèle. C'est à cause de nouveau paradigme de développement qu'une autre problématique se développe.

La méthodologie ESL, contrairement à une méthodologie classique de développement logicielle, le développement en cascade, ne peut se permettre d'attendre la moitié du cycle de développement avant de passer dans la phase de vérification. De ce fait, un troisième domaine important de la recherche en conception de systèmes embarqués apparait, celui de la vérification. En effet, le flot ESL entrecoupe ses phases de raffinement de phases de vérification en gardant en tête les préoccupations principales que sont le coût, les performances, la surface, la puissance, et le temps de mise sur le marché. Il est donc important de développer des outils permettant de générer des estimations de ces paramètres à tous les stades de la conception afin de guider l'exploration. Les outils dynamiques impliquent des environnements de simulation à différents niveaux d'abstraction ainsi que des suites d'analyse, ce que proposent des outils comme SpaceStudio, HLS Vivado, ou encore Cofluent. Le fait d'introduire la vérification à haut niveau a donc deux avantages : réduire l'espace d'exploration de manière incrémentale et repérer les impasses qui pourraient mener à l'échec d'un projet le plus tôt possible.

La plupart des solutions proposées par la recherche répondent à des parties de ces problèmes, toutefois, au meilleur de nos connaissances, il n'existe aucune méthode qui implémente l'intégralité

du flot ESL dans le domaine du système embarqué. C'est pourquoi le but de ce travail de recherche est d'étendre un flot de conception déjà existant afin de répondre à l'intégralité des étapes de la méthodologie ESL. La contribution principale de ce travail est la création d'un outil logiciel permettant le passage d'un code séquentiel C vers un format modulaire propice à l'exploration des compromis de partitionnements.

1.5 Objectifs

Afin de répondre à la problématique énoncée ci-dessus, le travail a été décomposé en plusieurs objectifs ou axes de développement :

- Analyser un ensemble de logiciels, privés ou non, disponibles sur le marché afin de construire le flot ESL.
- Analyser les parties du flot encore à concevoir et réaliser une solution logicielle capable de lier ensemble les différentes parties du flot déjà conçues ensemble.
- Permettre un accès facilité à un flot de codesign logiciel/matériel en proposant un point d'entrée codé en C et des outils pour guider la traduction vers la bibliothèque orientée système SystemC.
- Tester l'efficacité du flot de conception en le mettant à l'épreuve sur une application.

Puisque la contribution majeure de cette maîtrise est le développement d'un outil logiciel, il est important de définir des sous-objectifs liés à cette tâche. La conception logicielle portera sur la création d'un traducteur permettant la transformation d'une spécification logicielle vers une spécification propice au partitionnement. Cet outil devra répondre à ces objectifs de conception :

- Permettre une traduction assistée du code.
- Présenter une interface claire à l'utilisateur.
- Produire un code de sortie clair.
- Ne pas modifier la fonctionnalité du code source.
- Imposer un ensemble le plus restreint possible de contraintes sur le format du code d'entrée.

1.6 Résumé des contributions

Afin de composer ce flot de conception, plusieurs étapes auront été nécessaires :

- Premièrement, une analyse en profondeur d'ouvrages de référence a été effectuée. Cette analyse a permis de créer une vue d'ensemble du domaine de la conception de systèmes embarqués et a aidé à comprendre les enjeux du domaine afin d'affiner la problématique et les objectifs de cette maîtrise.
- Dans un second temps, une recherche documentaire a permis d'examiner les solutions propriétaires disponibles sur le marché ainsi que celles offertes à Polytechnique Montréal. Suite à cette recherche, une sélection d'outils a été effectuée afin de composer le flot ESL.
- Lors d'une troisième étape, une solution logicielle a été développée afin de compléter le flot ESL proposé. Cette solution permet la transition d'un code C fonctionnel vers une architecture modulaire propice à l'exploration architecturale.
- Dans une dernière étape, la méthodologie complète a été mise à l'épreuve sur un exemple de traitement d'image : un filtre de Canny (détecteur de contours).

1.7 Organisation du mémoire

L'organisation de la suite de ce mémoire est présentée dans cette section.

Dans le chapitre 2, un survol du flot vous sera présenté : les différents outils choisis et leur intégration dans le flot ESL présenté dans l'introduction. Une fois ce premier survol effectué, une description plus détaillée de chaque étape du flot sera effectuée.

Le chapitre 3 expliquera le développement et le fonctionnement du logiciel conçu lors de ce projet de maîtrise et nommé dans ce qui suit *C2Space*, permettant la jonction entre les analyses menées par l'outil Pareon et le code d'entrée de SpaceStudio.

Un test du flot complet sera détaillé lors du chapitre 4 afin d'évaluer la validité du flot

Le test donnera lieu à une discussion dans le chapitre 5 afin de présenter l'intérêt des résultats et donner quelques pistes d'amélioration de la solution.

CHAPITRE 2 PRÉSENTATION DU FLOT

2.1 Présentation générale

La solution présentée dans ce mémoire repose sur un flot de conception basé sur le flot ESL présenté en introduction. Afin de répondre aux différentes étapes de ce flot théorique, nous utilisons différentes solutions.

La figure 2-1-1 représente comment ces différentes solutions viennent s'intégrer au flot ESL.

Le point d'entrée de notre méthodologie est une application logicielle décrite en C. L'intérêt de choisir un tel point d'entrée est de permettre aux utilisateurs du flot d'accélérer rapidement une application purement logicielle en faisant passer certains de ses noyaux de calculs dans le domaine matériel. Bien que le but soit de permettre un passage le plus rapide possible d'une spécification C à un modèle de codéveloppement logiciel/matériel, nous avons décidé d'imposer un ensemble de contraintes au code C afin de rendre possible la traduction du code par l'outil C2Space. Ainsi, certaines constructions propres au langage C doivent être enlevées du code. Ces conditions seront décrites en détail dans la troisième section de ce mémoire. Afin de guider l'exploration du design, le code C doit être accompagné d'un ensemble de contraintes propres au domaine des systèmes embarqués (consommation de puissance, rapidité d'exécution, temps de développement et quantité de ressources utilisées). Ces contraintes permettront tout au long du flot de vérifier que les solutions retenues respectent bien l'intention originale de développement.

Cette démarche vient s'inscrire dans la phase un du flot ESL. Le code C représente une spécification exécutable fonctionnelle du système que nous voulons implémenter. Cette représentation est, comme le recommande la méthodologie, la plus exempte possible d'artefacts d'implémentation, en cela qu'elle ne fait encore aucun postulat sur la manière dont l'application va être scindée entre les ressources matérielles et logicielles. Elle permet de s'assurer de la fonctionnalité du système. On notera toutefois quelques concessions faites à la méthodologie ESL, qui préconise normalement une première étape la plus dénuée possible d'artefacts logiciels, de par la modification du code pour permettre l'utilisation de C2Space ainsi que par l'utilisation d'une spécification fonctionnelle, qui représente une première implémentation algorithmique.

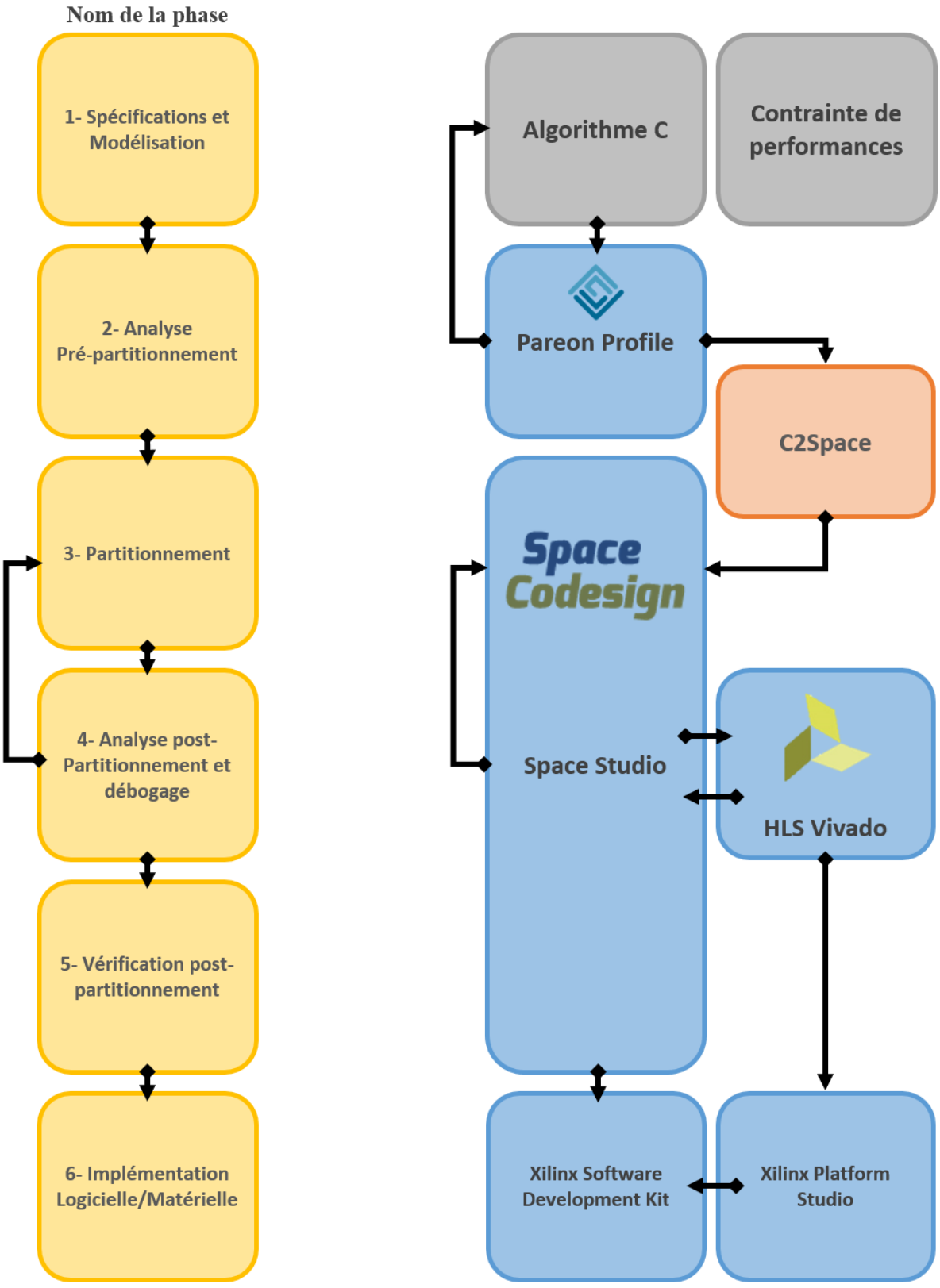


Figure 2.1-1 : Comparaison entre le flot ESL et le flot présenté

Dans la suite du flot, nous amenons notre spécification C dans l'outil développé par Vector Fabrics, Pareon Profile [16] afin d'effectuer une première analyse dynamique de l'application. Pareon fournit un compilateur instrumentant le code et permettant d'effectuer une analyse des dépendances de données lors de l'exécution en scrutant les accès mémoire. Toutes les informations récupérées lors de l'exécution sont alors compilées dans un fichier lisible dans l'interface graphique de Pareon. Ce rapport permet au développeur de naviguer dans l'arbre d'exécution du code et de rechercher les opportunités de parallélisme des différentes boucles composant le système. Si une boucle présente une opportunité de parallélisme, le logiciel calcule l'accélération attendue basée sur une exécution multicœur et la loi d'Amdahl. Le but de cette analyse est de guider la segmentation du code C séquentiel en partant du principe qu'une section du code présentant une opportunité d'accélération sur un processeur multicœur présentera aussi une grande opportunité d'accélération en tant que coprocesseur matériel.

Cette étape d'exploration de l'algorithme coïncide parfaitement avec la phase 2 de la méthodologie ESL, car elle permet d'éliminer une partie de l'espace de solutions sans trop compromettre l'abstraction de la spécification. Des modifications peuvent être rajoutées au code C afin de découvrir plus de parallélisme ou il peut être décidé de changer complètement d'algorithme si aucune opportunité de parallélisme n'est découverte (sans sacrifier la fonctionnalité attendue par le cahier des charges). À l'issue de cette phase, nous obtenons une implémentation algorithmique présentant des opportunités de parallélisme, qui vont servir à guider C2Space.

C2Space ne fait pas à proprement parler partie du flot ESL, mais sert à lier les différents éléments du flot. Cela répond à plusieurs objectifs. Tout d'abord, cela permet de lier deux outils qui n'ont aucune interface commune (SpaceStudio et Pareon Profile) sans intervention majeure du développeur. De plus, l'utilisation de cet outil permet de réaliser un des objectifs de ce mémoire qui est de rendre accessible le design de systèmes embarqués à des ingénieurs ayant peu d'expérience dans le domaine matériel. En effet, la bibliothèque C++ utilisée dans SpaceStudio est dérivée de SystemC, qui est une bibliothèque orientée système et qui introduit des notions telles que les modules et les communications et synchronisations intermodules. L'idée derrière C2Space est de traduire l'algorithme séquentiel précédemment analysé en un ensemble de modules SpaceStudio et de générer toutes les communications nécessaires entre le logiciel et le matériel afin de conserver la fonctionnalité originale du code séquentiel. C2Space a été développé en C++ et utilise à sa base la bibliothèque Clang [26], une façade (*frontend*) C++ du compilateur LLVM, afin

d'effectuer une analyse statique du code source. Au cours de l'analyse, l'utilisateur doit fournir certaines informations afin de guider le découpage du code source en modules (ces informations découlent des résultats de l'analyse de Pareon). Une fois les informations compilées, C2Space procède alors à une réécriture intelligente du code. Le résultat de cette phase consiste en un ensemble de fichiers contenant les modules, des fichiers supports ainsi qu'un script python permettant la génération du projet sous SpaceStudio.

Dans la suite du flot de développement, nous reprenons le résultat généré par C2Space pour procéder à l'exploration architecturale du design. Suite à la phase précédente, le code source est maintenant séparé en modules SpaceLib (la bibliothèque basée sur SystemC de SpaceStudio). Le flot interne de SpaceStudio suit alors son cours. SpaceStudio cible, à l'heure actuelle, les plateformes basées sur les FPGA Xilinx. Ainsi, le flot classique est axé sur le partitionnement des modules entre les différentes ressources présentes sur les plateformes.

Dans notre flot, il est possible d'influencer trois catégories de paramètres afin de générer un partitionnement. La première catégorie est la configuration matérielle. Il est possible de définir quels éléments nous voulons voir présents dans notre configuration. Ainsi, nous pouvons spécifier des processeurs (ARM, microBlaze), mais aussi des IP prédéfinies (comme des minuteries, des banques mémoires). La seconde catégorie est bien évidemment le partitionnement lui-même, c'est-à-dire l'allocation des ressources matérielles aux blocs applicatifs. Enfin, la troisième façon d'influencer le design est le guidage de la synthèse haut niveau des modules matériels. Selon les contraintes de synthèse, les modules matériels n'exhiberont pas les mêmes propriétés en termes d'espace utilisé, de puissance et de performance. L'approche d'exploration sélectionnée est basée sur une réduction progressive de la complexité de l'espace de solutions tel qu'illustré à la figure 2.1-2.

Une fois le partitionnement effectué, il est possible de simuler le fonctionnement du système à haut niveau pour vérifier le respect des fonctionnalités. La simulation donne aussi accès à beaucoup d'informations comme le taux d'utilisation des processeurs disponibles, l'utilisation des bus et le temps pris par les communications intermodules. Les informations concernant l'utilisation des ressources du FPGA (pour les modules destinés à devenir des coprocesseurs) deviennent disponibles par la suite du flot et peuvent être réintégrées aux diagnostics fournis par SpaceStudio pour avoir un premier aperçu global des compromis du partitionnement.

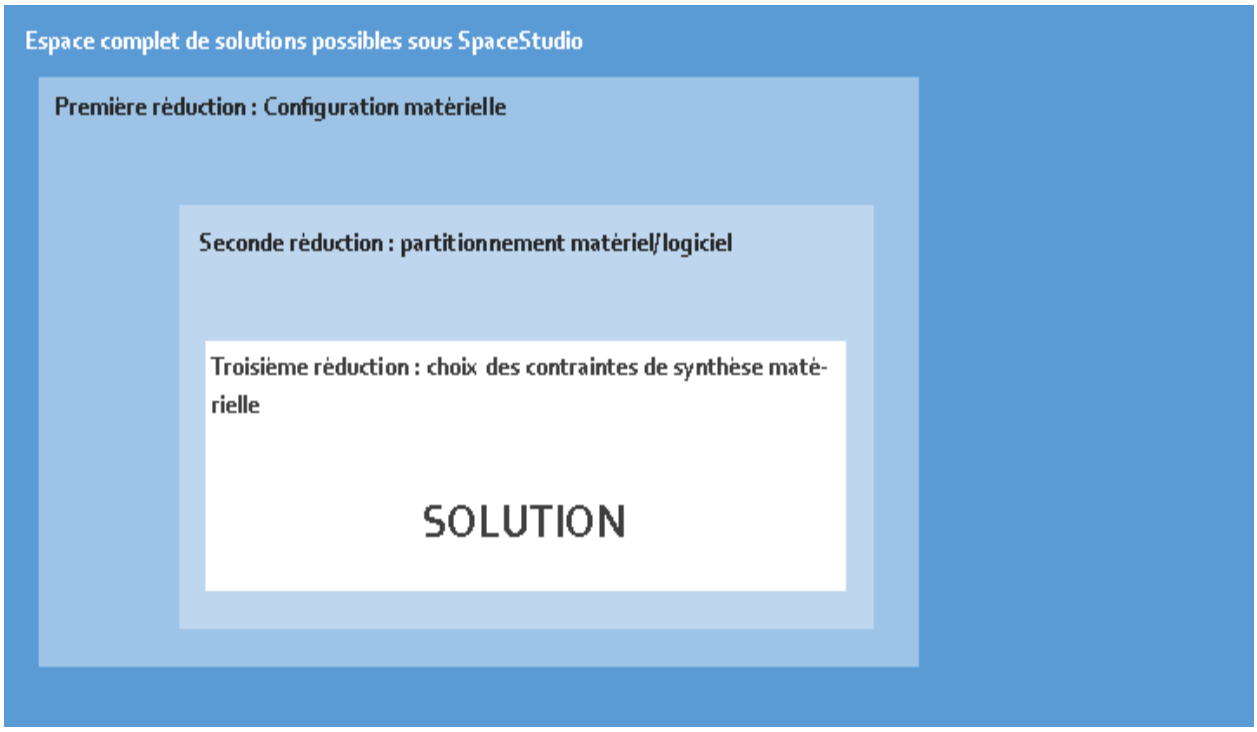


Figure 2.1-2 : Réduction de l'espace de solution dans notre méthodologie

Ces informations sont générées par HLS Vivado, après que celui-ci ait procédé à l'optimisation des modules par le biais son interface. SpaceStudio permet l'importation des modules destinés à être spécifiés pour le FPGA directement dans HLS Vivado. Une fois cette transition faite, il est possible de donner des consignes d'optimisation au synthétiseur afin de générer un artefact matériel plus ou moins imposant. L'utilisation de ces contraintes permet d'équilibrer performance, espace et puissance requise afin de rencontrer les contraintes de la spécification. Les constats effectués lors de l'analyse algorithmique sont utilisés ici pour guider facilement le développeur vers l'optimisation des boucles ou nids de boucles possédant un fort potentiel de déroulement et/ou de pipelinage.

Si les résultats obtenus ne sont pas suffisants pour rencontrer les contraintes, l'exploration architecturale continue. Il est alors possible de créer une liste de configurations afin de créer différents compromis en termes de vitesse, coût, surface, puissance.

Ces phases d'exploration menées par SpaceStudio représentent les phases trois à cinq du flot de développement ESL. Durant ces étapes, nous sélectionnons un partitionnement de nos éléments logiciels (les modules) vers nos ressources matérielles (processeurs ou coprocesseurs). Une phase d'exploration des différentes combinaisons dans un simulateur nous permet d'obtenir rapidement

un espace de compromis architecturaux. Il est important de noter qu'une partie de l'abstraction de la solution est abandonnée dès lors que nous utilisons SpaceStudio, car le choix de ce logiciel limite le nombre de plateformes matérielles que nous pouvons sélectionner (en l'occurrence des plateformes basées sur les FPGAs de Xilinx comme la Zedboard [27] de Avnet). Toutefois, la flexibilité des FPGAs permet d'obtenir des résultats satisfaisants et le développement basé sur une plateforme permet de réduire les temps de mise sur le marché d'une solution.

Une fois cette phase d'exploration terminée, et qu'une solution répondant au cahier des charges a été validée en simulation, on peut passer à la dernière phase du flot : l'implémentation. Le passage du projet SpaceStudio vers une implémentation est géré en deux parties distinctes. Les éléments logiciels de la configuration sont compilés pour le processeur ciblé et sont prêts à être utilisés. La logique de communication, les pilotes (*drivers*) et le Paquetage de Support de Plateforme (BSP) sont générés automatiquement par SpaceStudio. Les éléments matériels quant à eux passent à nouveau dans la suite de synthèse haut-niveau Vivado, en spécifiant les contraintes qui avaient été précédemment sélectionnées. Les descriptions matérielles de ces éléments sont alors générées et intégrées en tant qu'IP dans la suite de visualisation de plateforme XPS. Une fois la plateforme entièrement générée, le fichier de configuration de la plateforme (*bitstream*) est créé et l'intégralité du système est unifié dans l'environnement de développement EDK. Il est alors possible de passer aux tests de l'implémentation en téléversant le fichier de configuration du FPGA ainsi que les différents fichiers applicatifs (drivers, modules logiciels, etc.) dans la carte de développement. On peut alors déboguer le système en conditions réelles à l'aide des outils de débogage présents dans l'environnement de développement. Si le flot a été bien respecté, cette étape de vérification ne devrait pas être trop chronophage, car la majorité des problèmes doivent déjà avoir été repérés aux différents niveaux d'abstraction.

Cette dernière étape de vérification clôt notre flot. La situation présentée ici est la situation optimale, dans laquelle tout se passe bien. Toutefois, il est possible que pour certains projets il faille reprendre une étape précédente, si l'exploration se heurte à un mur (non-respect des contraintes). Il peut être nécessaire d'avoir recours à un autre algorithme de base ou encore de raffiner les communications afin de rencontrer les contraintes.

Bien qu'un retour en arrière puisse être requis, ce flot essaye de faire apparaitre les problèmes à haut niveau, afin de pouvoir les résoudre lorsque l'abstraction est encore grande, et que les modifications peuvent encore être rapidement implémentées et testées.

CHAPITRE 3 C2SPACE : UN OUTIL DE TRADUCTION ASSISTÉE

Ce chapitre présente la contribution majeure de ce mémoire, à savoir C2Space, un outil de traduction semi-automatisée de code C, basé sur la bibliothèque Clang, la partie frontale du compilateur source à source (aussi appelé transpilateur) LLVM.

La problématique d'un tel outil repose sur un fait simple : le langage C est trop peu complexe pour capturer l'ensemble des artefacts d'un système matériel, à commencer par le fait que le langage C soit purement séquentiel. Ainsi, un outil faisant passer un tel code vers une implémentation orientée matérielle, bien qu'abstraite, demande la création d'« information » supplémentaire. Le logiciel doit alors prendre des décisions motivées par des constats externes au code lui-même. Un des problèmes les plus évidents est la segmentation du code : où séparer le code pour obtenir des modules efficaces ? Mais bien d'autres problématiques peuvent être relevées, par exemple comment stocker les éléments d'une boucle de traitement de tableau ou encore comment les envoyer d'un module à l'autre. Toutes ces questions font de C2Space un outil plus complexe qu'un simple traducteur « un pour un ». C2Space essaye de traduire un paradigme de programmation vers un autre paradigme.

L'approche que nous avons choisie pour effectuer cette transformation est basée sur l'analyse du potentiel de parallélisme de la source, à l'aide de l'analyseur dynamique Pareon Profile, suivie d'une analyse statique, typique des compilateurs, pour récupérer les informations nécessaires à la réécriture du code. L'utilisateur est sollicité pour donner des informations complémentaires au traducteur à l'aide d'une simple interface console.

Dans ce chapitre, nous parlerons en détail de la bibliothèque Clang ainsi que des principaux autres compilateurs sources à sources existants. Nous présenterons ensuite l'architecture de C2Space et passerons en revue tous les éléments qui le constituent en utilisant un code exemple démontrant les fonctionnalités de l'outil.

3.1 Les transpilateurs

Les transpilateurs sont une catégorie à part entière de compilateurs. Le but d'un transpilateur est d'amener un code source, écrit dans un langage particulier, vers un autre langage tout en gardant le comportement de l'original. Pour parvenir à cet objectif, un transpilateur va utiliser les

techniques habituelles de la partie frontale d'un compilateur classique. Le compilateur va tout d'abord procéder aux analyses lexicales et syntaxiques du langage, en vérifiant respectivement la validité des jetons (éléments singuliers du langage comme des mots réservés ou des noms de variables) puis la validité des constructions de ces jetons, en accord avec les règles de construction du langage. Pour certains langages, comme le C, une phase de précalcul peut être effectuée avant l'analyse syntaxique afin de remplacer certains jetons du code par des équivalents (définis par les macros, comme #define). À l'issue de l'analyse syntaxique, le code source a été entièrement consommé et la représentation de celui-ci est un arbre de syntaxe (les jetons sont assemblés et les branches de l'arbre représentent les relations syntaxiques qu'entretiennent ceux-ci) comme illustré sur la figure. 3.1-1.

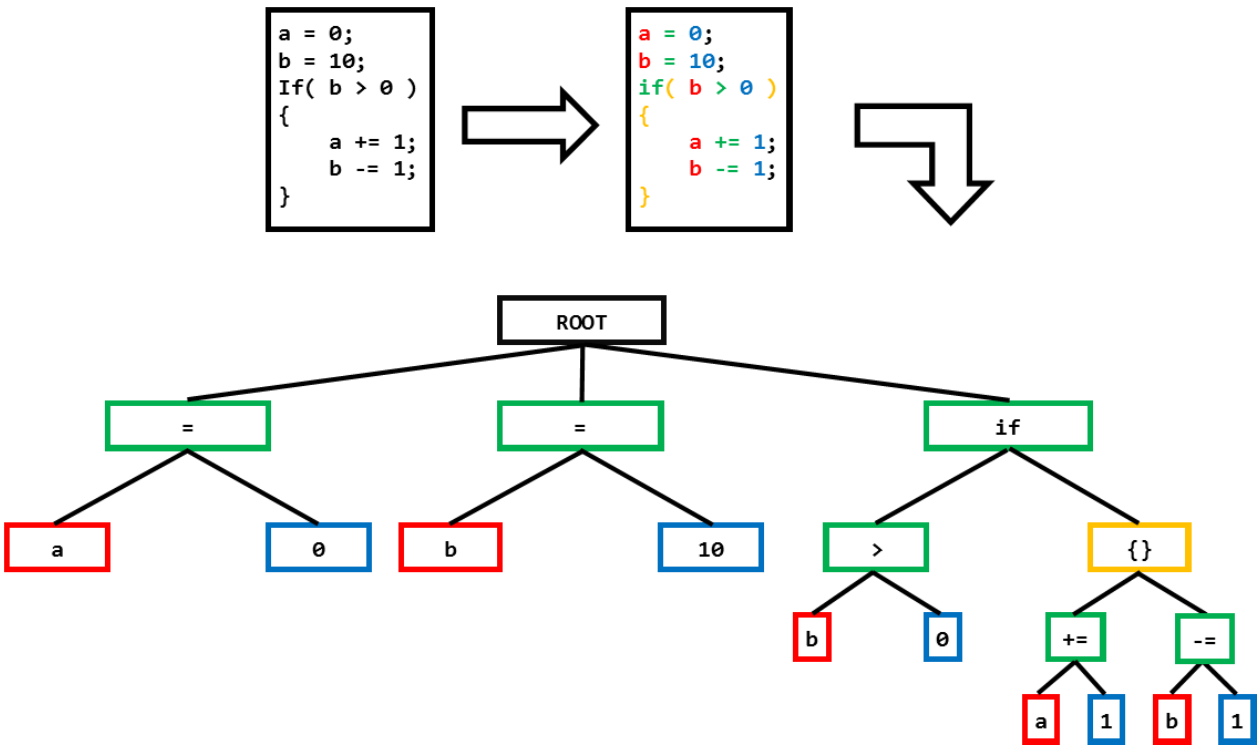


Figure 3.1-1 : Illustration de la partie frontale d'un compilateur

C'est à partir de cette représentation sous forme d'arbre syntaxique que la plupart des transpilateurs effectuent le reformatage du code vers un autre langage. L'arbre de syntaxe est consommé par un ou plusieurs visiteurs afin de repérer des constructions typiques du langage cible. Une fois que l'arbre complet a pu être évalué, chaque construction se voit réécrite dans le langage cible.

3.1.1 L'infrastructure de compilation ROSE

Le compilateur ROSE [28] est une infrastructure code ouvert de compilation permettant l'analyse et la traduction de code source à source développé au laboratoire national Lawrence Livermore en 2000. Il prend en charge notamment les codes sources C, C++ et Fortran, mais aussi les directives de parallélisation OpenMP. Il propose une bibliothèque C++ permettant l'interfaçage avec l'utilisateur. Les principaux buts de ROSE [29] sont (1) de rendre la puissance d'un compilateur accessible à des utilisateurs non avertis afin d'améliorer les performances et le rendement des développeurs et (2) de fournir une infrastructure pour les développeurs permettant la création d'outils personnalisés d'analyse et de traduction de code source à source. Ainsi, il est possible d'utiliser ROSE afin de procéder à des passes d'analyse comme la création de graphes d'appels ou encore l'analyse des pointeurs dans le code. L'outil ROSE permet aussi de créer des traducteurs de code. La méthode à suivre est proche de celle décrite ci-dessus. Le code source est analysé par l'infrastructure de ROSE afin de créer une représentation intermédiaire du code. Cette représentation intermédiaire est un arbre de syntaxe abstrait (AST) orienté objet. Les nœuds sont représentés par des sous-classes de la classe principale `SgNode`. Une fois la représentation intermédiaire créée, il est nécessaire de traverser l'arbre afin d'y appliquer des modifications, rendues plus faciles par des méthodes contenues dans la classe `SageBuilder`. Une fois les modifications voulues implémentées dans le code, ROSE propose un outil pour retransformer l'arbre de syntaxe modifié en code source. À partir de cette étape, l'infrastructure permet soit de terminer le flot, soit de compiler la nouvelle source à l'aide d'un compilateur commercial. Il est intéressant de noter que l'analyse syntaxique du code menant à la représentation intermédiaire est déléguée à l'outil commercial EDG. Cette partie de la bibliothèque est donc fournie sous la forme d'un fichier binaire.

Plusieurs traducteurs utilisant le compilateur ROSE ont récemment été développés. En 2013, un groupe de recherche [30] a développé un outil de traduction de boucles imbriquées utilisant la méthode appelée transformation unimodulaire [31]. Cette méthode basée sur une analyse mathématique des dépendances sur les boucles imbriquées permet de trouver une transformation du nid de boucles levant les dépendances. Les auteurs ont donc automatisé ce processus à l'aide des fonctions de remplacement d'éléments dans l'arbre de syntaxe abstrait de ROSE. Les résultats sont positifs pour des codes séquentiels à optimiser pour des processeurs et des cartes graphiques.

Toutefois, cette méthode présente des limitations, car elle nécessite des boucles parfaitement imbriquées.

Aloor et Nandivada [32] ont quant à eux présenté en 2015, un modèle de travailleur unique pour OpenMP, une extension du langage C/C++ permettant les exécutions en parallèle de certaines parties d'un code en appliquant des `pragmas` qui seront interprétés par le compilateur. Les auteurs avancent que la façon dont les compilateurs allouent les travailleurs (fils d'exécution) aux tâches à effectuer peut affecter le comportement du code parallélisé. S'il y a 4 tâches et 2 travailleurs, chacun des deux travailleurs recevra deux tâches à accomplir. Cette méthode peut être très efficace, mais peut mener à des problèmes dans le cas où le code à paralléliser nécessiterait des barrières (par exemple, si la partie S2 des tâches doit s'exécuter après que la partie S1 soit terminée sur toutes les tâches), car les barrières affectent les travailleurs et non les tâches. Un nombre important de transformations doivent être appliquées au code pour assurer le bon comportement et affectent la lisibilité du code. Les auteurs proposent un langage UW-OpenMP pour lequel l'hypothèse « un travailleur est alloué à chaque tâche » est toujours vérifiée. Ainsi les problèmes de barrière disparaissent. Le langage UW-OpenMP n'est pas destiné à être compilé tel quel. Un outil de traduction développé à l'aide de l'infrastructure ROSE permet le passage automatique d'un code UW-OpenMP à un code OpenMP équivalent suite à une série de transformations. La preuve formelle de l'exactitude de cette transformation est développée dans l'article, ce qui assure un bon degré de confiance sur le code généré (bien qu'après transformation le code devient obfusqué).

3.1.2 Clang : Une partie frontale C/C++ pour le compilateur LLVM

Un autre compilateur source à source intéressant est Clang. Cet outil, qui est en réalité la partie frontale d'un compilateur code ouvert re-ciblable appelé LLVM, a été rendu libre d'accès en 2007 [33]. Ce compilateur fonctionne sur la base d'une représentation intermédiaire spécialement développée. En partant de cette représentation intermédiaire, LLVM est capable d'effectuer optimisations et génération de code pour de nombreux processeurs. À l'aide de librairies bien documentées, il est donc possible de se servir de LLVM comme de la partie arrière (*backend*) d'un compilateur pour n'importe quel langage source. C'est sur cette idée qu'a été basé le compilateur Clang : proposer une partie frontale performante pour les langages basés sur C qui vient se greffer sur les outils de LLVM. Les objectifs de Clang sont (1) de proposer un compilateur efficace, fiable, explicite et compatible avec GCC et (2) de proposer, de par son architecture modulaire, un

environnement propice au développement d'outils destinés à différentes catégories de clients (traduction de code, analyse statique, génération). Clang, en tant que compilateur présente ainsi des avantages comparativement à GCC, notamment par la génération de commentaires de corrections d'erreurs plus détaillés. Cela est rendu possible par la génération d'une représentation du code sous forme d'arbre de syntaxe abstraite (AST) bien plus détaillée que la représentation générée par GCC. Cet arbre de syntaxe est basé sur un modèle orienté objet où les nœuds sont tous des descendants de deux classes abstraites mères : `Stmt` et `Decl`. Clang propose ainsi une série d'outils organisés en librairies permettant d'agir de diverses manières sur cet arbre (la bibliothèque `ClangRewrite`, par exemple, permet la réécriture de portions de l'arbre dans le langage source). La méthodologie d'utilisation de Clang en tant que transpilateur est comme suit. Le code source est converti en arbre de syntaxe abstraite. À tout moment, une correspondance existe entre les nœuds de l'arbre et la position du curseur dans le tampon source. Il est alors possible de naviguer à travers l'arbre en recherchant certains types de nœuds ou bien de chercher des constructions particulières dans cet arbre à l'aide d'un système proche des expressions régulières. Une fois les éléments identifiés, il est possible d'utiliser un outil de réécriture de source permettant de remplacer certaines parties de la source ou bien de réécrire un intervalle de l'arbre dans le langage source.

Beaucoup de projets basés sur Clang ont aussi vu le jour ces dernières années. Kaushik et Patel [34] présentent en 2013 un outil basé sur Clang appelé SystemC-clang. Cette infrastructure permet l'analyse d'un modèle SystemC mixte (possédant des modules décrits à la fois au niveau transfert registre et au niveau transactionnel). À l'issue de l'analyse, une représentation intermédiaire du système modélisée sous la forme d'un ensemble de classes est générée permettant des travaux tels que des transformations de code ou encore de l'analyse statique. Les auteurs présentent en exemple un module d'extension prenant en entrée cette représentation intermédiaire afin de récupérer les informations nécessaires à l'ordonnancement de la simulation SystemC sur plusieurs processeurs en parallèle. Afin de parvenir à cette représentation intermédiaire, les auteurs ont utilisé les capacités de Clang. À partir de l'arbre de syntaxe abstraite généré par Clang, ils développent un visiteur cherchant les informations pertinentes de chacun des éléments qu'ils veulent rajouter à leur représentation. Par exemple, à chaque fois que le visiteur rencontre une déclaration de classe, il vérifie si cette classe hérite de `SC_MODULE`, la classe représentant les modules SystemC. Si c'est le cas, de multiples informations structurelles, comme le type d'accès des ports ou le type de données, sont enregistrées en traversant la branche correspondante de l'arbre. L'architecture de l'outil

SystemC-clang est pensée pour l'ajout de modules complémentaires tirant parti des informations contenues dans leur représentation intermédiaire. Il serait ainsi possible de créer un traducteur à l'aide de cette infrastructure.

Une autre initiative menée en 2013 par des employées de la firme Google [35] visait à appliquer des transformations sur de larges quantités de code afin de respecter les nouveaux standards pouvant être introduits par des changements dans des bibliothèques ou dans les normes. Deux objectifs principaux sont mis en avant pour cet outil : (1) sa rigueur lors de certaines transformations complexes (par exemple la distinction entre deux méthodes ayant le même nom, mais n'appartenant pas à la même classe) et (2) sa capacité de mise à l'échelle (analyser des corpus de code importants en un temps raisonnable). Afin de réaliser ces objectifs, un flot de travail est développé. En premier lieu, un indexeur de sources permet de décrire la manière de compiler une base de données C++ en un ensemble d'arbres de syntaxe abstraite. Par la suite, un ensemble d'outils de transformations ciblant des nœuds précis (appelés `ASTMatchers`) traversent ces ASTs et enregistrent des instructions de transformations à appliquer avec toutes les informations contextuelles nécessaires. Enfin, un consommateur de transformations s'occupe d'appliquer chacune de ces modifications. Les unités de traductions produites par Clang étant indépendantes, les outils `ASTMatchers` peuvent être appliqués en parallèle sur de multiples fichiers. Cela est achevé à l'aide de la librairie `MapReduce`. La méthode de recherche de nœuds spécifiques diffère de celle de l'article présenté précédemment par l'utilisation des `ASTMatchers`, sorte d'expressions régulières pouvant cibler une structure particulière dans l'arbre (composée de plusieurs nœuds, présentant ou non des caractéristiques spécifiques). À ces `ASTMatchers` sont associées des fonctions à exécuter en cas de correspondance avec une partie de l'unité de traduction. De par l'architecture de ce projet, ClangMR s'adapte facilement à n'importe quelle transformation sans avoir à modifier tout le pipeline. Il suffit de définir un nouvel ASTMatcher, ainsi que sa fonction de transformation associée et de l'ajouter au flot d'exécution.

3.1.3 Choix du transpilateur

Comme nous l'avons vu dans la section 3.1, il existe deux grands transpilateurs code ouvert, Clang et Rose, et il a été nécessaire de sélectionner l'un des deux afin de créer notre outil de traduction. Bien que les deux outils présentent des fonctionnalités semblables, nous avons décidé d'utiliser Clang pour plusieurs raisons. La première raison est que la partie analyse syntaxique de Clang,

contrairement à ROSE, est à code ouvert, nous permettant l'accès aux sources, ce qui nous a aidés lors de la phase préliminaire de compréhension de l'outil. La seconde raison pour laquelle nous avons choisi Clang est la facilité d'installation de celui-ci : une simple commande nous permet d'aller chercher les fichiers sources de LLVM et Clang dans un dépôt de fichiers et de compiler ceux-ci. Le processus d'installation de ROSE est bien plus laborieux et est moins bien documenté (à noter qu'il existe toutefois une image de machine virtuelle comprenant une version de ROSE préinstallée disponible sur le site du projet). Une autre raison qui nous a poussés à utiliser Clang est une meilleure documentation de celui-ci comparativement à ROSE. Plusieurs tutoriels vulgarisant différents aspects de l'outil sont disponibles, aussi bien sur le site du projet que sur des sites indépendants. Enfin, la raison majeure qui a motivé ce choix est la présence du système de ciblage de nœuds spécifiques développé sur l'outil Clang, qui nous a beaucoup aidés lors du développement à réduire la quantité de code nécessaire à l'implémentation de certaines actions.

3.1.4 Fonctionnement de Clang

Dans cette section, nous allons expliquer plus en détail le fonctionnement des outils de Clang. Nous avons choisi de cibler les concepts qui sont extensivement utilisés dans C2Space.

3.1.4.1 L'arbre de syntaxe abstraite

Clang permet l'analyse d'un code source à l'aide d'un arbre de syntaxe abstraite (AST). L'avantage majeur de l'AST de Clang est sa précision comparativement à d'autres compilateurs comme GCC. En effet, cet arbre va garder en mémoire toutes les informations qui composaient le code original. Ainsi, pour une portion de code de ce type,

```
1  variable2 = ((((variable1)))) + 2;
```

Extrait 3.1-1 : Illustration de l'exhaustivité de l'AST de Clang

Les 4 ensembles de parenthèses sont conservés dans l'AST et ne sont pas supprimés par quelconque optimisation. Cette propriété rend l'AST de Clang idéal pour nous permettre d'analyser le code source tel qu'il a été écrit.

Les nœuds de l'AST de Clang sont représentés par des classes C++. Contrairement à d'autres représentations où tous les nœuds dérivent d'une classe unique, la hiérarchie est basée sur plusieurs nœuds ancêtres comme `Decl`, `Stmt` ou encore `Type` qui possèdent chacun leur propre descendance.

Certains nœuds quant à eux, sont complètement indépendants. La traversée d'un tel AST pour analyse est effectuée de la manière qui suit : en démarrant du nœud racine `TranslationUnitDecl` et en traversant chacun des nœuds enfants récursivement. La traversée est implémentée spécifiquement pour chaque enfant. À la traversée, les nœuds sont transtypés dans leur type concret afin de pouvoir effectuer des actions sur les attributs particuliers de chaque nœud.

Bien que la structure exhaustive de l'AST de Clang permette de conserver la sémantique du développeur, cela rend l'analyse de l'arbre complexe. Soit la portion de code de l'extrait 3.1-2, la représentation en arbre de syntaxe abstraite créée par Clang est présentée à l'extrait 3.1-3. On peut voir que même pour une portion de code très réduite, la représentation résultante reste assez conséquente et que certains nœuds peuvent être ajoutés bien que non écrits explicitement par le développeur. À titre d'exemple, les deux transtypages implicites d'une Lvalue en Rvalue aux lignes 11 et 15 de l'extrait 3.1-3, correspondant aux passages d'étiquette à valeur requis pour le calcul de l'expression `x/42` et au retour de fonction `return result` (respectivement pour `x` et `result` qui sont convertis en les valeurs qu'ils représentent). Lors de la traversée de l'AST pour analyse, il est donc important de penser à ces transformations implicites que subit le code.

3.1.4.2 Agir à la compilation

Afin de pouvoir accéder à l'AST généré par l'analyse syntaxique du code par Clang, la bibliothèque met à notre disposition un paquetage d'outillage appelée `tooling`. Ce paquetage fournit des classes utilitaires comme `CommonOptionsParser`, qui permet l'analyse des lignes de commandes ou encore `CompilationDatabase`, une interface pour les bases de données de compilation JSON. `Tooling` fournit aussi une série de classes permettant d'implémenter des actions après l'analyse syntaxique. La classe `ClangTool` permet d'appliquer facilement des actions à une base de données de compilation. Ces actions sont encapsulées dans des héritiers de la classe `ToolAction` et plus particulièrement de la classe `FrontendAction.` Les classes dérivées de `FrontendAction` peuvent être définies par l'utilisateur ou générées automatiquement (selon la méthode de navigation dans l'arbre). Dans les deux cas, les actions sont instanciées par une classe usine `FrontendActionFactory,` qui sera ensuite passée en paramètre à l'exécution de notre `ClangTool`. L'intérêt du patron usine est de permettre la création automatique d'une nouvelle action pour chaque unité de traduction analysée.

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}
```

Extrait 3.1-2 : code exemple tiré des tutoriels Clang

```
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... Nous supprimons une partie des déclarations internes à Clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

Extrait 3.1-3 : AST du code exemple de l’extrait 3.2-2

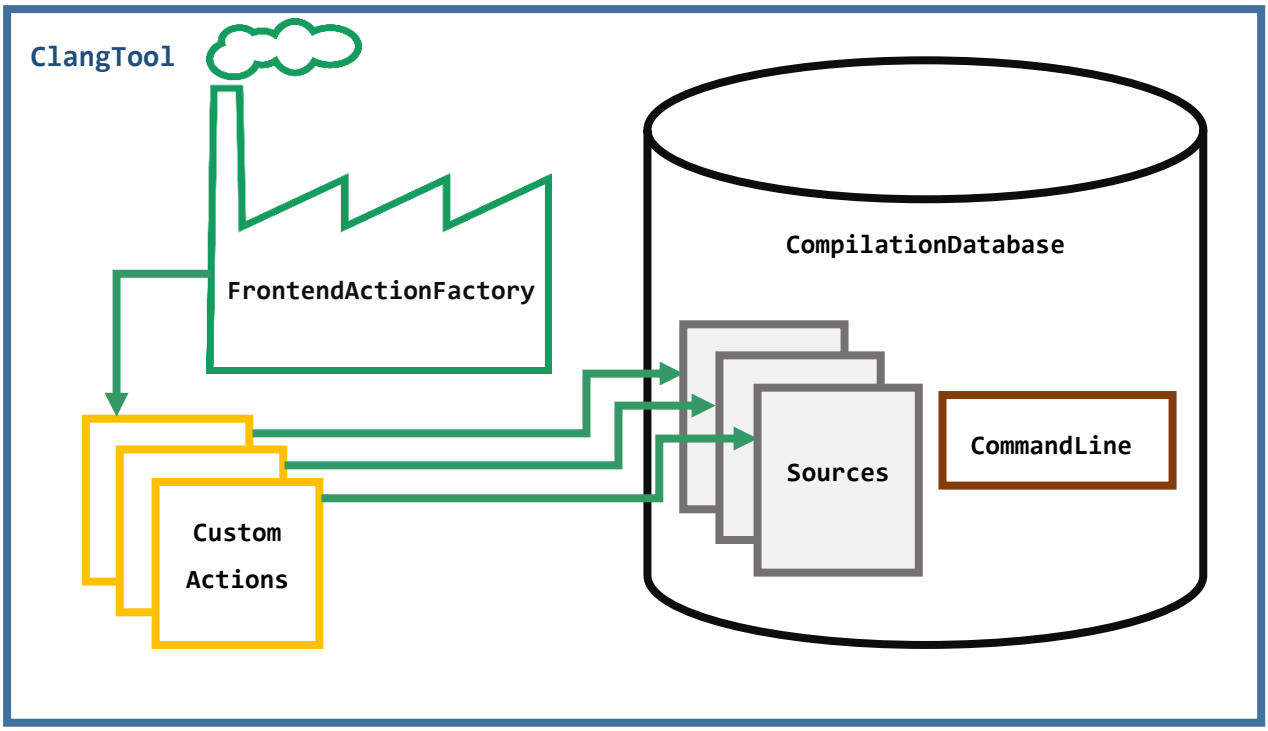


Figure 3.1-1 : Illustration de la structure de la classe `ClangTool`

Dans la partie suivante, nous expliquons comment peupler une action basée sur `FrontendAction`

3.1.4.3 Naviguer et agir sur l'arbre de syntaxe

Clang met à disposition deux structures permettant la navigation dans l'arbre : les consommateurs d'AST (`ASTConsumer`) et les patrons de correspondance (`ASTMatchers`). Chacune de ces deux constructions utilise la même méthode de traversée de l'arbre : un visiteur récursif discuté plus haut dans la section 3.1.2.

3.1.4.3.1 ASTConsumer

Le consommateur d'action est une classe permettant d'encapsuler un visiteur d'arbre. C'est ce visiteur qui aura la charge de définir le comportement approprié au passage sur chacun des nœuds de l'arbre. Ainsi, pour générer une action, nous devons créer une classe concrète basée sur le visiteur abstrait `RecursiveASTVisitor`. Dans cette classe concrète, nous pouvons redéfinir les méthodes de traversée spécifique à un type de nœud. Une fois notre visiteur concret créé, nous devons créer une instance concrète de `ASTConsumer`, prenant comme paramètre notre visiteur nouvellement créé. En dernier lieu, nous créons une classe d'action à la compilation concrète (basée sur `FrontendAction`, permettant la création de notre consommateur personnalisé.

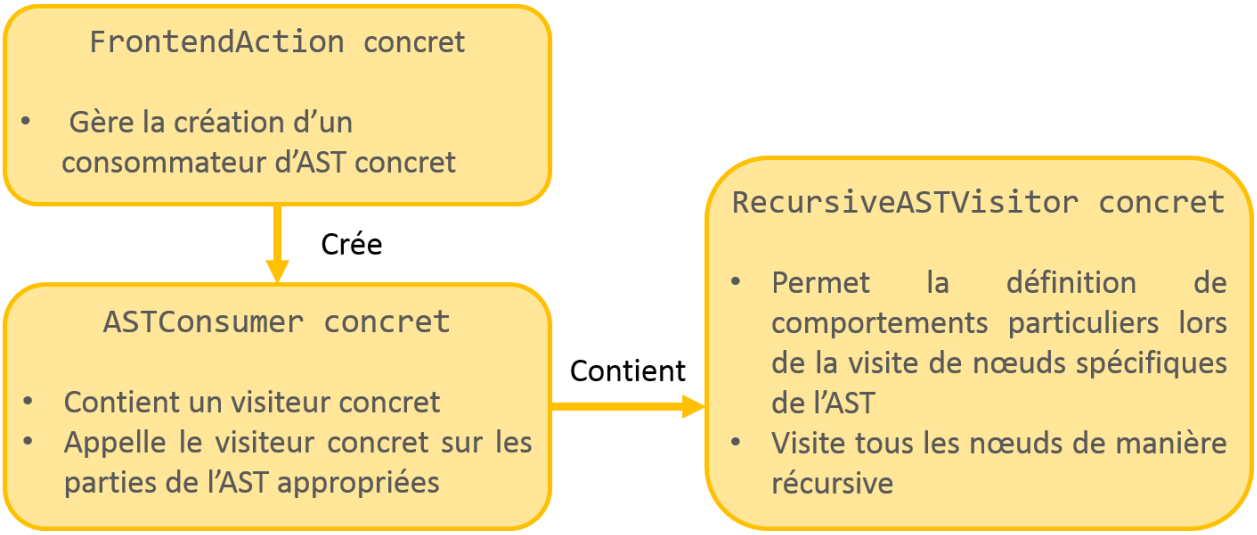


Figure 3.1-2 : Schéma simplifié de la solution de traversée basée sur le consommateur

3.1.4.3.2 ASTMatchers

La seconde méthode est basée sur l'utilisation de patrons de correspondance (appelés *matchers* dans la documentation). Ces patrons de correspondance sont créés en utilisant de manière astucieuse les concepts de fonctions variadiques et de programmation abstraite. L'implémentation prend la forme d'une série de fonctions représentant les différents types de nœuds possibles, comme la fonction de l'Extrait 3.1-4, qui va correspondre aux nœuds déclarant des fonctions. Ces fonctions permettent par composition de générer des descendants de la classe `Matcher`.

```
1  functionDecl()
```

Extrait 3.1-4 : Patrons de correspondance – Fonctions de correspondance

À ces fonctions de correspondance de nœud viennent s'ajouter deux autres catégories de fonctions : les fonctions de spécification et les fonctions de traversée. Les fonctions de spécification permettent de demander plus d'informations sur le nœud recherché par la fonction de correspondance. Par exemple, le patron de correspondance de l'extrait 3.1-5 retrouvera seulement les déclarations de fonctions qui possèdent deux arguments.

```
1  functionDecl(parameterCountIs(2))
```

Extrait 3.1-5 : Patrons de correspondance – Fonctions de spécification

Les fonctions de traversée permettent quant à elles de naviguer dans la descendance ou l'ascendance de la fonction de correspondance initiale. Le patron présenté à l'Extrait 3.1-6 illustre l'utilité de telles fonctions. Dans cet exemple, nous rajoutons à notre patron précédent une condition sur la descendance de la déclaration de fonction : les paramètres de la fonction. Nous spécifions que la correspondance n'aura lieu que si l'un des paramètres est de type entier.

```
1  functionDecl(
1      parameterCountIs(2),
2      hasAnyParameter(
3          (hasType(asString(«int»))
4      ))
```

Extrait 3.1-6 : Patrons de correspondance – Fonctions de traversée

Comme nous avons pu le voir au travers de ces exemples, la composition s'effectue de manière simple et aisément compréhensible. Les noms de fonctions tournés en langage naturel rendent la lecture d'un patron aussi simple qu'une description de celui-ci. À l'aide de ces trois familles de fonctions, il est possible de créer des recherches très complexes dans l'arbre de syntaxe.

Une fois un patron créé, il reste encore quelques éléments structurels à mettre en place. Tout d'abord, il est important de définir des liens sur les nœuds de notre patron, afin de pouvoir agir sur ceux-ci. Cela est achevé grâce à une méthode d'association définie dans l'objet renvoyé par chacune des fonctions présentées plus haut. Dans l'extrait 3.1-7, nous créons deux associations : une sur le nœud de déclaration de fonction et une sur le paramètre de type entier, que nous identifions respectivement avec les chaînes de caractères « `fDecl` » et « `intParam` ». Par la suite, quand nous devrons analyser nos résultats, nous pourrons nous référer à ces identifiants pour accéder aux différents nœuds.

```
2  functionDecl(
3      parameterCountIs(2),
4      hasAnyParameter(
5          (hasType(asString("int"))
6      ).bind("intParam")
7  ).bind("fDecl")
```

Extrait 3.1-7 : Patrons de correspondance – Méthodes d'association

Nous devons ensuite mettre en place la structure permettant l'utilisation de ce patron. La figure 3.1-3 illustre cette structure. Un objet `MatchFinder` sert de gestionnaire pour les différents patrons

de correspondance que nous voulons confronter à notre AST. Lors de l'ajout d'un patron à l'aide de la méthode `addMatcher()`, il faut aussi fournir une instance de la classe `MatchCallback`. Cette classe abstraite permet de définir l'action à entreprendre en cas de correspondance entre le patron et une partie de l'arbre. C'est dans cette classe enfant que nous pouvons utiliser les identifiants déclarés dans le patron pour accéder aux nœuds de l'AST. Une fois une paire `Matcher/MatchCallback` ajoutée au gestionnaire, on peut passer notre gestionnaire à l'outil de compilation `ClangTool`. Le gestionnaire va générer automatiquement la structure consommateur/visiteur vue dans la section précédente en utilisant les informations disponibles dans les patrons de correspondance. Contrairement à la première solution présentée, nous n'avons pas besoin de créer une classe dérivée de `FrontendAction`, car le `MatchFinder` implémente la fonction `createASTConsumer()` nécessaire au bon fonctionnement de la classe `ClangTool`. À chaque fois qu'une correspondance est repérée, le visiteur appelle la méthode `run()` de l'objet `MatchCallback` associé. L'intérêt de ce système repose dans sa simplicité d'utilisation et dans la puissance des patrons de correspondance, qui permettent de rechercher des éléments très précis dans l'arbre.

3.1.4.4 Transformer la source

Maintenant que nous avons toute la structure nécessaire pour accéder à l'arbre, nous avons besoin d'outils pour modifier la source. Ces outils sont fournis par Clang dans la classe `Rewriter`. Comme son nom l'indique, cette classe implémente des méthodes pour simplifier la transformation du code fourni en source à Clang. Pour ce faire, le `Rewriter` maintient une référence à la classe de contrôle des sources, le `SourceManager`. Cette classe de contrôle permet de renvoyer les tampons contenant les données des sources. Lorsqu'une modification est demandée (par exemple la suppression d'une ligne à l'emplacement `SourceLoc`), la classe de réécriture va demander au contrôleur de sources l'identité du fichier et le décalage dans le tampon correspondant à la position fournie. Si ce fichier n'avait pas encore été modifié, le Rewriter va copier le tampon original du fichier dans une classe interne `RewriteBuffer`. Cette classe a pour but d'enregistrer les modifications demandées par l'utilisateur tout en conservant un mappage entre les positions originales dans la source et dans le fichier modifié.

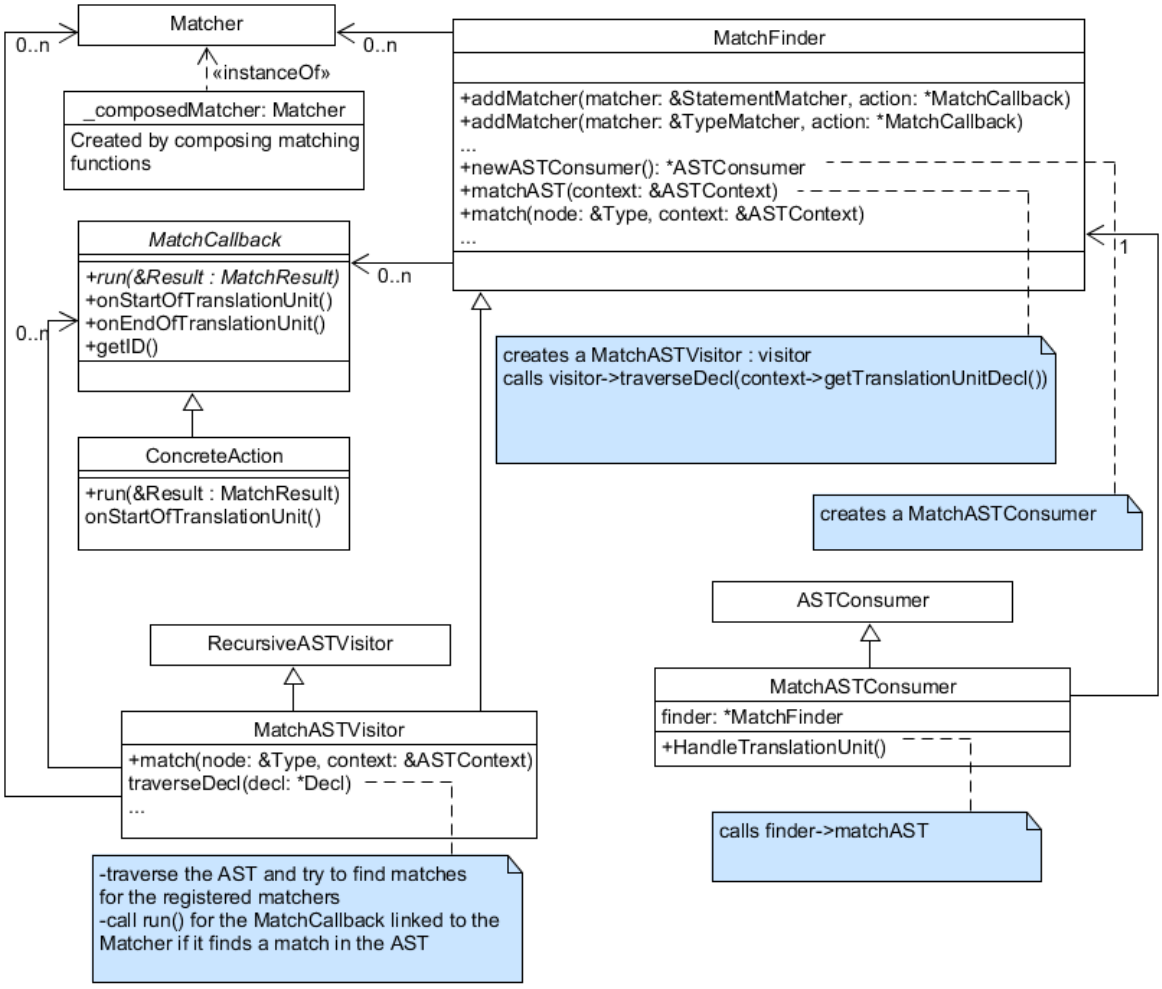
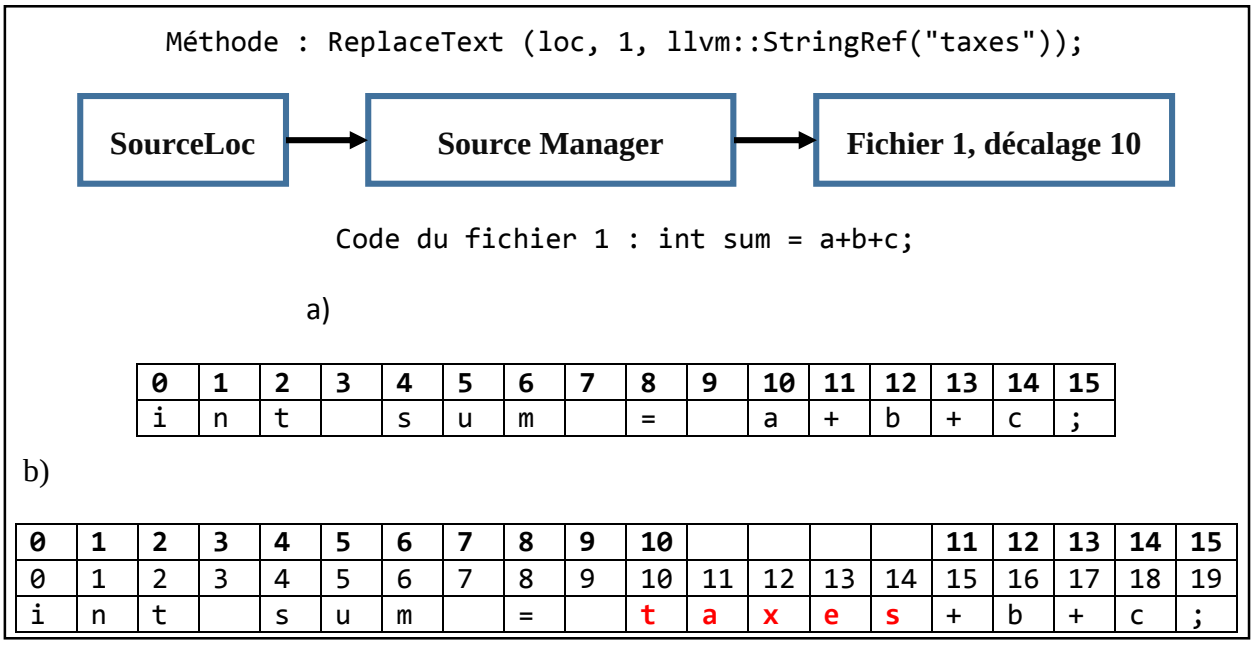


Figure 3.1-3 : UML simplifié de l'architecture des patrons de correspondance

Imaginons que nous voulons remplacer la variable `a` dans un ensemble de code par un autre nom de variable, par exemple `taxes`. À l'aide d'un patron de correspondance, nous avons récupéré les `sourceLocation` de toutes les occurrences de `a` dans le code. Nous demandons alors au `Rewriter` de remplacer ces occurrences par le nouveau nom de variable à l'aide de la méthode `ReplaceText()`. Le déroulement est illustré dans la figure ci-dessous. Le rewriter va solliciter le contrôleur de sources pour convertir chaque `sourceLocation` en un identifiant de fichier et un décalage dans celui-ci. Une fois le fichier et le décalage déterminés, le `Rewriter` va créer un tampon de réécriture en recopiant l'intégralité du tampon source du fichier 1. Puis il va procéder à la modification du texte à la position 10 tout en décalant les caractères présents à droite de la

position 10. Pendant ces opérations, le tampon de réécriture garde en mémoire la correspondance avec la source non modifiée (en gras dans la figure ci-dessous).



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	n	t		s	u	m		=		a	+	b	+	c	;

0	1	2	3	4	5	6	7	8	9	10					11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	n	t		s	u	m		=		t	a	x	e	s	+	b	+	c	;

Figure 3.1-4 : Illustration du fonctionnement interne de la classe Rewriter. a) tampon de la source du fichier 1. b) tampon de réécriture après la méthode de modification de la source.

Une fois les modifications effectuées, la classe `Rewriter` fournit des méthodes permettant l'enregistrement des modifications dans le tampon source ou la récupération de tout ou partie du tampon de réécriture en utilisant le système de `SourceLoc` du fichier original.

L'explication des outils de réécriture clôt cette section d'introduction à l'outil Clang. Nous avons pu voir qu'un grand nombre de solutions nous sont fournies dans le cadre de la transformation de code source. Dans la section suivante, nous expliquons comment nous avons mis en œuvre ces outils afin de créer C2Space.

3.2 Présentation de C2Space

Le projet de création de C2Space tire son origine d'un besoin dans le domaine du design de systèmes embarqués. Beaucoup d'entreprises possèdent une base d'algorithmes codés dans des langages haut niveau. Parfois, afin d'augmenter la performance de ces algorithmes pour rentrer dans le cadre d'une application précise, il est nécessaire de rajouter de la puissance de calcul. Dans cette optique, une des meilleures solutions est une plateforme hétérogène basée sur un FPGA. Cette solution permet une bonne amélioration des performances ainsi qu'une phase de développement

assez rapide et peu couteuse. Toutefois, le développement d'un tel système peut s'avérer complexe si l'équipe manque d'expérience. Nous avons donc décidé de créer un outil permettant, le plus facilement possible, de transformer un code écrit dans un langage haut niveau en une spécification pour plateforme FPGA. Pour ce faire, nous voulons créer un outil qui choisirait automatiquement la meilleure solution de partitionnement des tâches à partir d'une analyse du code. Cet automate doit générer une solution pour un logiciel de design ESL, afin que l'exploration du design puisse se poursuivre.

La motivation originale est donc de créer un programme de traduction entièrement automatisé. Toutefois, nous nous sommes vite rendu compte qu'une telle solution n'était pas envisageable. En effet, le langage d'entrée que nous avons décidé de choisir, le C, manque d'information. Les choix de design d'un développeur système sont motivés par des éléments externes au code source lui-même (par exemple, la façon de faire transiter les données, les possibilités de concurrence ou encore la possibilité de pipeliner un processus). De par la nature séquentielle du C, ce genre d'information ne peut être extraite facilement d'une analyse statique de la source. Nous avons donc décidé de recibler l'outil comme un assistant à la réécriture du code. Cet outil permet, à l'aide de quelques informations données par l'utilisateur, de réorganiser le code source afin de créer une solution pour le logiciel ESL SpaceStudio.

C2Space cherche à améliorer les performances d'un code source en se concentrant sur la création d'un ou plusieurs coprocesseurs matériels afin d'accélérer les parties les plus intensives en calcul de l'algorithme original et de libérer du temps processeur pour d'autres tâches. Cette approche rend C2Space particulièrement adapté aux algorithmes orientés données. Afin de repérer ces parties intensives en calcul, nous avons décidé de nous appuyer sur la solution fournie par la société Vector Fabrics, Pareon Profile, qui nous permet d'analyser le potentiel de parallélisme logiciel d'un code. La logique qui motive cette approche est que, si nous sommes capables d'aller chercher une accélération conséquente sur un processeur multicœur sur une portion assez importante du temps d'exécution du code, nous devrions être capable, en créant un coprocesseur adapté, d'aller chercher des accélérations encore plus importantes. Pareon nous permet d'obtenir facilement ce genre d'informations à l'aide d'une interface graphique. Celle-ci décrit les portions de code occupant le plus du temps processeur, ainsi que les dépendances qui pourraient ou non empêcher une parallélisation massive d'un noyau de calcul. Ces informations sont utilisées pour guider C2Space dans la segmentation du code source C en modules SpaceStudio. La segmentation peut s'effectuer

au niveau de deux structures C : les fonctions et les boucles. Ce choix est motivé par trois raisons. Dans un premier temps, cela permet de se prévenir dans une certaine mesure de problèmes de portée de variable lors de la traduction. Dans un second temps, la majorité du potentiel de parallélisme des algorithmes orientés données se trouve à l'intérieur des boucles. Enfin, les outils de synthèse haut niveau ciblent aussi ces structures. Prendre en charge ces deux types nous semble donc satisfaisant.

Les prochaines sections ont pour but d'expliquer plus en détail le fonctionnement de C2Space. Nous commencerons par expliquer les fonctionnalités de l'outil et les problématiques de la traduction. Nous continuerons avec une brève liste des contraintes sur le format du code source. Nous présenterons finalement ensuite l'implémentation interne de l'outil, en justifiant certains choix de conception.

3.2.1 Capacités de l'outil

C2Space prend en charge la création d'un projet SpaceStudio à partir d'une source écrite en C séquentiel. Afin de réaliser ceci, l'outil passe par plusieurs étapes.

En premier lieu, l'utilisateur est sollicité pour définir quels sont les éléments du code source qu'il désire modulariser. L'utilisateur doit alors entrer le nom des fonctions retenues dans l'analyse effectuée par Pareon. Comme le modèle SpaceStudio a besoin d'un module en tant que point d'entrée du code, la fonction `main()` est automatiquement ajoutée à cette liste.

Par la suite, l'outil va effectuer plusieurs passes sur le code source à l'aide des outils fournis par Clang. Ces passes ont pour but de récupérer de l'information sur le code source afin de guider la réécriture de celui-ci. Pendant la collecte d'informations, certaines informations complémentaires sont demandées à l'utilisateur, comme le sens de transit des données ciblées par pointeurs passés en paramètres d'une fonction à modulariser. C'est aussi pendant cette phase que l'outil va demander à l'utilisateur s'il préfère modulariser juste un corps de boucle contenu dans une des fonctions de la liste précédente (afin de scinder la fonction en deux modules distincts). L'intégralité de ces informations est collectée à l'intérieur de classes dédiées. Les instances de ces classes sont accessibles à l'aide de classes gestionnaires qui ne peuvent être instanciées qu'une seule fois (patron *Singleton*).

Lors de la troisième phase, les informations emmagasinées jusqu'ici sont consommées par des classes spécialisées dans la construction des modules SpaceStudio. Ces classes utilisent la classe Clang `Rewriter` pour gérer la réécriture du code. Plusieurs constructeurs différents existent selon le type de module à générer (module standard, module scindé, module boucle). De par l'architecture des classes constructrices, l'ajout de nouveaux constructeurs est très simple. Ces classes profitent d'une dernière passe sur l'unité de compilation afin d'accéder à la source. Un nouveau fichier est alors construit sur une structure de module standard dans laquelle nous venons ajouter le code nécessaire à la préservation de la fonctionnalité. Les interactions entre les modules sont générées automatiquement en suivant l'interface de programmation de SpaceStudio (*Read/Write*) et les informations récupérées lors de la phase deux. Un gestionnaire de mémoire répertorie les différentes données pouvant être stockées lors de l'implémentation et génère une table de décalage basique permettant de mapper ces données aux adresses correspondantes dans l'élément mémoire. Enfin, un fichier de script Python est automatiquement créé. Ce script permet la création de la solution SpaceStudio avec les modules que nous avons créés ainsi que les éléments mémoire requis.

Nous mettons à disposition un système de sauvegarde de configuration sommaire dans l'outil. Une fois que l'utilisateur a complété le processus, les informations qu'il a fournies sont enregistrées dans un fichier de configuration. Au démarrage de l'outil, il est alors possible de sélectionner ce fichier de configuration pour réinsérer les informations automatiquement sans avoir à reconfigurer C2Space.

Une fois la procédure complète effectuée, l'utilisateur peut continuer le développement depuis la plateforme SpaceStudio. Il peut alors simplement choisir de mener le partitionnement à terme avec peu de modifications ou procéder à une exploration architecturale plus exhaustive.

La Figure 3.2-1 résume l'architecture interne de C2Space.

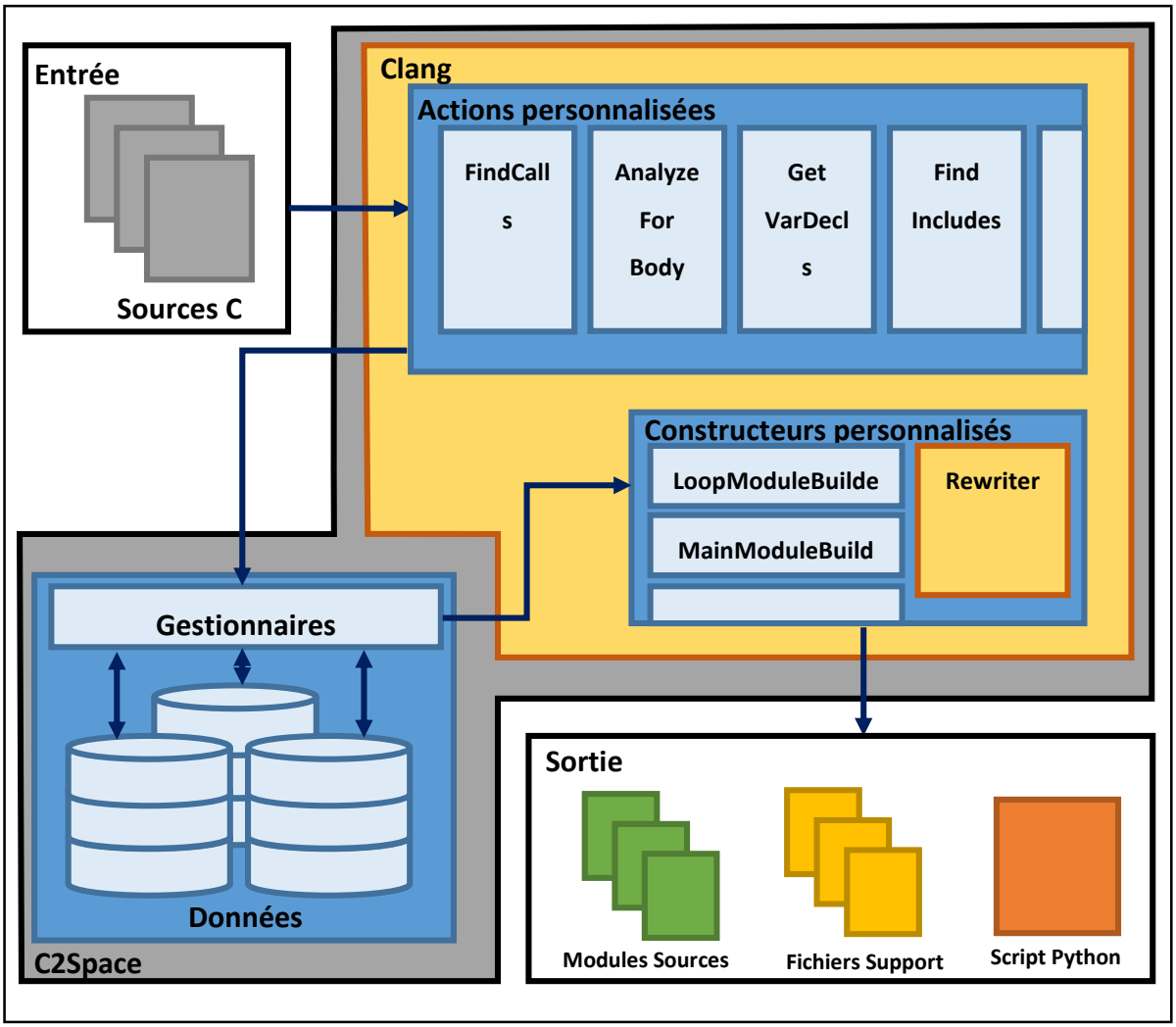


Figure 3.2-1 : Organisation de CtoSpace

3.2.2 Problématiques de la traduction

Dans une optique de traduction de code C vers des modules SpaceStudio, plusieurs problématiques apparaissent. Ces problématiques sont présentées dans cette section à l'aide d'exemples sur des extraits de code triviaux. La manière dont ces problématiques ont été relevées sera expliquée dans la section 3.2.4.

3.2.2.1 Nomenclature

Avant toute chose, il est important de définir quelques concepts simples qui seront utilisés lors de cette section. Le graphe d'appel suivant sert d'exemple :

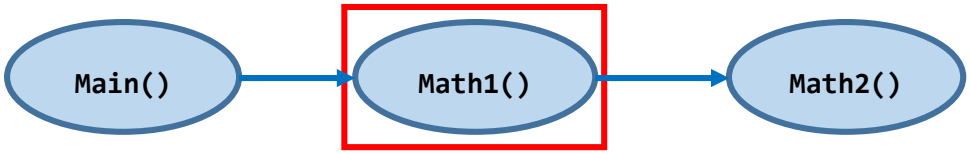


Figure 3.2-2 : Exemple de graphe d'appel

- **Fonction analysée** : Dans le cadre de la modularisation d'une fonction, la fonction analysée est la fonction sur laquelle nous travaillons, pour laquelle nous sommes en train de construire le module. Dans la figure 3.2-2, il s'agit de la fonction `Math1()`.
- **Fonction appelante** : Il s'agit des fonctions dans lesquelles la fonction analysée est appelée (ici `Main()`).
- **Fonction appelée** : Il s'agit des fonctions auxquelles notre fonction analysée a directement recours lors de son exécution. Dans la figure ci-dessus, il s'agit de la fonction `Math2()`. Si jamais la fonction `Math2()` avait recours à une autre fonction, on ne considérerait pas cette autre fonction comme fonction appelée par `Math1()`.
- **Fonction destinée à être modularisée** : Lors de cette section, il sera important de bien faire la différence entre les fonctions qui sont destinées à être modularisées et les fonctions qui vont rester telles quelles. Lorsque nous parlerons d'une fonction destinée à la modularisation, nous utiliserons l'acronyme DAM.

De la même manière, nous utilisons les termes **module appelant**, **appelé** et **analysé**.

3.2.2.2 D'une fonction à un module

Pour bien comprendre les enjeux de l'outil C2Space, il est important de comprendre les transformations que subit le code source dans le cas le plus simple, c'est-à-dire transformer une fonction en module. Prenons comme exemples les deux fonctions présentées à l'Extrait 3.2-1 :

```c
int math(int a, int b) {
  a = a + b;
  b = b * b;
  a = a - b;
  return a;
}
```

```c
int main() {
  int intA, intB, intC;
  intA = 1;
  intB = 2;
  intC = math(intA, intB);
  fprintf("result %d", intC);
  return 0;
}
```

Extrait 3.2-1 : Deux fonctions à modulariser

La fonction `main()` instancie quelques variables, puis fait appel à la fonction `math()` qui effectue des opérations mathématiques sur les variables `a` et `b`. Comme nous l'avons dit plus haut, la fonction `main()`, qui représente le point d'entrée du programme, va devoir être modularisée. Afin de tirer parti des capacités de SpaceStudio, nous désirons aussi modulariser `math()` afin d'obtenir deux modules intercommunicants.

Contrairement aux fonctions présentées ici, un module SpaceStudio est implémenté comme une classe C++ dérivant de la classe `SpaceBaseModule`, celle-ci étant une spécialisation de la classe `sc_module` de SystemC. Nous avons donc commencé par imaginer un modèle générique de module SpaceStudio sur lequel nous nous appuierons pour réaliser la transformation d'une fonction vers un module. Le fichier d'entête de ce modèle générique est présenté dans l'Extrait 3.2-2.

Comme nous pouvons le voir, un module générique basé sur une fonction possède un nombre limité de méthodes. Pour plus de lisibilité, nous avons mis en caractères gras et soulignés les parties destinées à être modifiées par nos constructeurs selon le contenu de la fonction analysée. Ainsi, les informations nécessaires à la bonne génération de l'entête sont les suivantes :

- Le nom de la fonction analysée : afin de renseigner les champs `NOM_MODULE`.
- Les noms et les types des paramètres et du retour de la fonction analysée : pour remplacer les champs `ArgType` et `argName`, mais aussi pour instancier les fonctions de communications `send_` et `read_`.

- Le nom des fonctions appelantes DAM et appelées DAM : afin de générer les fonctions de communication correspondantes en remplaçant les champs `NOM_MODULE_EXT`.

```
1  // ********************************************************************
2  // Generic standardModule
3  // ********************************************************************
4  #ifndef NOM_MODULE_H
5  #define NOM_MODULE_H
6  #include "systemc.h"
7  #include "SpaceBaseModule.h"
8  #include "ApplicationDefinitions.h"
9  class NOM_MODULE : public SpaceBaseModule{
10
11   public:
12   SC_HAS_PROCESS(NOM_MODULE);
13   /// Constructor
14   NOM_MODULE(sc_module_name zName, double dClockPeriod, sc_time_unit Unit,
   unsigned char ucID, unsigned char ucPriority, bool bVerbose);
15   /// Methods
16   void thread(void);
17
18   private:
19   /// Methods
20   void action();
21   ArgType read_ArgType_from_NOM_MODULE_EXT();
22   void read_ArgType_from_NOM_MODULE_EXT(type *data, int size);
23   void send_ArgType_to_NOM_MODULE_EXT(ArgType data);
1    void send_ArgType_to_NOM_MODULE_EXT(type *data, int size);
24   ArgType argName1;
25   ArgType argName2;
26 };
27 #endif
```

Extrait 3.2-2 : Module SpaceStudio générique minimal (entête)

Dans le cas de l'exemple de l'extrait 3.2-1, si nous analysions l'entête du module généré à partir de la fonction `math()`, nous aurions à instancier les fonctions de communication `read_int_from_main()` et `send_int_to_main()`. Le nombre de fonctions de communication peut vite augmenter si la fonction analysée est appelée par beaucoup d'autres fonctions et que celle-ci possède des types différents au sein de ses paramètres.

Nous pouvons passer à l'implémentation des méthodes du modèle générique. L'extrait suivant nous montre la méthode `thread()` générique, indispensable à chacun des modules.

```
1  void NOM_MODULE::thread(void){
2    while (1){
3      //Read the operands
4      argName = readOperand();
5      //Perform Action
6      action();
7    }
8  }
```

Extrait 3.2-3 : thread générique

Cette fonction représente le cœur du module et la manière dont celui-ci va se conduire. Dans le cas du passage d'une fonction vers un module, nous mettons en place dans la boucle infinie une suite d'opérations simples : les opérandes nécessaires à la réalisation de la fonction analysée sont d'abord lues à l'aide des fonctions de communications instanciées automatiquement. Ces fonctions sont bloquantes. Par la suite, nous effectuons un appel à la méthode `action()`, qui contient la fonctionnalité originale de la fonction analysée. Si cette fonction renvoyait un résultat, celui-ci est pris en charge dans le corps de la fonction analysée, et le processus peut recommencer.

Passons maintenant à l'implémentation de la fonctionnalité dans la méthode `action()`.

```
1  void NOM_MODULE::action(){
2    argName1 = argName1 * argName2;
3    sendOperand(argName1);
4  }
```

Extrait 3.2-4: méthode action générique

La méthode `action()` encapsule le comportement de la fonction analysée. Dans les cas les plus simples, nous recopions simplement le corps de la fonction analysée dans cette méthode. Toutefois, certains cas peuvent être plus complexes. Si la fonction originale contenait un retour, celui-ci est remplacé par un appel à la fonction d'envoi d'opérande. Dans le cas où la fonction analysée contient des appels à d'autres fonctions DAM, d'autres transformations sont nécessaires. Considérons le cas de l'extrait 3.2-5, et observons l'action du module analysé `main`. On peut voir que, dans le corps de `main()`, un appel à `math()`, autre fonction DAM est effectué. L'extrait de code suivant montre comment est transformée la fonction analysée dans la méthode `action()` de son module.

```c
//Original analyzed function
int main() {
  int intA, intB, intC;
  intA = 1;
  intB = 2;
  intC = math(intA, intB); //DAM function
  fprintf("result %d", intC);
  return 0;
}
```

```c
//C2Space generated method
void main::action()
{
  int intA, intB, intC;
  intA = 1;
  intB = 2;
  send_int_to_math(intA);
  send_int_to_math(intB);
  intC = read_int_from_math(); //Replace
  fprintf("result %d", intC);
}
```

Extrait 3.2-5 : Interaction Fonction analysée/Fonction DAM appelée

Le fait d'avoir un appel à une fonction DAM force la modification de la source pour correspondre au protocole de communication du module standard. Ainsi, plutôt que d'effectuer un appel standard, nous remplaçons l'appel par une série d'envois de paramètre et un retour via les fonctions `read` et `send` du module analysé. Cela requiert d'avoir des informations supplémentaires sur le code source :

- Le nom des paramètres de la fonction appelée, associé à leur type respectif, pour utiliser la bonne fonction d'envoi sur la bonne variable.
- La position de l'appel dans la source, afin de pouvoir remplacer la portion de code qui nous intéresse
- Les positions de début et de fin du corps de la fonction analysée afin de pouvoir recopier celle-ci dans notre module.

Les dernières méthodes de l'entête générique de l'extrait 3.2-2 sont les fonctions d'envoi et de récupération de données. L'Extrait 3.2-6 résume comment celles-ci sont construites :

```
2  void NOM_MODULE::send_type_to_NOM_MODULE_EXT(type data){
3    ModuleWrite(NOM_MODULE_EXT_ID, SPACE_BLOCKING, &data);
4  }
5  void NOM_MODULE::send_type_to_NOM_MODULE_EXT(type *data, int size){
6    ModuleWrite(NOM_MODULE_EXT_ID, SPACE_BLOCKING, data, size);
7  }
8  type NOM_MODULE::read_type_from_NOM_MODULE_EXT(){
9    type data;
10   ModuleRead(NOM_MODULE_EXT_ID, SPACE_BLOCKING, &data);
11   return data;
12 }
13 void NOM_MODULE::read_type_from_NOM_MODULE_EXT(type *data, int size){
14   type data;
15   ModuleRead(NOM_MODULE_EXT_ID, SPACE_BLOCKING, data, size);
16   return data;
17 }
```

Extrait 3.2-6 : méthodes send et read génériques

Ces méthodes servent en réalité d'adaptateurs aux fonctions de SpaceStudio, `ModuleWrite()` et `ModuleRead()`. Ces quatre fonctions reçoivent ou envoient la ou les données présentes dans `data`. On peut choisir de rendre ces envois et réceptions bloquants ou non, bien que dans notre cas, nous rendons toutes ces méthodes bloquantes afin d'éviter de rajouter de la synchronisation. Le dernier paramètre de ces fonctions est l'identifiant du module dans le système de SpaceStudio, qui est créé automatiquement à partir du nom du module mis en majuscules. La création de ces modules nécessite encore une fois les informations de communications entre la fonction analysée et les fonctions appelantes DAM et appelées DAM.

3.2.2.3 Multiples modules appelants

Comme nous avons pu le voir dans la section précédente, la forme générique de module que nous proposons reste assez simple. Pourtant, sous certaines conditions, celle-ci peut se retrouver complexifiée. C'est le cas lorsque la fonction analysée peut être appelée par de multiples fonctions appelantes DAM. Regardons le graphe d'appel de la figure 3.2-3 :

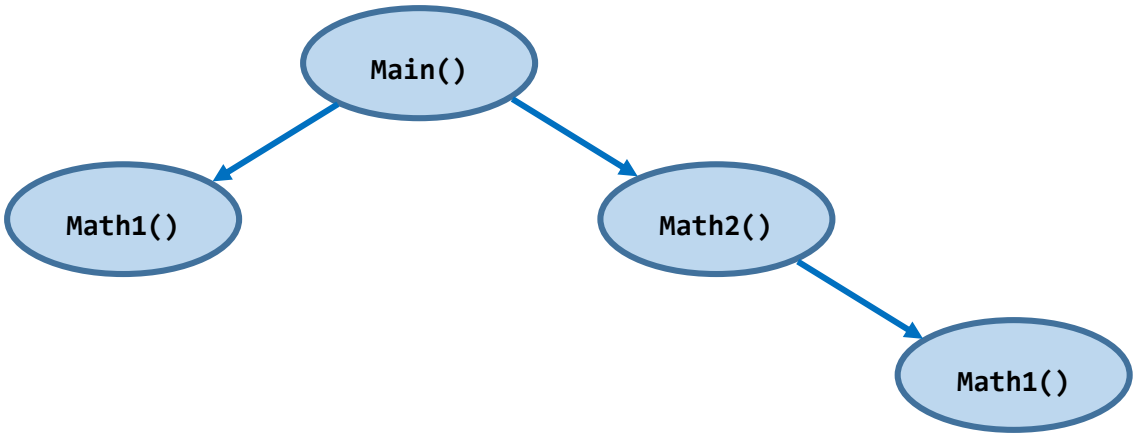


Figure 3.2-3: Graphe d'appel pour multiples appelants

Pour l'exemple, nous considérons que toutes les fonctions sont à modulariser. Nous nous apercevons que la fonction `Math1()` possède deux fonctions appelantes DAM. Une fois que les fonctions sont transformées en modules SpaceStudio, il est nécessaire de savoir quel module appelant est actuellement en train de communiquer avec le module analysé. Cela est dû à la structure de l'API de communication de SpaceStudio (`ModuleRead`/`ModuleWrite` que nous avons dans la section 3.2.2.2), qui demande un identifiant pour le destinataire. Nous avons donc imaginé un modèle générique amélioré dans le cas où il existe plusieurs modules appelants. Ce modèle est présenté dans l'Extrait 3.2-7, où les noms des fonctions présentées dans la figure 3.2-3 sont utilisés.

Dans le cas où plusieurs modules appellent le module analysé, le code du `thread()` change. Une nouvelle boucle infinie est mise en place pour effectuer une première phase de recherche d'interlocuteur. Cette recherche est effectuée à l'aide de méthodes spécialement instanciées : `try_to_begin_with_...()`. Ces méthodes utilisent la version non bloquante de l'API de SpaceStudio et renvoient `true` si la fonction a bien réussi à communiquer avec le module appelant en question. Une fois qu'un des modules appelants a engagé la communication, la variable `interactor` est initialisée à une valeur spécifique. Cette valeur est ensuite confrontée dans un `switch` pour amorcer le protocole classique que nous avons vu dans l'extrait 3.2-3 avec l'interlocuteur adéquat.

```
1  void Math1::thread(void) {
2    int interactor = 0;
3    while (1) {
4      //Searching for interactor
5      while (1) {
6        if (try_to_begin_with_Main()) {
7          interactor = 1;
8          break;
9        }
10       if (try_to_begin_with_Math2()) {
11         interactor = 2;
12         break;
13       }
14     }
15
16     switch(interactor) {
17       case 1:
18         argName1 = read_int_from_Main();
19         action();
20         break;
21       case 2:
22         argName1 = read_int_from_Math2();
23         action();
24         break
25     }
26   }
27 }
```

Extrait 3.2-7: Exemple de thread avec plusieurs modules appelants

Du côté des modules appelants, une méthode supplémentaire est instanciée, `start_communication_with_...()`, qui est un envoi bloquant. Cette méthode est utilisée lors du remplacement de l'appel à la fonction analysée avant d'envoyer les paramètres de celle-ci. Ainsi, si le module analysé est déjà occupé par l'autre module appelant, le premier module appelant devra attendre son tour.

3.2.2.4 Le problème des pointeurs

Un des aspects du langage C est l'utilisation des pointeurs de données afin de faire transiter des données au travers du programme. Toutefois, l'idée même de pointeurs est compliquée à transcrire dans un système matériel où plusieurs éléments bien séparés peuvent communiquer. En effet, lorsque les fonctionnalités sont partitionnées sur différents processeurs/coprocesseurs, le passage d'une adresse mémoire n'est pas toujours possible, et il est nécessaire d'envoyer l'intégralité des données référencées par le pointeur directement vers la mémoire interne du coprocesseur ou bien

vers une mémoire externe partagée. Nous avons décidé de concentrer nos efforts sur les fonctions DAM contenant des pointeurs en tant que paramètres.

Nous effectuons alors un postulat : l'utilisation de pointeurs de données dans un appel de fonction est motivée par l'envoi simultané d'une quantité importante de données. Nous considérons donc que l'utilisation de pointeurs signifie la présence d'un traitement itératif dans la fonction analysée. Ainsi, les pointeurs peuvent être considérés comme des tables de données qui peuvent soit être modifiées par la fonction appelante (auquel cas il s'agit de données sortantes), soit être utilisées en tant que données pour un calcul (il s'agit alors de données entrantes). Il est aussi possible qu'un seul pointeur représente un ensemble de données à la fois entrantes et sortantes.

Ce postulat souligne aussi l'utilisation probable d'une ou plusieurs boucles à l'intérieur même de la fonction. Cela est intéressant, car généralement, les plus grandes accélérations dans un algorithme orienté données sont obtenues par la parallélisation des boucles.

Nous avons donc décidé de développer un autre modèle de module destiné à prendre en compte ces paramètres pointeurs et à les envoyer vers une mémoire externe. Cette mémoire externe est ensuite lue ou écrite par les modules qui communiquent. Cela permet un accès partagé à des données. Nous prévenons les problèmes de concurrence en synchronisant les deux modules. Les extraits suivants passent en revue les points principaux de ce nouveau modèle.

```
1  class NOM_MODULE : public SpacebaseModule {
2    public:
3      SC_HAS_PROCESS(NOM_MODULE);
4      Module_convolutionAbs(sc_module_name zName, double dClockPeriod,
   sc_time_unit Unit, unsigned char ucID, unsigned char ucPriority, bool
   bVerbose);
5      void thread();
6    private:
7      void action(); /* former parameters : int a, int *b  ...*/
8      ArgType read_ArgType_from_NOM_MODULE_EXT();
9      void send_ArgType_to_NOM_MODULE_EXT(ArgType data);
10     void send_ack_to_MODULE_NAME_EXT(int ack);
11     void wait_ack_from_MODULE_NAME_EXT);
12     /// Members
13     int a;
14 }
```

Extrait 3.2-8 : Définition de la classe Module à boucle

Dans ce premier extrait, nous voyons le modèle imaginé pour le fichier d'entête. Les différences par rapport au modèle standard de l'extrait 3.2-2 sont affichées en caractères gras. Les paramètres

pointeurs ne sont pas enregistrés en tant que variables membres et sont traités via une autre interface. De plus, deux nouvelles fonctions de communications bloquantes `send_ack_to_()` et `wait_ack_from_()` sont déclarées. Celles-ci permettent de synchroniser le module analysé et le module appelé pour éviter la corruption des données partagées en mémoire. Analysons maintenant les changements dans les fichiers .cpp des modules.

```
1  //Original Analyzed Function
2  void math(int *table1Out, int *table2In, int factor) {
3    for (int i = 0; i < 3; ++i) {
4      for (int j = 0; j < 3; ++j) {
5        table1Out[i * 3 + j] = (table2In[j * 3 + i] + table2In[0]) * factor;
6      }
7    }
8  }
```

```
1  //Transformed Function
2  void math::action() { //former call : math(int *table1Out, int *table2In,
   int factor)
3    for (int i = 0; i < 3; ++i) {
4      for (int j = 0; j < 3; ++j) {
5        write_int_to_memory_0((MATH_TABLE1OUT_TBL_OFFSET + i * 3 + j,
   (read_int_from_memory_0(MATH_TABLE2IN_TBL_OFFSET + j * 3 + i) +
   read_int_from_memory_0(MATH_TABLE2IN_TBL_OFFSET + 0)) * factor, 1);
6      }
7    }
8  }
```

Extrait 3.2-9 : Méthode d'action d'un Module à boucle

Dans ce nouvel extrait de code, nous voyons les transformations que doit subir la fonction analysée lorsque des données externes issues d'un pointeur sont accédées. Tout d'abord, chaque paramètre pointeur de fonction analysée DAM se voit attribué un décalage (ex : `MATH_TABLE1OUT_TBL_OFFSET`). Par la suite, chaque accès à des tables dans le code de la fonction se voit remplacé par une des deux fonctions suivantes :

- `read_int_from_memory_x()` qui permet d'accéder à une ou plusieurs données en mémoire en précisant le décalage dans cette mémoire, et
- `write_int_from_memory_x()` qui permet d'écrire une ou plusieurs données dans la mémoire en précisant : 1) à partir d'où écrire et 2) combien de données l'on doit transférer. Il faut aussi préciser quelle valeur écrire en mémoire.

Dans cet exemple, nous remplaçons l'écriture dans la `table1Out` par une opération d'écriture en mémoire. Les deux variables accédées dans la `table2In` le sont via la fonction de lecture. Les

indices d'accès sont concaténés au décalage enregistré afin de trouver les données correspondantes. Pour pouvoir réaliser automatiquement cette transformation, il faut plusieurs données :

- Parmi les paramètres de type pointeur, il faut savoir quels pointeurs représentent des données entrantes, sortantes ou encore entrantes/sortantes.
- Il faut récupérer les positions des accès à ces paramètres de type pointeurs dans le corps de la fonction analysée.
- Il faut récupérer et analyser le contenu des indices afin de pouvoir les réutiliser lors de la reconstruction du code.
- Il faut générer les décalages correspondants à chaque table et les stocker pour réutilisation.

Une fois ces informations obtenues, nous pouvons générer le module analysé. Il faut maintenant remplacer les anciens appels à la fonction analysée par un nouveau protocole.

```
1  void main() {
2    int data1[9];
3    int data2[9] = { 1, 1, 1, 0, 0, 0, -1, -1, -1 };
4    math(&data1, &data2, 3); //Appel à une fonction DAM
5  }
```

```
1  void main::action() {
2  int data1[9];
3  int data2[9] = { 1, 1, 1, 0, 0, 0, -1, -1, -1 };
4  //////                AUTO REPLACE - Begin                //////
5  //Former function call : math(&data1, &data2, 3)
6  // In/inout tables loading:
7  write_int_to_memory_0(MODULE_MATH_TABLE2IN_TBL_OFFSET, &data2, 9);
8  //Protocol : We warn the module that data is ready
9  send_ack_to_math();
10 //Sending the parameters of the former function
11 send_int_to_math(3);
12 //Waiting for called module to finish
13 wait_ack_from_math();
14 //Retrieving out/inout Tables
15 read_int_from_memory_0(MODULE_MATH_TABLE2IN_TBL_OFFSET, &data1, 9);
16 //////                AUTO REPLACE - End                //////
17 }
```

Extrait 3.2-10 : Protocole de remplacement d'un appel d'une fonction DAM à boucles

On peut voir que le remplacement des appels est bien plus complexe dans le cas des modules à boucles. En effet, des étapes de synchronisation supplémentaires sont nécessaires afin d'empêcher la corruption des données envoyées dans la mémoire externe. À la ligne 7 du module, nous envoyons les paramètres pointeurs de type entrant vers la mémoire externe. Afin d'empêcher la

lecture de ces données par le module `math` avant que l'écriture ne soit complétée, nous envoyons un accusé d'envoi à la ligne 9. Nous envoyons ensuite de manière standard les paramètres non pointeurs (ici l'entier 3). Enfin, nous attendons un accusé de réception ligne 13 afin de pouvoir recharger les tables dans les variables déclarées comme sortantes.

Il est important de noter que les considérations de la section 3.2.2.3 sur les multiples modules appelants sont aussi bien prises en charge dans le cas des modules à boucles présentés dans cette section.

Bien que nous ayons présenté la méthode ci-dessus permettant de stocker les paramètres pointeurs dans une mémoire externe, cela n'a pas toujours d'intérêt dans le cadre d'un système hétérogène. Par exemple, la donnée représentée par le pointeur peut être trop petite pour justifier l'utilisation d'une mémoire externe. Nous permettons donc à l'utilisateur de choisir le type d'envoi de données pour chaque variable membre (via mémoire externe ou via fonctions de communication classiques). Si la communication classique est choisie, le paramètre est instancié en tant que variable membre et envoyé ou reçu selon son statut (entrant ou sortant). Pour ce type de paramètre, nous utilisons les fonctions permettant l'envoi ou la lecture de plusieurs données présentées dans l'extrait 3.2-6.

3.2.2.5 Réduction de la taille des noyaux de calcul

Une des fonctionnalités que nous désirons mettre en place dans C2Space est d'isoler des parties du code avec une granularité plus faible que la fonction C. Cela a pour but de permettre la modularisation de noyaux de calcul à taille limitée afin d'isoler seulement les parties de l'algorithme que nous souhaitons paralléliser massivement. Puisque les boucles sont les cibles principales de la recherche d'accélération, nous avons décidé d'inclure un second niveau possible de segmentation : la boucle elle-même. Le principe de cette segmentation est simple : une fois une fonction modularisée, l'utilisateur a la possibilité d'isoler un corps de boucle de cette fonction. Si un corps de boucle est sélectionné, nous faisons une analyse de toutes les variables utilisées dans la portée correspondante. Nous comparons cette liste aux variables déclarées dans la même portée. Les variables qui n'ont pas été déclarées deviennent une liste de paramètres pour une fonction nouvellement créée qui vient encapsuler le corps de boucle. Le corps de boucle dans la fonction

originale est remplacé par un appel à la fonction nouvellement créée avec envoi des paramètres nécessaires. La Figure 3.2-4 illustre ce procédé.

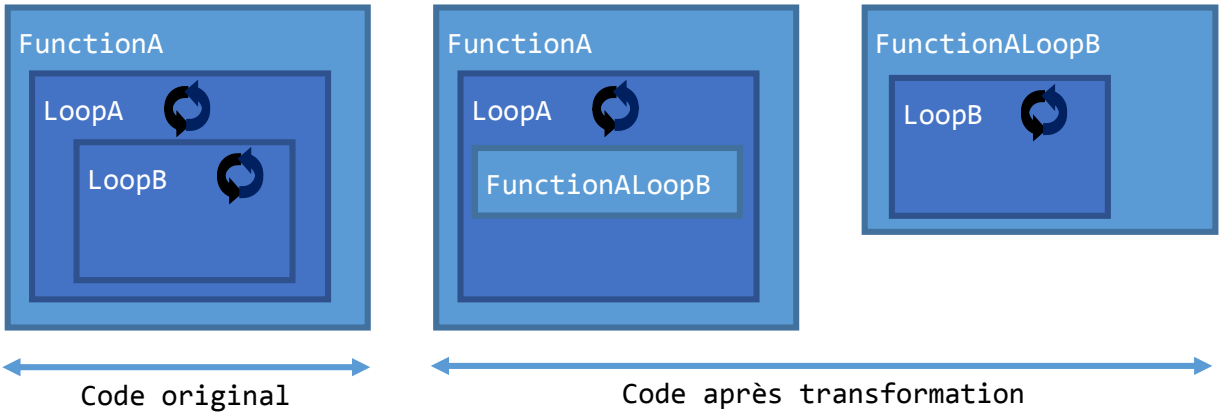


Figure 3.2-4 : Séparation d'un corps de boucle du reste de la fonction

Comme indiqué sur la Figure 3.2-4, le résultat est un code source avec une fonction supplémentaire. Cette nouvelle fonction est ensuite tout simplement rajoutée à la liste des fonctions DAM et subit le même traitement que les autres fonctions. Le résultat de cette transformation est un module supplémentaire encapsulant le comportement isolé par l'utilisateur. Cette transformation requiert cependant un grand contrôle sur le code source afin de récupérer les informations nécessaires :

- La liste des variables déclarées à l'intérieur de la boucle.
- La liste des variables utilisées dans le corps de boucle, leur type et leur utilisation (entrant, sortant).

Nous avons présenté les principaux défis que nous avons rencontrés lors de l'implémentation de C2Space et comment nous avons décidé de les résoudre. Cela nous a permis de dresser une liste des informations nécessaires pour que le traducteur puisse fonctionner.

3.2.3 Contraintes sur la source

Afin de rendre possible l'analyse de la source par notre outil, nous avons dû déterminer certaines contraintes sur le format de celle-ci. Nous avons essayé de rendre cette liste de contraintes la plus courte possible. Le code source entrant dans C2Space doit respecter les normes de C11 [36] à l'exception des règles suivantes :

- Lorsque l'on accède aux données référencées par un pointeur, il faut utiliser la notation "table" comme `a[0]`.
- Les indices d'une table doivent être exprimés sous la forme d'un tableau une dimension. C2Space ne prend pas actuellement en charge les tableaux à plus d'une dimension.
- Si une fonction possède un type de retour, elle doit être appelée depuis une assignation ou un calcul. Ex : `int a = func()` ou `b = a + func()`.
- Les fonctions destinées à être modularisées ne peuvent pas utiliser de variables globales.

3.2.4 L'architecture de C2Space

Maintenant que nous avons couvert le comportement global de C2Space, nous pouvons nous intéresser aux détails de l'implémentation. C2Space a été construit avec les concepts de la programmation orientée objet en tête. Nous avons essayé de rendre le code le plus compartimenté possible pour rendre plus aisées sa compréhension et son évolution. Les classes créées ont été séparées en quatre paquetages distincts remplissant des fonctions bien spécifiques au sein du programme : gestion, données, actions de compilation et reconstruction de la source. Les classes de Clang sont utilisées au sein de ces différents paquetages. Cette partie n'étant pas indispensable à la compréhension du mémoire, nous avons décidé de la rendre disponible en Annexe A.

CHAPITRE 4 EXPÉRIMENTATION DU FLOT : FILTRE DE CANNY

Dans ce chapitre, nous allons aborder la mise en application de la solution présentée dans les chapitres précédents au travers d'un exemple d'application logicielle : un filtre permettant la détection de contours dans une image, appelé filtre de Canny. Grâce à cet exemple, nous allons traverser les différentes étapes du flot présenté dans les chapitres 2 et 3 afin d'analyser l'efficacité de la solution développée.

4.1 Spécifications et modélisation de l'algorithme

4.1.1 Explication de l'algorithme

Pour la présentation de l'algorithme, nous utilisons une image (figure 4.1-1) pour expliquer les différentes étapes du processus.

Figure 4.1-1 : image originale $\mathcal{I}$

L'algorithme du filtre de Canny dans sa forme la plus simple repose sur 4 étapes. La première étape consiste à effectuer un flou gaussien sur l'image afin de supprimer les composantes hautes fréquences indésirables, comme les défauts de capteurs ou les textures fines comme les poils du tigre dans l'image exemple (les composantes hautes fréquences désirables étant les contours eux-mêmes). Nous profitons de cette étape pour passer l'image en niveaux de gris, afin de réduire la quantité de calculs et de transferts de données. Afin de créer le flou gaussien, nous procédons à une convolution entre l'image originale et un filtre gaussien (figure 4.1-2).

$Gauss$

16	26	16
26	41	26
16	26	16

$\mathcal{g}$

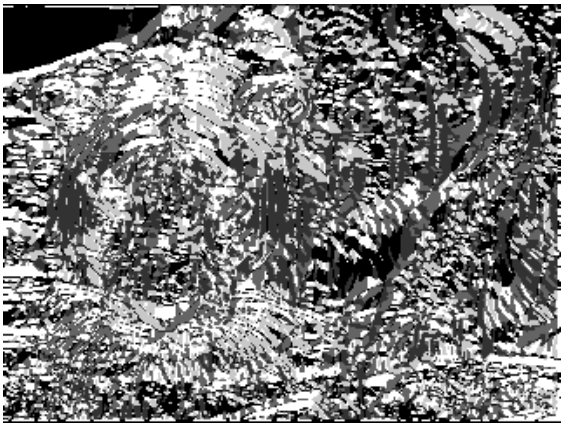
(1) $GreyScale = \mathcal{I}_{red} \times 0.21 + \mathcal{I}_{green} \times 0.72 + \mathcal{I}_{blue} \times 0.07$

(2) $Gauss = GreyScale * \mathcal{g}$

Figure 4.1-2 : Étape 1 : Application du flou gaussien

Une fois cette première étape de flou gaussien effectuée, la deuxième étape de l'algorithme est de créer une carte des gradients de l'image ainsi qu'une carte de l'orientation de ces gradients. Cela consiste à repérer les changements rapides d'intensité dans l'image. Pour ce faire, nous réalisons deux nouvelles convolutions sur l'image : l'une pour repérer les changements horizontaux et l'autre pour repérer les changements verticaux. À partir de ces deux convolutions, nous générons la carte des gradients en calculant la racine carrée de la somme des carrés des gradients verticaux et horizontaux. L'orientation du gradient pour un pixel spécifique est donnée par l'arc-tangente du rapport d'intensité du gradient horizontal sur le gradient vertical pour ce pixel. Une fois l'orientation du gradient trouvée, celle-ci est classée dans une des quatre catégories d'orientation que nous avons définies : nord/sud, est/ouest, nord-ouest/sud-est et nord-est/sud-ouest. Cette étape est résumée dans la figure 4.1-3.

$Grad$



$Theta$

(1) $Vgrad = Gauss * \mathcal{v}$

(2) $Hgrad = Gauss * \mathcal{h}$

(3) $Grad = \sqrt{Hgrad^2 + Vgrad^2}$

(4) $Theta[i,j] = \arctan\left(\frac{Hgrad[i,j]}{Vgrad[i,j]}\right)$

-1	0	1
-1	0	1
-1	0	1

$\mathcal{h}$

-1	-1	-1
0	0	0
1	1	1

$\mathcal{v}$

Figure 4.1-3 : Étape 2 : Génération de la carte des gradients et de l'orientation des gradients

La troisième étape du détecteur de contours utilise les tables que nous avons créées dans l'étape précédente afin de générer une bordure d'épaisseur d'un pixel par un processus appelé suppression des non-maximums. Cela consiste à traverser chaque pixel de l'image et à le comparer à ses voisins dans la direction du gradient. Si ce pixel a la plus grande valeur parmi ses voisins, il est conservé en tant que bordure, sinon il est considéré comme faisant partie de l'arrière-plan. On récupère alors l'image $Grad$, composée de bordures d'un pixel de large comme illustré dans la figure 4.1-4.

Figure 4.1-4 : Étape 3 : Suppression des non-maximums

Une fois cette étape terminée, un dernier filtrage de l'image est effectué pour supprimer les contours faibles. Ce filtre effectue un seuillage double, avec une valeur basse et une valeur haute. Si le pixel analysé est inférieur à la limite basse, il est considéré comme arrière-plan. S'il est supérieur à la valeur haute, alors il est définitivement considéré comme un pixel d'arête. Enfin, si le pixel est entre les deux valeurs de seuil, il n'est conservé que si au moins un de ses huit voisins directs est un pixel fort. Le résultat final du filtre est présenté ci-dessous.

Figure 4.1-5 : Étape finale : filtre à hystérésis

4.1.2 Implémentation C

Nous pouvons dès à présent déterminer plusieurs informations d'analyse en observant le déroulement de l'algorithme. Premièrement, le flot du détecteur est pratiquement entièrement séquentiel. En effet, l'exécution de l'étape deux ne peut se faire en parallèle de celle de l'étape une. On en déduit le flot de données suivant :

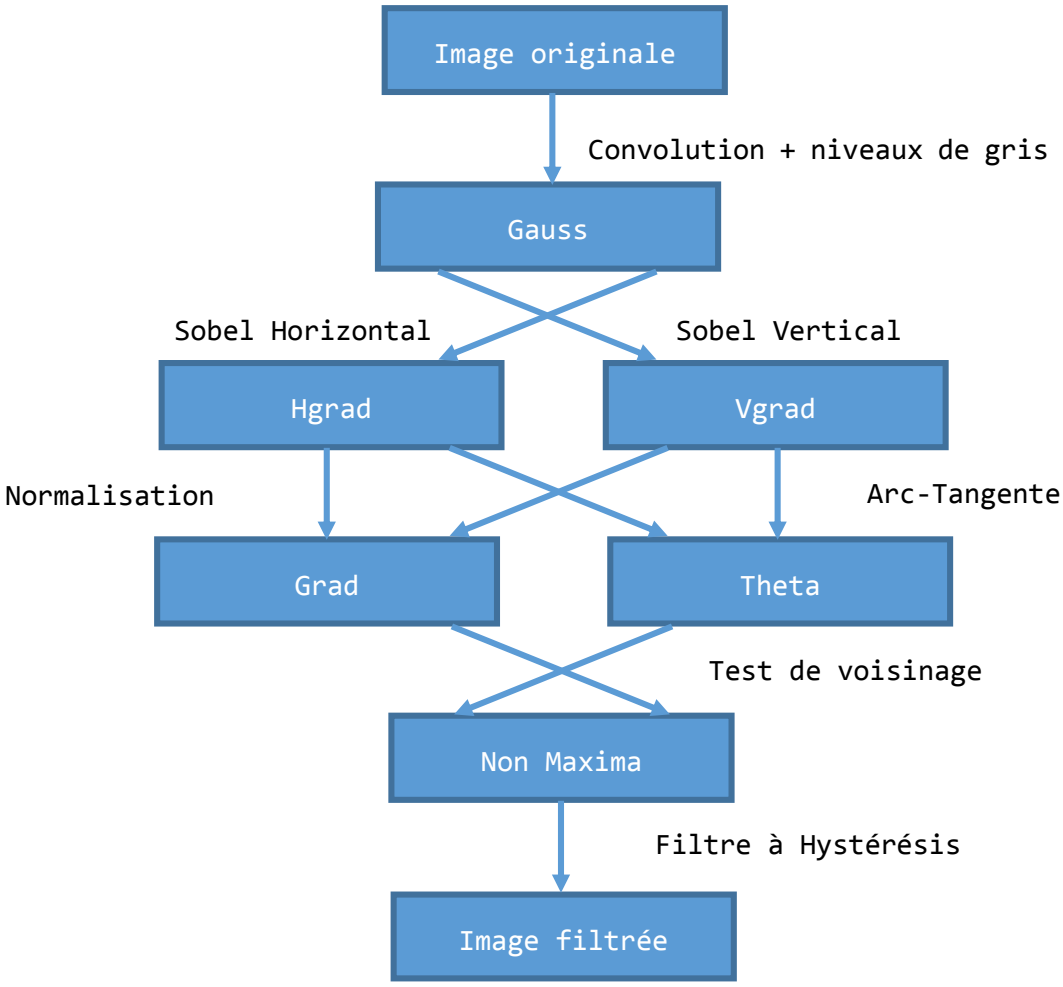


Figure 4.1-6 : Premier flot de données

En revanche, tous les traitements présentés ci-dessus sont entièrement parallélisables à condition d'avoir accès aux images complètes.

Suite à ce constat, nous avons créé une première implémentation C de l'algorithme. Notre algorithme génère 4 tables destinées à accueillir les données nécessaires au bon déroulement du filtrage. La fonction principale de l'implémentation appelle tour à tour chacun des filtres présentés

précédemment : passage en niveaux de gris, filtre gaussien, calcul des gradients verticaux et horizontaux, calcul des gradients normalisés, calcul de l'orientation des gradients, suppression des non-maximums et filtre à hystérésis. En plus de ces fonctions principales, nous avons défini une fonction séparée permettant la convolution de deux tables de 9 pixels. Enfin nous avons défini une série de fonctions utilitaires, permettant notamment l'ouverture d'un fichier BMP (à l'aide d'une librairie à code ouvert, LibBmp [37]), la récupération d'une fenêtre 9 pixels au sein d'une image pour la convolution et la sauvegarde d'une table en tant qu'image BMP.

L'intérêt de cette implémentation pour notre logiciel de traduction de code est de facilement permettre la segmentation du code en module (car notre point d'entrée sur la modularisation dans C2Space est la fonction C).

Un dernier point important de notre implémentation logicielle est que nous passons de fonction en fonction des pointeurs de données pointant sur des tables représentant des images entières (maximum 512*512 pixels).

Les images présentées précédemment proviennent directement de cette implémentation. Nous avons aussi créé une fonction de vérification de l'image filtrée générée afin de s'assurer que le résultat est similaire à une image de référence (cela nous permettra par la suite de vérifier que le filtre a bien fonctionné sans avoir à afficher l'image).

Cette implémentation conclut la première partie du flot ESL proposé, dans laquelle nous avons modélisé une première fois le système étudié.

4.2 Analyse prépartitionnement avec Pareon

La deuxième partie du flot consiste en une première analyse de notre modélisation avant même de procéder au partitionnement matériel/logiciel du code. Cette première analyse nous permet de procéder au raffinement du modèle logiciel présenté dans la partie 4.1. L'analyse nous permet aussi d'orienter le partitionnement lors de l'étape suivante en fournissant des informations sur le potentiel de parallélisme de l'implémentation ainsi que les parties les plus intensives en temps de calcul.

Pour procéder à l'analyse, nous utilisons le logiciel de profilage de code Pareon Profile présenté dans le chapitre 2. Pour ce faire, nous compilons notre code avec le compilateur modifié de Pareon

afin d'instrumenter le code. Une fois le code exécuté, nous pouvons alors accéder aux résultats de profilage dans l'interface graphique de Pareon.

4.2.1 Conditions de test

Pour obtenir une analyse réaliste, nous avons retravaillé le code logiciel pour supprimer tout le code non indispensable au calcul de l'algorithme : suppression des impressions dans la console, suppression des sauvegardes d'image, et suppression des appels à la fonction de vérification. Une première exécution avec ces parties de code est effectuée afin de s'assurer du bon fonctionnement du filtre.

De plus, afin d'avoir des temps d'exécution mesurables, nous effectuons 100 itérations du filtre sur une image de 64*64 pixels (figure 4.2-1).

Figure 4.2-1 : Protocole d'expérimentation pour l'analyse prépartitionnement

4.2.2 Analyse du potentiel de parallélisme

Au cours de notre analyse, nous avons récupéré de multiples informations sur l'algorithme. Nous présentons les résultats dans l'ordre de découverte.

En premier lieu, nous observons la répartition du temps de calcul dans les différents appels de notre arbre d'appel. Cette répartition est montrée à la figure 4.2-2. La boucle 158 représente les 100 itérations discutées plus haut. Le premier constat est que sur les 99.9% du temps d'exécution passé dans cette boucle, deux fonctions occupent plus de 80% du temps processeur. Ces deux fonctions sont `calculateGrad`, qui génère la carte des gradients de l'image et `calculateTheta`, qui génère la carte des orientations de ces mêmes gradients avec respectivement 50.75% et 31.25% du temps d'exécution totale. Ces deux fonctions sont donc des candidates idéales à un partitionnement sur coprocesseur. Toutefois, il est nécessaire de s'assurer que ces boucles présentent un bon potentiel de parallélisme. Pareon nous permet d'analyser plus en avant l'arborescence d'appels afin

d'analyser le corps de chacune de ces fonctions. La figure 4.2-3 nous montre cette analyse pour la fonction `calculatetheta`.

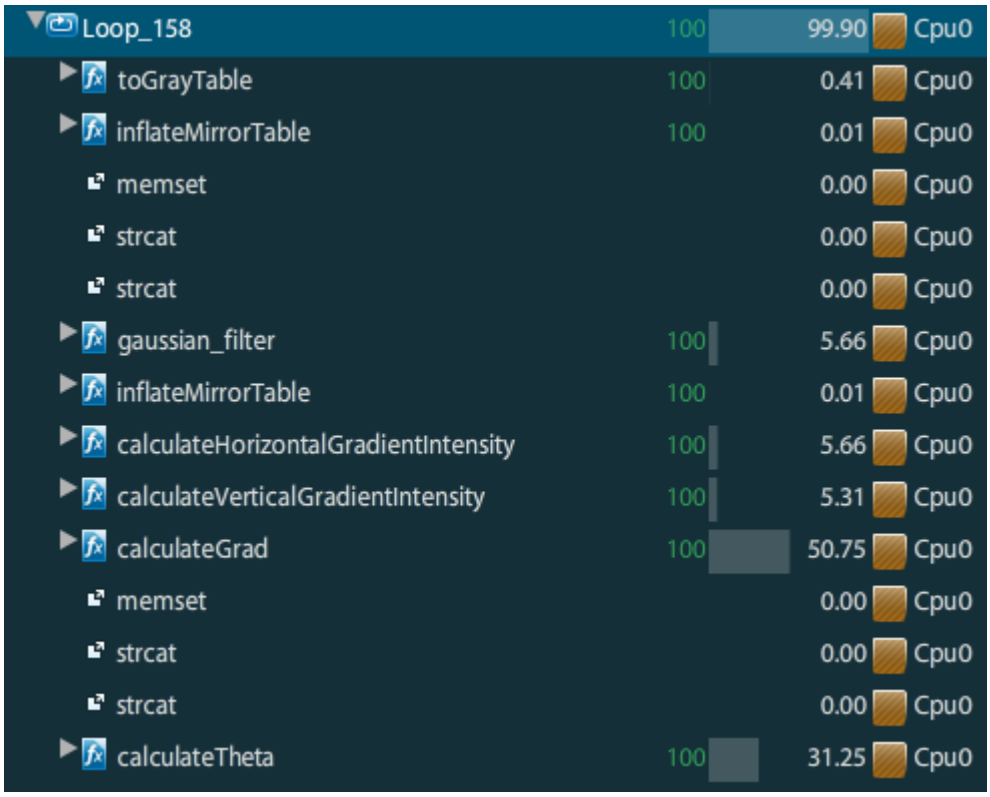


Figure 4.2-2 : Liste des portions de temps d'exécution principales de l'implémentation

La figure 4.2-3 montre une partie de l'arbre d'appel du programme, présentée de manière graphique. La partie inférieure de la figure affiche les dépendances que nous pouvons trouver dans la boucle 271 du code (cette boucle représente la recherche d'orientation du gradient pour une colonne de pixels de l'image). La première dépendance correspond à la variable d'induction de la boucle, utilisée pour identifier l'indice de la ligne. Cette dépendance peut être levée de manière triviale dans le cadre d'un déroulement de boucle. Deux autres dépendances ont été repérées par Pareon et correspondent à la recherche des valeurs minimales et maximales des orientations des pixels. Ces calculs comparent le résultat trouvé à l'itération précédente avec celui de l'itération actuelle et bloquent donc la parallélisation du code. Toutefois, ces deux valeurs étaient calculées à des fins de tests lors du débogage de l'implémentation et n'ont en réalité pas d'incidence sur le déroulement de l'algorithme. Nous pouvons donc signaler à Pareon que ces deux dépendances peuvent être ignorées et rédiger une note expliquant la raison.

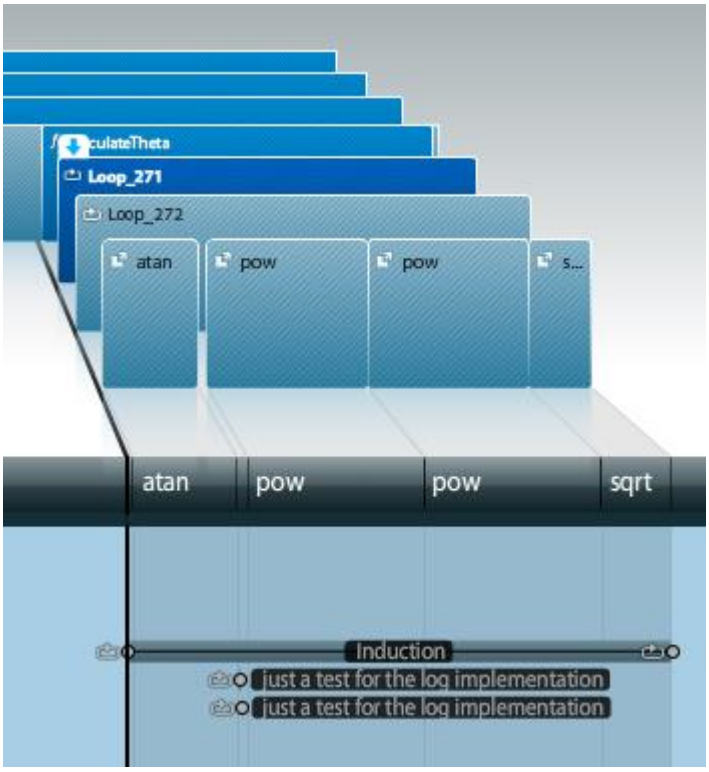


Figure 4.2-3 : Dépendances de données dans la boucle 271

Puisqu'aucune dépendance ne gêne la parallélisation de cette boucle, nous pouvons demander à Pareon de paralléliser cette section du code et d'estimer les gains de vitesse qu'engendre cette modification. La boucle principale de la fonction `calculateGrad` ne présentant aucune dépendance, nous pouvons la paralléliser elle aussi, pour obtenir les résultats présentés sur la figure 4.2-4 et la table 4.2-1.

Nous pouvons observer sur la figure 4.2-4 que les principaux postes de dépense de l'algorithme ont changé. Après parallélisation des segments `calculateGrad` et `calculateTheta`, les segments `gaussian_filter`, `calculateVerticalGradientIntesity` et `calculateHorizontalGradientIntensity` sont devenus bien plus représentatifs dans le temps d'exécution total.

La table 4.2-1 quant à elle, résume les gains apportés par la parallélisation des deux segments `calculateGrad` et `calculateTheta`. Ces deux segments ont subi une accélération de x4, mais étant donné que le reste du code n'a pas été parallélisé, la loi d'Amdahl limite l'accélération globale à x2.6 pour une réduction du temps d'exécution de 3.3 secondes à 1.3 seconde.

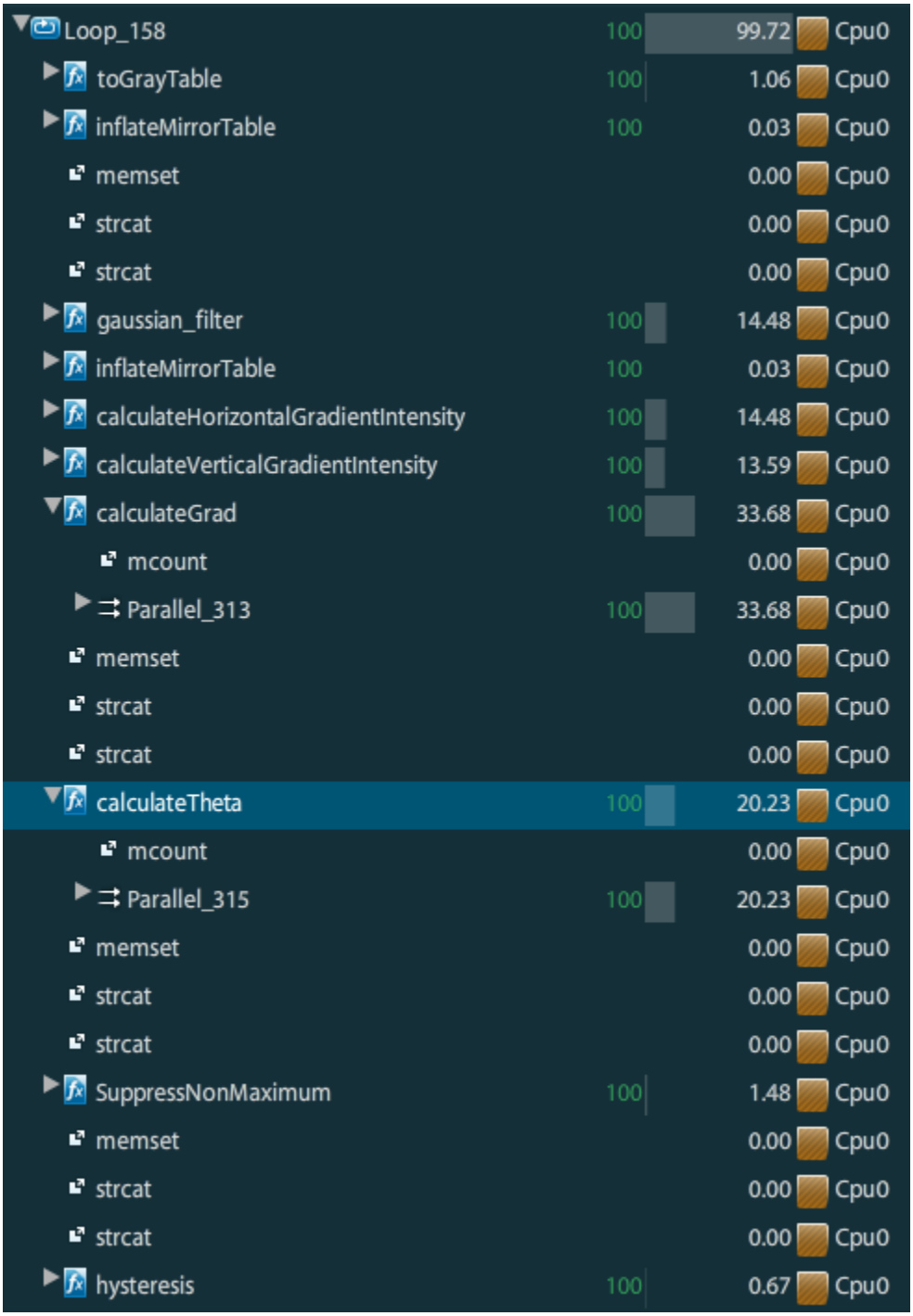


Figure 4.2-4 : Répartition du temps d'exécution après parallélisation (x4) des segments calculateTheta et calculateGrad

Tableau 4.2-1 : Résultats de la première analyse de potentiel de parallélisme

Temps avant parallélisation	Temps après parallélisation	Facteur d'accélération
3.3s	1.3	x2.6

4.2.3 Analyse du potentiel d'optimisation des parties intensives en calcul

Pour la deuxième partie de l'analyse, nous allons identifier quelles sont les parties du code qui présentent un bon potentiel d'optimisation logicielle afin d'améliorer les performances séquentielles de notre algorithme.

Premièrement, nous analysons à nouveau les parties les plus intensives en temps de calcul pour nous apercevoir que le temps passé dans les fonctions de la bibliothèque standard `math`, `pow()`, `atan()` et `sqrt()` (nécessaires à la fois pour le calcul de la carte des gradients et de l'orientation de ceux-ci), représente une partie très importante du temps de calcul total (figure 4.2-5). La fonction `pow()` est utilisée 6 fois avec un temps d'exécution de 10% du temps total à chaque fois. Les fonctions `sqrt()` et `atan()` représentent quant à elles 18% du temps d'exécution total. Bien que les fonctions `sqrt()` et `atan()` ne puissent pas être remplacées facilement, la fonction puissance peut être remplacée par une multiplication, car nous calculons seulement le carré de nos valeurs.

Deuxièmement, une fois la parallélisation effectuée, nous avons pu nous apercevoir qu'une partie non négligeable du temps d'exécution (21 %) est utilisée par la fonction responsable d'aller chercher une fenêtre de 9 pixels dans l'image (cette fonction est utilisée dans 3 fonctions différentes, avec un temps d'exécution de 7 % du temps d'exécution total pour chacune de ces occurrences). Ce temps de calcul peut être évité en abandonnant la recherche de fenêtre pour travailler directement sur l'image lors des différentes convolutions. Cela revient concrètement à fusionner les fonctions `convolutionAbs()` et `getGrayWindow()` à l'intérieur des fonctions qui requièrent ces deux appels. Ces constats sont illustrés à la figure 4.2-6.

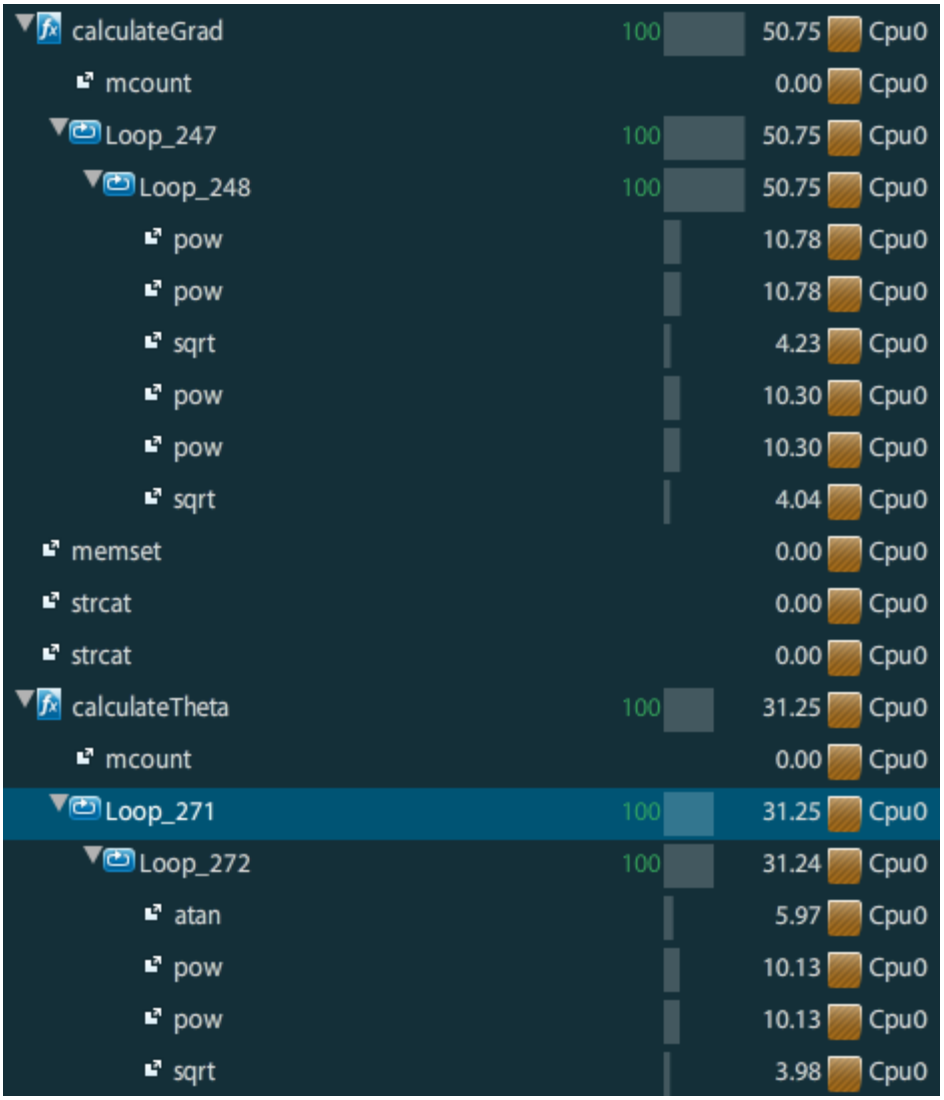


Figure 4.2-5 : Temps d'exécution des fonctions de la bibliothèque standard

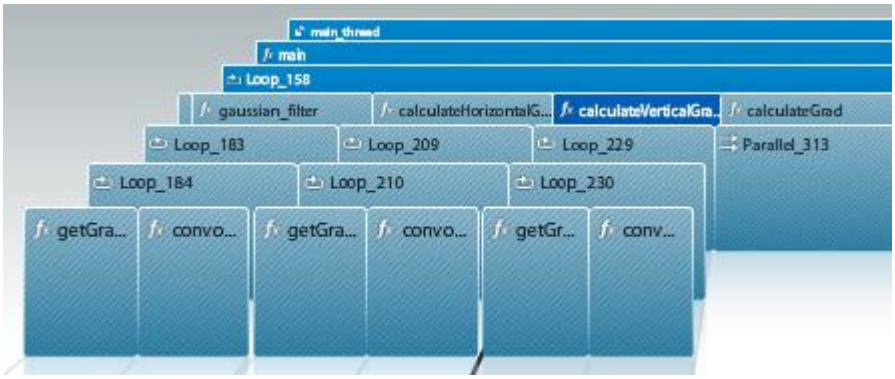


Figure 4.2-6 : Dépenses engendrées par les fonctions getGrayWindow() et convolutionAbs()

Enfin, nous nous sommes rendu compte que nous parcourions l'image complète de manière non nécessaire à cause de la segmentation de l'algorithme en plusieurs fonctions spécifiques. En effet, les étapes de calcul du gradient vertical, horizontal, de la normalisation des gradients et de calcul d'orientation peuvent toutes s'effectuer en une seule traversée de l'image. Une des solutions à ce problème est de construire une fonction regroupant ces 4 fonctionnalités.

Toutes ces considérations ont conduit à une nouvelle implémentation du code. Le flot de données dans cette nouvelle implémentation est comme suit :

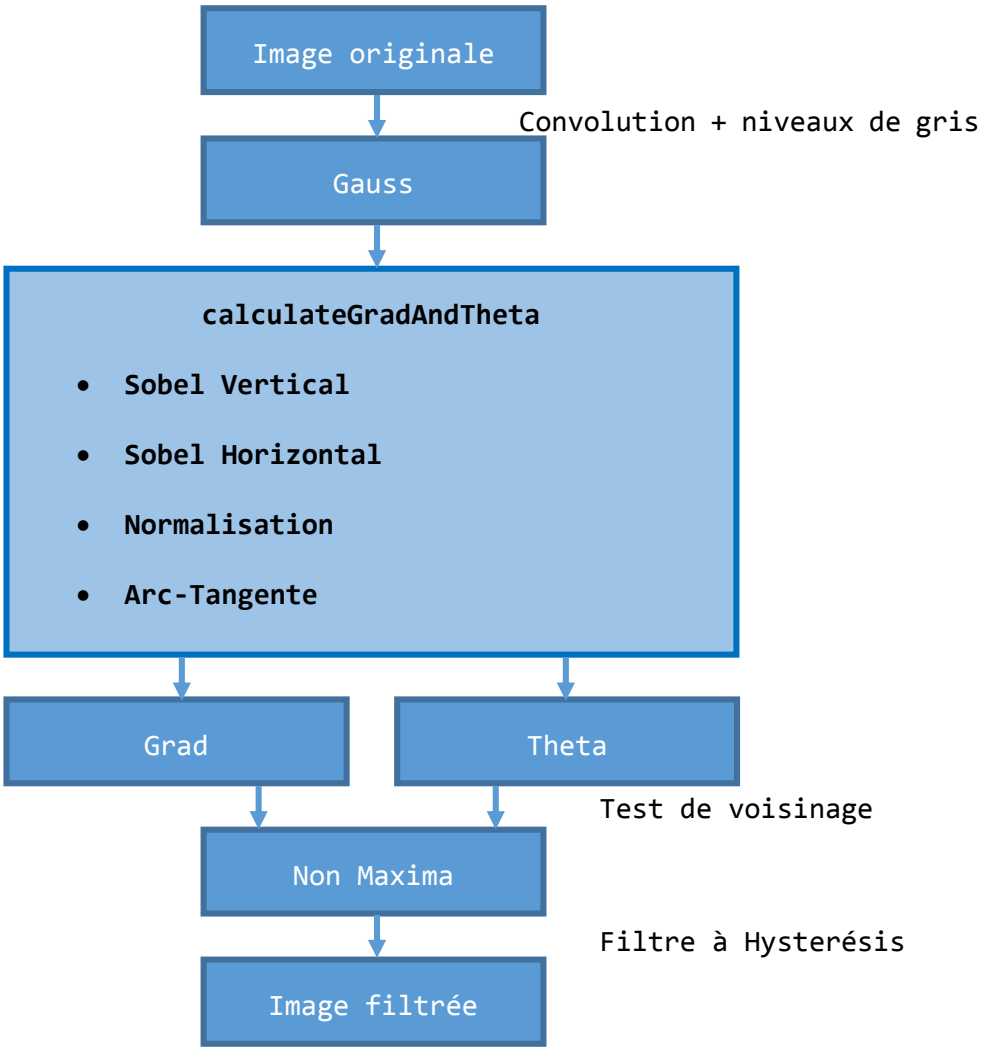


Figure 4.2-7 : Nouvelle implémentation de l'algorithme

Les différents traitements jusqu'ici segmentés ont tous été regroupés dans une seule fonction `calculateGradAndTheta()`. Cela permet de supprimer deux parcours de l'image complète ainsi qu'un parcours de convolution. Nous avons aussi procédé aux changements énoncés

précédemment : suppression des fonctions `pow()` et fusion du code de la fonction `convolutionAbs()` dans les fonctions concernées (`gaussian_filter()` et `calculateGradAndTheta()`). Les résultats de l'analyse de prépartitionnement sont expliqués ci-dessous :

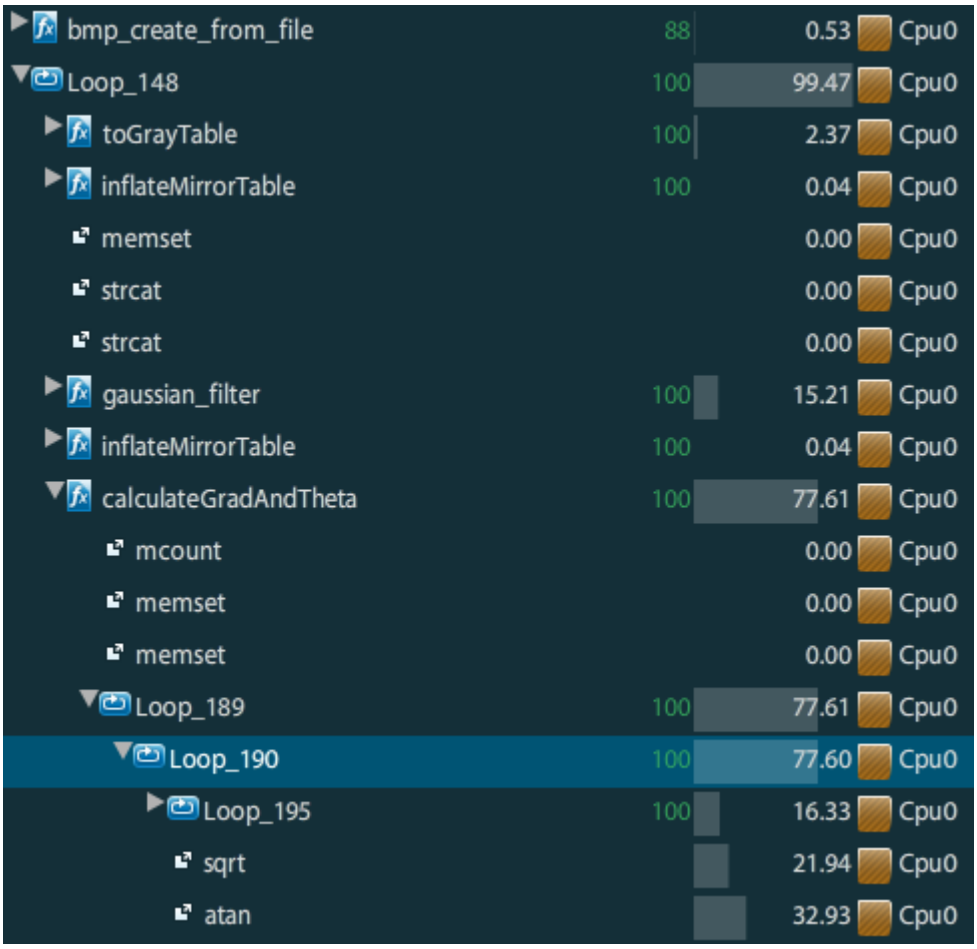


Figure 4.2-8 : Postes de dépenses de la nouvelle implémentation (temps d'éxécution : 0.601s)

Comme on peut le voir sur la figure 4.2-8, la nouvelle fonction `calculateGradAndTheta()` occupe 77.6% du temps total d'exécution. Ce temps est réparti dans trois fonctionnalités : le nid de boucle 195 procédant aux convolutions verticales et horizontales (16.3 %), la racine carrée `sqrt()` (21.9 %) et l'arc tangente `atan()` (32.9 %). Les 25 % restants du temps sont répartis dans les différents autres traitements de l'image, avec notamment 15.2 % du temps passé dans la convolution du filtre gaussien.

Comme pour l'implémentation précédente, nous avons tenté de procéder à la parallélisation de la fonction la plus intensive, `calculateGradAndTheta()`. Une fois cette transformation effectuée, la

portion de temps utilisée par la fonction `gaussian_filter()` est devenue assez imposante pour justifier sa parallélisation. Les accélérations de ces différentes transformations sont exposées dans la table suivante et contrastées avec celle de la première implémentation.

Tableau 4.2-2 : Résultats finaux de l'analyse prépartitionnement

Configuration	**Temps d'exécution**	**Temps après parallélisation**	**Facteur d'accélération**	**Accélération par rapport à (1)**
(1)	3.3 s	1.3 s	2.6	2.6
(2)	2.1 s	0.559 s	3.7	5.9
(3)	0.601 s	0.250 s	2.4	13.2
(4)	0.601 s	0.189 s	3.2	17.5

Légende des configurations

(1) :Ancienne Implémentation

(2) :Fusion des convolutions + suppression des itérations non-nécessaires = création de la fontion calculateGradAndTheta()

(3) : (2) + suppression des opérateurs puissance : parallélisation de la fonction calculateGradAndTheta()

(4) : (3) + parallélisation de la fonction gaussian_filter()

L'analyse effectuée sur le code avant de commencer le partitionnement nous a permis d'accomplir trois objectifs :

1. Nous avons pu améliorer la structure même du code avant de passer à l'environnement de co-simulation afin de tirer une accélération de 5.5 par rapport à l'implémentation originale.
2. Nous avons identifié les sections du code recélant le plus grand potentiel de parallélisme grâce aux estimations de Pareon.

3. Nous avons aussi identifié les fonctions les plus intensives en calcul : la fonction `atan()` dans la configuration (4) représente 49ms du temps d'exécution total, soit plus d'un quart du temps d'exécution et représente donc une excellente candidate d'étude pour aller chercher encore plus d'accélération en passant par une approximation.

Suite à cette analyse, nous allons passer aux parties 3 et 4 de notre flot ESL : le partitionnement matériel logiciel et l'analyse post-partitionnement. Ces étapes sont effectuées dans SpaceStudio. Toutefois, pour pouvoir utiliser notre code sur SpaceStudio, celui-ci doit subir une série de transformations. Ces transformations sont gérées par l'outil que nous avons développé dans ce mémoire : C2Space.

4.3 Traduction assistée du code avec C2Space

L'outil présenté au Chapitre 3 est ici mis à contribution afin de traduire le code logiciel séquentiel en modules SpaceStudio, qui représentent un ensemble de tâches séparées, pouvant être allouées à des ressources matérielles ou logicielles.

Dans le cadre d'une exploration réelle, nous n'aurions traduit que les deux cas les plus intéressants : la segmentation `main/gaussian_filter/calculateGradAndTheta` et la segmentation `main/CalculateGradAndTheta`.

Toutefois, dans le cadre de notre expérimentation, nous avons testé plus de configurations afin de pouvoir comparer et contraster nos résultats. Les différentes configurations sont listées dans le Tableau 4.3-1.

Toutes ces configurations sont générées automatiquement par C2Space avec un minimum d'entrées utilisateur (fonctions à modulariser, taille des variables pointeurs, méthode de communication). Une fois le code généré, le script Python permet d'importer directement les modules et sources dans la suite SpaceStudio. Des modifications mineures doivent être effectuées dans le code généré : envoi manuel des paramètres normalement envoyés au `main` par ligne de commande, utilisation d'un script utilisateur pour charger les données traitées (ici les images) dans la machine virtuelle pour la cosimulation et changement des tailles de pile des modules matériels (dans le cas où les tailles par défaut ne sont pas suffisantes).

Tableau 4.3-1 : Liste des configurations générées par C2Space

Configuration	Code d'origine	Modules
1	Première Implémentation	`main`
2	Première Implémentation	`main, gaussian_filter`
3	Première Implémentation	`main, calculateTheta, calculateGrad`
4	Fusion	`main`
5	Fusion	`main, calculateGradAndTheta`

Certaines des capacités du logiciel C2Space n'ont pas été utiles lors de cette exploration architecturale, mais ont été testées et sont fonctionnelles. Celles-ci peuvent avoir des utilités lors de l'exploration d'algorithmes différents. Les fonctionnalités non utilisées sont : la fission d'une boucle hors de son module original, le passage de variables volumineuses en mémoire externe et les modules appelants multiples.

4.4 Partitionnement, analyse et débogage avec SpaceStudio et HLS Vivado

4.4.1 Flot de travail sous SpaceStudio

Maintenant que nous avons un code portable sur SpaceStudio, nous allons pouvoir commencer l'exploration de nos différentes configurations. La première étape est de modifier légèrement le code traduit afin de le rendre exécutable. Comme dit plus haut, cela consiste à fournir les variables que nous aurions données en ligne de commande directement dans le module main. De plus, nous devons préparer l'environnement de test en mettant à disposition les images dans le répertoire de l'exécutable (ou bien via un script défini par l'utilisateur pour la co-simulation).

Il est aussi important de s'assurer que l'intégralité des variables allouées de manière statique dans chaque module ne dépasse pas la taille de pile par défaut des modules SpaceStudio (`0x16000`). Si cela est le cas, il faut rajouter une ligne dans le constructeur du module pour augmenter la taille de pile.

Une fois ces modifications effectuées, nous pouvons passer à la vérification fonctionnelle de notre système de modules. Ceci est pris en charge par la partie Elix de SpaceStudio. Pour cette vérification, la notion matérielle/logicielle n'est pas prise en compte. Le code est compilé et exécuté sans minutage. Cet outil permet de nous assurer que la transformation de l'algorithme séquentiel en réseau de modules est fonctionnelle. Dans notre cas, une fois les modifications ci-dessus effectuées, le code s'exécute sans problème.

Une fois que notre code s'exécute en fonctionnel, nous pouvons passer à la phase d'exploration architecturale à proprement parler. À l'aide de la partie Simtek de SpaceStudio, un environnement de co-simulation, nous pouvons définir une architecture matérielle, pouvant contenir des mémoires externes (BRAM), des processeurs matériels ou présynthétisés, des minuteries, des bus, etc. Le logiciel nous permet ensuite de choisir comment allouer nos modules à nos ressources. Ainsi, nous pouvons choisir d'allouer un module à un processeur, auquel cas ce module sera un fil d'exécution au sein de ce processeur, ou bien de l'allouer au bus, auquel ce module sera considéré comme un coprocesseur matériel. De plus, nous pouvons choisir le mode de communication intermodules pour chaque lien entre modules. Ces modes de communications peuvent permettre l'utilisation d'un DMA (accès direct à la mémoire) si le processeur choisi en est équipé ou bien l'utilisation par défaut de files de communications matérielles ou logicielles.

La co-simulation de SpaceStudio donne accès à un panel d'informations via son interface de scrutation SpaceMonitor. Nous obtenons alors des informations sur le temps d'exécution de chaque module sur le processeur, les taux d'utilisation des cœurs de celui-ci. Ainsi qu'un chronogramme des communications intermodules au cours de l'exécution. La simulation calcule un temps d'exécution simulé sur la plateforme cible. Ces temps d'exécution seront notre métrique de comparaison.

Les temps d'exécution avant prise en compte des délais d'exécution matériels sont regroupés dans la table 4.4.1.

En effet, bien que le simulateur de SpaceStudio donne une approximation des temps d'exécution des modules logiciels et des communications intermodules, celui-ci ne prend pas en charge les temps de simulation des modules matériels. Par défaut, ceux-ci sont considérés comme nuls. Nous devons donc utiliser la suite de synthèse haut-niveau HLS Vivado afin de synthétiser nos modules matériels et ainsi obtenir une approximation du temps d'exécution de ceux-ci, que nous pourrons ajouter (*back-annotation* en anglais) dans notre simulation à l'aide de la fonction `computeFor()`. Ce temps d'exécution sera fonction des directives d'optimisation que nous fournirons à Vivado lors de la synthèse (pipelinage, déroulement de boucle, etc.). Ces directives ont aussi une influence directe sur la taille que viendra occuper notre module matériel sur le FPGA. Cette utilisation des ressources matérielles nous servira de seconde métrique afin de comparer et contraster les gains de performance obtenus.

Tableau 4.4-1 : Temps d'exécution de différentes configurations avant ajout des temps d'exécution matériels

Expérience	Configuration utilisée	Partitionnement	Méthode de communication	Temps d'exécution
a	1	N/A : Full SW	N/A	0.57 s
b	2	gaussianFilter HW	Fifo HW	0.58 s
c	3	calculateGrad HW calculateTheta HW	Fifo HW	0.56 s
d	4	N/A : Full SW	N/A	0.44 s
e	5	calculateGradAndTheta HW	Fifo HW	0.19 s

Les résultats ci-dessus, bien qu'incomplets peuvent déjà être analysés sur certains points.

Tout d'abord, ces résultats permettent d'attester de la justesse de l'analyse de prépartitionnement. En effet, l'accélération de la configuration `d` (configuration logicielle pure de la nouvelle implémentation) par rapport à la configuration `a` (configuration logicielle pure de l'ancienne

implémentation) correspond à l'accélération de la solution logicielle (2) par rapport à la solution (1) de la table 4.2-2 (x1.57 vs x1.29). Cependant, l'accélération liée à la suppression des opérateurs puissance ne semble pas se retrouver sur ces résultats de simulation. Cela est peut-être lié au fait que le compilateur de SpaceStudio optimise ces appels de fonction alors que le compilateur de Pareon ne le fait pas. Nous ne prendrons donc plus en compte l'accélération liée à cette modification dans la suite de notre analyse. Nous remarquons de toute manière que nos temps de simulation sont plus proches de ceux des solutions logicielles après suppression des opérateurs puissance (`a` : 0.57 s `d` :0.44 s Vs. 0.60 s) que de ceux où les opérateurs puissance sont présents.

La deuxième constatation que nous pouvons effectuer est que le partitionnement d'éléments matériels dans les configurations utilisant l'implémentation naïve n'a que très peu d'effet sur les temps d'exécution. Dans le cas de la configuration `b`, cela s'explique facilement par le fait que nous avons fait passer en matériel une partie du code qui ne représentait qu'une petite portion du temps d'exécution logiciel total (5.66 %). Le gain de temps est entièrement contrebalancé par les délais de communication introduits par le partitionnement, conduisant même à un léger ralentissement du temps d'exécution, et ce, avant même d'avoir pris en compte le temps d'exécution du module partitionné en matériel. Le cas de la configuration `c` est plus complexe. En effet, l'accélération causée par cette configuration est presque nulle, alors que les modules placés en matériel sont censés représenter une portion importante du temps d'exécution. Toutefois, afin d'amener ces deux modules en matériel, nous créons une quantité non négligeable de communications intermodules (4 envois de 4356 données et 2 réceptions de cette même quantité). Cette quantité de transfert de données finit par contrecarrer l'accélération procurée par les coprocesseurs.

Ces deux premiers partitionnements nous apprennent des leçons importantes sur les enjeux à prendre en compte lors du choix des modules à mettre en matériel. Il est tout d'abord important de prendre en considération la participation au temps d'exécution total du module visé afin d'observer une accélération du temps de calcul logiciel, comme énoncé par la loi d'Amdahl. Toutefois, ce choix doit aussi être contrasté par la quantité de données à transférer pour faire fonctionner ce module afin que les délais de communications ne viennent pas détruire l'accélération créée. C'est avec ces concepts en tête que nous avons créé la configuration `e`, dans laquelle nous fusionnons une partie importante du temps d'exécution dans un seul module afin de maximiser le temps d'exécution passé en matériel tout en réduisant les délais de communications. Ainsi, le même traitement dans la configuration naïve aurait demandé 6 envois et 4 retours de table de 4356

données par itération du filtre tandis que notre seconde implémentation ne demande qu'un envoi et 2 retours. Nous observons alors une réelle accélération du temps d'exécution logiciel (x2.3). Il nous reste à observer le temps d'exécution de notre module matériel avant de conclure quelle accélération nous avons réellement créée dans notre application.

4.4.2 Optimisation du module matériel sous HLS Vivado

Ici, nous allons nous concentrer sur la configuration `e` et plus précisément sur la synthèse haut niveau du module `calculateGradAndTheta`. HLS Vivado nous permet de « marquer » certains éléments du code source pour pouvoir les repérer plus facilement dans notre rapport de synthèse. Nous allons nous intéresser à quatre corps de boucle dans notre analyse que nous avons intitulés `L1`, `L2`, `L3`, `L4` et qui représentent respectivement les itérations sur les axes x et y de l'image complète et les itérations sur les axes x et y de la fenêtre glissante de la convolution.

```
1     .
2     .
3     .
4   L1: for (i = 1; i < width + 1; i++) {
5   L2:   for (j = 1; j < height + 1; j++) {
6           //conv
7           int sum_pixelv, sum_pixelh;
8           sum_pixelv = 0;
9           sum_pixelh = 0;
10  L3:     for (x = 0; x < 3; x++) {
11  L4:       for (y = 0; y < 3; y++) {
12              sum_pixelv += (grayscaleIn[(i + x - 1) * MAX_WIDTH + j + y - 1] *
    v_sobel_filter[x * 3 + y]);
13              sum_pixelh += (grayscaleIn[(i + x - 1) * MAX_WIDTH + j + y - 1] *
    h_sobel_filter[x * 3 + y]);
14            }
15          }
16    .
17    .
18    .
```

Extrait 4.4-1 : Étiquetage des corps de boucle de la fonction d'action du module calculateGradTheta

Nous créons plusieurs scénarios de synthèse à l'aide des directives de déroulement de HLS Vivado qui se présentent sous la forme de pragma (par exemple `HLS unroll`). Ces scénarios visent à dérouler des parties de plus en plus grandes des boucles :

- `No unroll` : Pas de déroulement, on laisse l'outil de synthèse libre de faire les optimisations qui lui conviennent

- `L3L4` : On déroule totalement le noyau de convolution
- `L2x2L3L4` : On déroule totalement le noyau de convolution et on crée deux chemins de données pour la boucle L2 (déroulement facteur 2)
- `L2x4L3L4` : identique au scénario précédent, mais avec un déroulement de facteur 4
- `L2x8L3L4` : déroulement de facteur 8

Afin de pouvoir synthétiser notre module, nous avons dû apporter quelques modifications au code source :

- Les valeurs limites des boucles qui étaient variables ont été rendues fixes (64 au lieu de `width` et `height` dans les boucles `L1` et `L2`)
- Les variables membres qui sont allouées dynamiquement dans le constructeur et qui représentent les tables d'entrée et de sortie doivent être allouées de manière statique afin de permettre à l'outil de synthèse de connaitre la taille de ces variables.
- Enfin, l'opération `atan()` a dû être modifiée, car celle-ci n'était pas synthétisable par HLS Vivado. Nous avons opté pour une approche simple de comparaison du ratio gradient Vertical/gradient Horizontal avec les valeurs limites des quatre orientations possibles :

```
1  temp = (double)sum_pixelv / ((double)sum_pixelh);
2  if (temp < -2.4750)
3    ThTableOut[MAX_WIDTH * i + j] = 255;
4  else if (2.4750 < temp)
5    ThTableOut[MAX_WIDTH * i + j] = 255;
6  else if (-2.4750 <= temp && temp < -0.4040)
7    ThTableOut[MAX_WIDTH * i + j] = 100;
8  else if (-0.4040 <= temp && temp < 0.4040)
9    ThTableOut[MAX_WIDTH * i + j] = 50;
10 else if (0.4040 <= temp && temp < 2.4750)
11   ThTableOut[MAX_WIDTH * i + j] = 200;
```

Extrait 4.4-2 : Modification de la fonction atan()

Les différents résultats de synthèse sont présentés dans les graphiques suivants afin de pouvoir comparer l'accélération et les coûts en termes de ressources du FPGA pour chacun des scénarios d'optimisation.

La première chose que l'on peut observer est que la configuration `L3L4`, pour une augmentation très faible des ressources nécessaires, présente une accélération intéressante de x1.56. Le déroulement de la boucle `L2` quant à elle, ne semble pas influencer la rapidité d'exécution de notre algorithme (ou très peu), et ce, à un coût important en ressources notamment en unités de calcul de signal numérique (chaque nouveau déroulement voit le nombre d'unités doubler). Le déroulement de la boucle `L2` semble donc être un gaspillage de ressources dispensable et la configuration `L3L4` semble être le meilleur compromis accessible lors de cette analyse de synthèse.

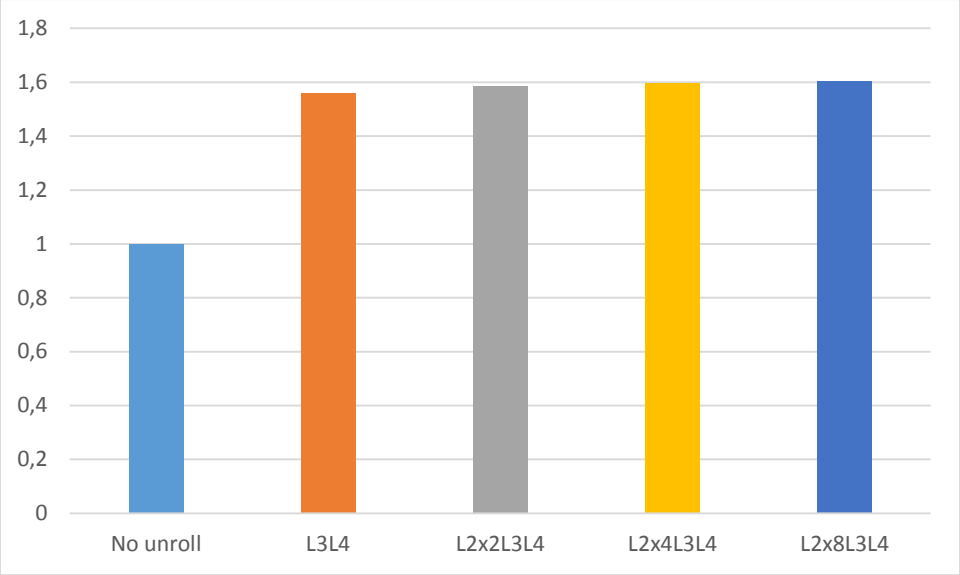


Figure 4.4-1 : Accélération du module matériel en fonction des directives d'optimisation (temps original : 3.99ms)

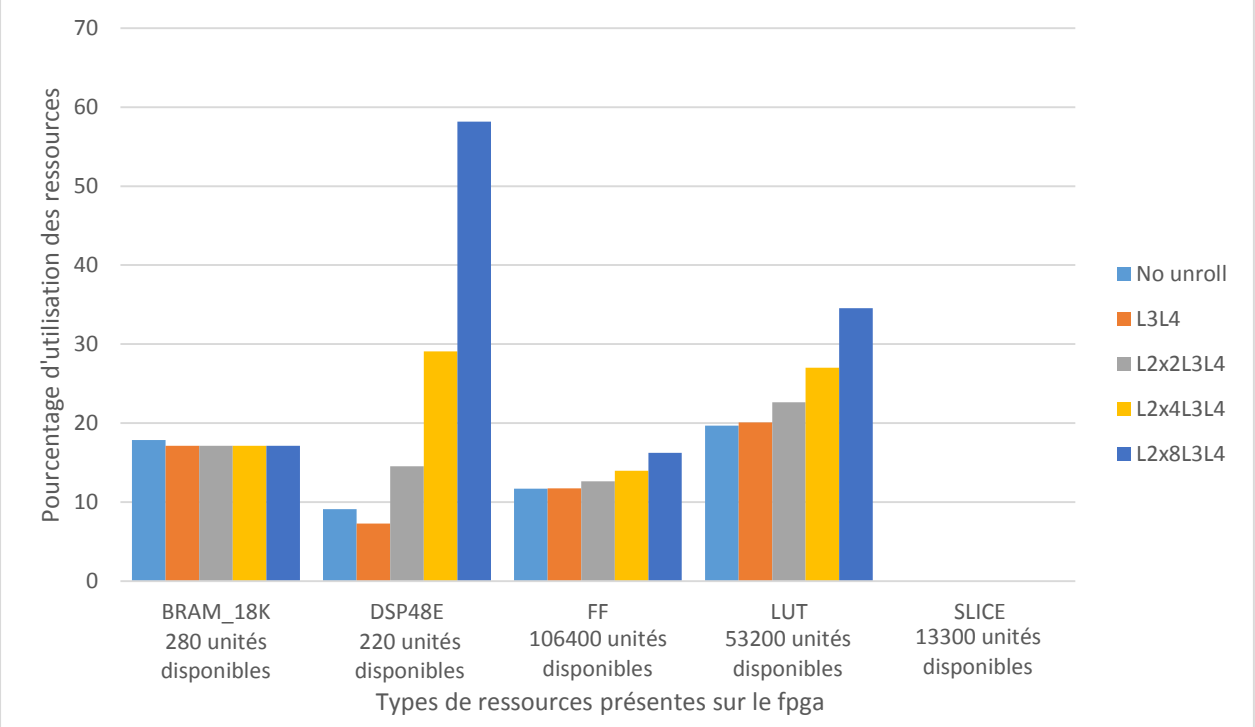


Figure 4.4-2 : Comparaison de la consommation des ressources du FPGA zinq-7000 pour les différents scénarios d'optimisation

4.5 Discussion des résultats

4.5.1 Productivité

Le flot que nous avons présenté permet la création rapide d'un système électronique complet. Les solutions proposées par Vivado (XPS, EDK et HLS) promettent déjà une accélération du cycle de développement comprise entre 4x et 10x par rapport à un cycle de développement RTL traditionnel (d'après la base utilisateur de Vivado). Cela est dû à l'utilisation de techniques haut niveau comme la synthèse automatisée ainsi qu'à l'environnement dédié.

SpaceStudio, qui vient se greffer au-dessus de ces outils, promet une accélération supplémentaire de la productivité, pour aller chercher la limite basse de 6x par rapport aux techniques traditionnelles. Cela est explicable de par l'ajout d'un environnement de cosimulation à haut niveau d'abstraction, ainsi que grâce à la création automatique de toute la logique « colle » nécessaire à l'envoi du projet vers les outils de Vivado.

Notre solution repousse encore ces limites grâce à la génération assistée de projet SpaceStudio (qui était faite à la main auparavant). De plus, l'ajout au flot d'une solution d'analyse prépartitionnement permet de réduire la durée d'exploration architecturale sous SpaceStudio en réduisant l'espace des solutions, et ce lorsque que l'exécution du modèle testé peut s'effectuer en une fraction du temps qui aurait été nécessaire dans les étapes en aval. Il est difficile de donner une appréciation de l'accélération supplémentaire donnée par le flot présenté, par manque d'expérience dans le milieu du développement de systèmes électroniques. On peut toutefois supposer que celui-ci est supérieur à l'accélération promise par SpaceStudio.

4.5.2 Performances

Nous pouvons maintenant agglomérer nos résultats de simulation de la partie logicielle et de la partie matérielle en multipliant par 100 les résultats récupérés lors de l'analyse de synthèse et en les injectant dans les résultats de co-simulation.

Tableau 4.5-1 : Résultats finaux de temps d'exécution en fonction des différentes directives d'optimisation du module matériel

Expérience	Scénario d'optimisation	Partitionnement	Méthode de communication	Temps d'exécution
d		N/A : Full SW	N/A	0.44 s
e	No Unroll	calculateGradAndTheta HW	FIFO HW	0.19 s+0.399 s = 0589 s
e	L3L4	calculateGradAndTheta HW	FIFO HW	0.19 s + 0.255 s = 0.445 s
e	L2x2L3L4	calculateGradAndTheta HW	FIFO HW	0.19 s + 0.251 s = 0.441
e	L2x4L3L4	calculateGradAndTheta HW	FIFO HW	0.19 s + 0.249 s =0.439
e	L2x8L3L4	calculateGradAndTheta HW	FIFO HW	0.19 s + 0.248 s = 0.438

Après avoir rajouté les temps d'exécution matérielle, nous nous apercevons que les temps totaux après partitionnement matériel sont à peine égaux aux temps entièrement logiciels. Cela veut dire que nous n'avons pas réussi à aller chercher d'accélération sur le filtre de Canny. Toutefois, nous avons libéré le processeur d'une charge de calcul de ~ 0.25 ms toutes les 0.44 ms. Ce qui veut dire que dans le cadre d'un système complet utilisant ce filtre de Canny (par exemple, un système de guidage de robot), nous pourrions exécuter d'autres fils d'exécution logiciels pendant que le coprocesseur travaille, et ainsi aller chercher une accélération sur l'application complète.

Il reste cependant un dernier paramètre à prendre en compte, qui remet en cause les constatations effectuées dans le paragraphe précédent. La modification que nous avons effectuée sur le calcul de l'arc-tangente dans le module `calculateGradTheta` n'a pas été répercutée sur notre temps

d'exécution entièrement logiciel. Après avoir effectué à nouveau cette simulation, nous obtenons un temps total de simulation pour notre filtre de 0.16 s, soit moins que le temps d'exécution de la solution partitionnée, avant même d'ajouter le temps d'exécution matériel. Notre solution partitionnée est donc finalement moins rapide que la solution logicielle pure.

Tableau 4.5-2 : Valeurs de temps d'exécution, réajustées pour prendre en compte la modification du code source lors de la synthèse haut niveau

Expérience	Scénario d'optimisation	Partitionnement	Méthode de communication	Temps d'exécution
d		N/A : Full SW	N/A	0.16s
e	No Unroll	calculateGradAndTheta HW	Fifo HW	0.19s+0.399s = 0.589s
e	L3L4	calculateGradAndTheta HW	Fifo HW	0.19s + 0.255s = 0.445s

Ces résultats sont décevants au vu des valeurs attendues lors de l'analyse prépartitionnement. Toutefois, ceux-ci peuvent s'expliquer par différentes raisons.

Tout d'abord, l'analyse prépartitionnement s'est montrée imprécise pour certaines fonctions (comme l'opérateur puissance), ce qui a eu pour effet d'augmenter artificiellement la part de temps de calcul du module `calculateGradTheta`. La répartition des postes de dépenses de la solution logicielle pure dans l'environnement de cosimulation est présentée dans la Figure 4.5-1

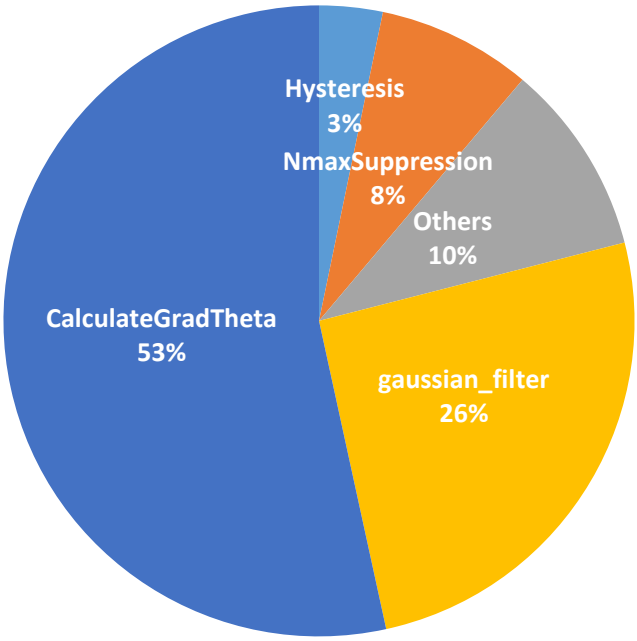


Figure 4.5-1 : Répartition du temps d'exécution de la cosimulation dans les fonctions principales de l'implémentation entièrement logicielle sous SpaceStudio

La fonction `calculateGradTheta()` occupe moins de temps que prévu (53% vs. 77.6%) tandis que la fonction `gaussian_filter()` en occupe plus (26% vs. 15.2%). Ces résultats faussent donc l'espérance d'accélération. Avec les résultats ci-dessus, la loi d'Amdahl prédit une accélération maximale de 2.1x du code complet avec une infinité de cœurs (contre le 4.3x de l'analyse prépartitionnement).

Dans un deuxième temps, le problème des communications intermodules se pose. En effet, avant même de prendre en compte les temps d'exécution matérielle, notre partitionnement rend le code plus lent, et ce, à cause des délais de communications (1 table en entrée et 2 tables en sortie).

La Figure 4.5-2 illustre bien que même en atteignant la limite théorique d'accélération du module matériel, le surcoût en communications empêche tout gain potentiel. Il est donc nécessaire de trouver une solution pour réduire les coûts de communication avant même de s'intéresser à l'accélération du module matériel.

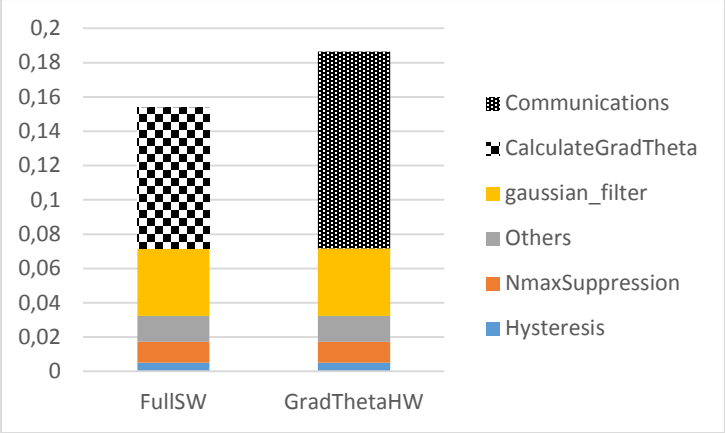


Figure 4.5-2 : Postes de dépense du temps d'exécution de la solution entièrement logicielle contre la solution avec partition matérielle

La troisième raison que nous pouvons avancer est l'utilisation de métriques peu adaptées pour définir quelles parties du code logiciel paralléliser. Nous nous sommes concentrés sur deux métriques lors de l'analyse de prépartitionnement : le potentiel de parallélisme et la part du temps d'exécution total d'un segment de code. Bien que ces métriques nous donnent effectivement des informations intéressantes sur le potentiel passage en matériel d'un segment de code, elles s'avèrent non suffisantes. Pour permettre une analyse plus précise, deux métriques devraient s'ajouter aux métriques déjà utilisées. La première est le coût de communication, c'est-à-dire le temps nécessaire selon l'architecture ciblée pour faire transiter les données entrantes et sortantes du coprocesseur. La seconde est le temps d'exécution logiciel de la partie destinée à passer en matériel. Si le coût de communication est plus grand que le temps d'exécution logiciel, l'analyse peut s'arrêter, car aucune accélération n'est possible. Cela explique pourquoi nos résultats étaient cohérents jusqu'au retrait de la fonction `atan()` de notre code et présentaient un potentiel d'accélération, comme illustré sur la Figure 4.5-3.

La figure 4.5-3 montre bien que, lorsque l'on considérait la solution avec `atan()`, le ratio coût de communication/temps d'exécution était avantageux et laissait présager une possible accélération. Si l'on vient rajouter à ce ratio les deux métriques utilisées originalement (portion du temps total et potentiel de parallélisme), on peut alors avoir un grand degré de confiance quant à la possible accélération de cette portion de code, et ce très tôt dans le processus de développement.

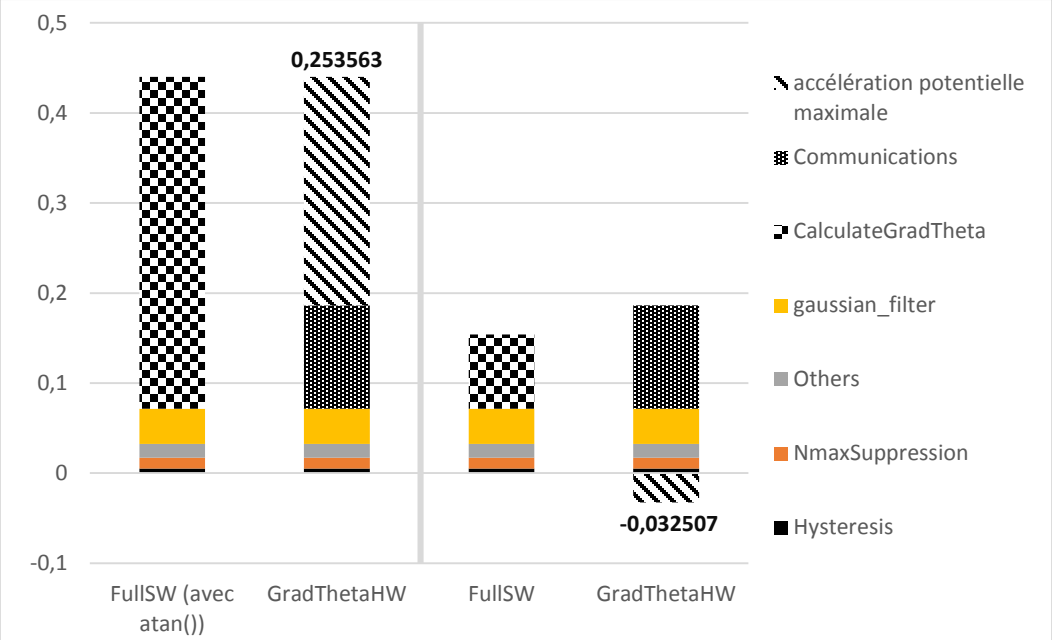


Figure 4.5-3 : Accélération potentielle du partitionnement avant et après retrait de la fonction atan() de la solution logicielle

L'équation (1) ci-dessous présente le gain de temps maximal atteignable avec une infinité de cœurs parallèles pour un segment de code et prend en compte les deux métriques présentées. L'équation (2) exprime l'accélération maximale de notre algorithme si le segment est accéléré infiniment. Cette équation exprime la relation entre la portion du temps passé dans le segment parallélisé et le temps total d'exécution de l'algorithme. Le potentiel de parallélisme vient quant à lui modérer la valeur finale de l'équation (2). On peut alors utiliser les équations (3) et (4).

(1) $$gain\ de\ temps\ potentiel\ maximal\ sur\ le\ segment = temps\ d'exécution\ logicielle\ du\ segment - coût\ des\ communications$$

(2) $$accélération\ potentielle\ maximale\ de\ l'algorithme = \frac{1}{1 - \frac{gain\ de\ temps\ potentiel\ maximal\ sur\ le\ segment}{Temps\ total\ d'exécution\ logicielle}}$$

(3) $$gain\ de\ temps\ sur\ le\ segment = \frac{temps\ d'exécution\ logicielle\ du\ segment}{potentiel\ de\ parallélisme} - coût\ des\ communications$$

$$(4)\ accélération\ de\ l'algorithme = \frac{1}{1 - \frac{gain\ de\ temps\ sur\ le\ segment}{Temps\ total\ d'éxécution\ logicielle}}$$

Enfin, la dernière raison des résultats que nous avons présentés repose sur le choix du processeur effectuant les tâches logicielles : un ARM Cortex-A9 cadencé entre 0.8 Ghz et 2 Ghz. Cela veut dire que dans le pire des cas, le processeur traite les informations ~5x plus vite que la partition matérielle (12.5x plus vite dans le meilleur des cas). Ainsi, pour qu'un partitionnement matériel soit valable, avant même de prendre en compte les coûts de communication, il faut pouvoir aller chercher une accélération assez importante pour contrebalancer la baisse de cadencement du FPGA

CHAPITRE 5 CONCLUSION ET RECOMMANDATIONS

5.1 Récapitulatif

Au cours de ce mémoire, nous avons pu répondre aux différents objectifs qui ont été posés lors de l'introduction.

Nous avons, dans un premier temps, proposé un flot ESL étendu à l'aide d'une solution ESL existante, SpaceStudio, sur laquelle nous sommes venus greffer une solution d'analyse de prépartitionnement, Pareon Profile. Le but de cette première analyse est de guider le processus de partitionnement en repérant les sections du code source les plus propices à un passage en coprocesseur.

Nous avons ensuite créé une solution logicielle permettant de relier le code source à l'environnement d'analyse du partitionnement en permettant la transformation d'artefacts C (fonction et corps de boucle) en modules SpaceStudio. Les décisions de partitionnement du code original sont motivées par l'analyse effectuée par Pareon Profile.

Cette solution logicielle C2Space utilise un transpilateur, Clang. Celui-ci permet la création d'un arbre de syntaxe abstraite du code source sur lequel il est possible de naviguer. Nous profitons de cette fonctionnalité pour rassembler des informations afin de préparer la transformation du code source. Une fois toutes les informations recueillies, C2Space crée un ensemble de fichiers permettant de continuer le travail du flot ESL sur la plateforme SpaceStudio.

Le reste du flot de travail est pris en charge par SpaceStudio. Durant cette phase, nous analysons tour à tour les temps d'exécution de la partie logicielle du code source (hébergée sur le processeur) et de la partie matérielle (synthétisée pour un FPGA par HLS Vivado). Nous analysons de plus les temps de communication générés par le partitionnement afin d'aboutir aux résultats finaux d'accélération.

Nous avons par la suite pu tester notre flot étendu sur un exemple : le filtre de détection de contours de Canny. Nous sommes partis d'une implémentation naïve de l'algorithme que nous avons raffinée lors de l'analyse de prépartitionnement. Cette analyse nous a aussi permis de repérer la section du code présentant le plus grand potentiel de parallélisme. Cette section est devenue la cible de notre logiciel de traduction afin de permettre de créer un projet SpaceStudio pour continuer

l'analyse. Bien que les résultats finaux n'exhibent pas d'accélération du code matériel, ce test nous a permis d'effectuer plusieurs constatations :

- Le flot lui-même est fonctionnel. Les différentes solutions logicielles s'intègrent bien les unes aux autres et l'outil de traduction C2Space exhibe le comportement attendu en permettant la création d'un projet SpaceStudio en une fraction du temps nécessaire pour créer celui-ci à la main.
- L'analyse de prépartitionnement présente des informations qui nous permettent d'exclure rapidement des schémas de partitionnement de l'espace des solutions à l'aide de deux métriques, la portion du temps total d'exécution et le potentiel de parallélisme.
- Les métriques présentées ci-dessus sont nécessaires, mais non suffisantes pour s'assurer d'une accélération réelle lors de l'étape de cosimulation. Deux autres métriques, liées à l'architecture ciblée permettent d'augmenter le degré de confiance : le coût des communications et le temps d'exécution logicielle.
- Le raffinement de l'algorithme original doit être fait en prenant en compte l'architecture ciblée pour que l'analyse de prépartitionnement fournisse des résultats pertinents. Cela est notamment le cas de l'approximation de l'arc tangente (qui aurait dû être efffectuée lors de l'analyse de prépartitionnement.

5.2 Recommandations

Comme nous avons pu le voir dans le chapitre précédent, les travaux effectués dans ce mémoire présentent encore bien des défis à relever, aussi bien du côté du flot de conception proposé que du côté du traducteur de code source. Nous passons en revue les principaux axes d'amélioration pour ces deux aspects.

5.2.1 Le flot de conception ESL

L'un des premiers aspects du flot de conception qui pourrait être amélioré est l'analyse de prépartitionnement. Comme expliqué dans ce mémoire, l'analyse de prépartitionnement permet une réduction de l'espace de solution, et ce, à un moment de la conception où les itérations sur le design étudié sont peu dispendieuses en temps. Il est peu réaliste de penser que cette étape d'analyse puisse, à elle seule, donner des résultats satisfaisants dans une conception de système électronique,

mais elle permet toutefois d'isoler des candidats prometteurs, de repérer des écueils de conception et de procéder à une première passe d'optimisation sur le design étudié.

Ces qualités pourraient être améliorées si nous prenions en compte l'architecture ciblée pour mener nos analyses. Le code pourrait être compilé pour la machine cible afin d'obtenir des temps comparables à ceux de la cosimulation. De plus, une approximation des délais de communication pourrait être effectuée lors de la phase d'analyse du potentiel de parallélisme pour s'assurer que le coût des communications ne réduise pas à néant l'accélération attendue.

Le deuxième axe d'amélioration du flot serait une intégration plus forte des différentes étapes de celui-ci. En effet, actuellement chacun des trois logiciels principaux utilisés (SpaceStudio, Pareon Profile et C2Space) fonctionne en tant qu'unité séparée, Pareon et C2Space ne fonctionnant que sous un système d'exploitation Unix, tandis que SpaceStudio est un logiciel destiné au système Windows. L'intégration d'un système d'analyse prépartitionnement au sein de la suite SpaceStudio serait donc bénéfique à la fluidité de la méthode de travail et permettrait de fournir une interface unifiée.

5.2.2 Le traducteur C2Space

La solution de traduction de code source que nous avons développée dans ce mémoire pourrait bénéficier de plusieurs améliorations.

Tout d'abord, nous pourrions augmenter le nombre de points d'entrée pour la modularisation. Actuellement, C2Space n'est capable de modulariser que deux artefacts C, les corps de boucle et les fonctions. Nous pourrions étendre cette fonctionnalité en proposant à l'utilisateur de définir une portée personnalisée à modulariser. Cela permettrait plus de contrôle sur les éléments à amener en matériel.

Un autre axe d'amélioration serait de raffiner les protocoles de communications. En effet, dans l'état actuel du logiciel, nous ne sommes pas capables d'éditer des protocoles optimaux selon le type de travail qu'effectue le module. Rendre ce processus entièrement automatique semble peu réalisable. On pourrait toutefois envisager de rendre cette fonctionnalité réglable par l'utilisateur. L'idée serait de créer un ensemble de schémas de communication types, utilisés régulièrement dans l'industrie, et de laisser l'utilisateur choisir quel type de communication affecter à telle paire de modules. Cela demanderait toutefois une charge de travail de formalisation conséquente afin de

rationaliser les éléments caractérisant chacun de ces schémas. Ajouter cette fonctionnalité à l'outil pourrait toutefois améliorer grandement ses performances lors de l'analyse de partitionnement.

Enfin, le dernier axe d'amélioration serait de développer une véritable interface graphique permettant de créer différents scénarios de partitionnement, de les sauvegarder et de les mettre en rapport avec la source originale. Des informations complémentaires pourraient y être affichées comme l'arbre d'appel des différentes fonctions ou encore les informations de l'analyse de prépartitionnement.

BIBLIOGRAPHIE

[1] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE,* vol. 86, pp. 82-85, 1998.

[2] G. E. Moore, "Moore's Law at 40," *Understanding Moore's law: four decades of innovation,* 2006.

[3] H. D. Foster, "Why the design productivity gap never happened," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, 2013, pp. 581-584.

[4] A. Dewey, "The VHSIC Hardware Description Language," *VLSI Design,* vol. 5, pp. 33-9, 11/ 1984.

[5] M. Shahdad, "An overview of VHDL language and technology," presented at the Proceedings of the 23rd ACM/IEEE Design Automation Conference, Las Vegas, Nevada, USA, 1986.

[6] D. J. Smith, "VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C," in *Proceedings of 33rd Design Automation Conference, 3-7 June 1996*, New York, NY, USA, 1996, pp. 771-6.

[7] M. R. Barbacci and D. P. Siewiorek, "Automated exploration of the design space for register transfer (RT) systems," presented at the Proceedings of the 1st annual symposium on Computer architecture, 1973.

[8] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. Thomas, Jr., "A design methodology and computer aids for digital VLSI systems," *Circuits and Systems, IEEE Transactions on,* vol. 28, pp. 634-645, 1981.

[9] J. Cong, L. Bin, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhiru, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 30, pp. 473-491, 2011.

[10] B. K. Fawcett, "Digital logic synthesis-an overview," in *Proceedings of WESCON '93, 28-30 Sept. 1993*, New York, NY, USA, 1993, pp. 62-4.

[11] R. Camposano, "From behavior to structure: high-level synthesis," *Design & Test of Computers, IEEE,* vol. 7, pp. 8-19, 1990.

[12] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE,* vol. 85, pp. 349-365, 1997.

[13] G. Martin, B. Bailey, and A. Piziali, *ESL design and verification: a prescription for electronic system level methodology*: Morgan Kaufmann, 2010.

[14] J. Kriegel, A. Pegatoquet, M. Auguin, and F. Broekaert, "A performance estimation flow for embedded systems with mixed software/hardware modeling," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI), 18-21 July 2011*, Piscataway, NJ, USA, 2011, pp. 174-81.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, *et al.*, "The ASTREE analyzer," in *Programming Languages and Systems. 14th European Symposium on Programming, ESOP 2005. Held as Part of the Joint European Conferences on Theory and*

Practice of Software, ETAPS 2005. Proceedings, 4-8 April 2005, Berlin, Germany, 2005, pp. 21-30.

[16] *Pareon profile*. Available: https://www.vectorfabrics.com/en/blog/item/speed_up_bullet_by_30_in_3_minutes

[17] M. Anderson and S. K. Shukla, "APECS code synthesis: Extending ocarina for multi-threaded code synthesis from AADL models for Safety Critical applications," in *11th IEEE International Conference on Networking, Sensing and Control, ICNSC 2014, April 7, 2014 - April 9, 2014*, Miami, FL, United states, 2014, pp. 36-41.

[18] L. Filion, M.-A. Cantin, L. Moss, E. Aboulhamid, and G. Bois, "Space codesign: A systemC framework for fast exploration of hardware/software systems," in *Proceedings of the Design and Verification Conference and Exhibition*, 2007.

[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the FREENIX/Open Source Track. 2005 USENIX Annual Technical Conference, 10-15 April 2005*, Berkeley, CA, USA, 2005, pp. 41-6.

[20] M. Keaveney, A. McMahon, N. O'Keeffe, K. Keane, and J. O'Reilly, "The development of advanced verification environments using system Verilog," in *IET Irish Signals and Systems Conference, ISSC 2008, June 18, 2008 - June 19, 2008*, Galway, Ireland, 2008, pp. 325-330.

[21] K. Gruttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, *et al.*, "An ESL timing and power estimation and simulation framework for heterogeneous socs," in *14th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2014, July 14, 2014 - July 17, 2014*, Samos, Greece, 2014, pp. 181-190.

[22] C. Economakos, H. Sidiropoulos, and G. Economakos, "Rapid prototyping of digital controllers using FPGAs and ESL/HLS design methodologies," in *2013 19th International Conference on Automation and Computing (ICAC), 13-14 Sept. 2013*, Piscataway, NJ, USA, 2013, p. 6 pp.

[23] L. Moss, H. Guerard, G. Dare, and G. Bois, "An ESL methodology for rapid creation of embedded aerospace systems using hardware-software co-design on virtual platforms," in *SAE 2012 Aerospace Electronics and Avionics Systems Conference, AEAS 2012, October 30, 2012 - November 1, 2012*, Phoenix, AZ, United states, 2012.

[24] *Intel Cofluent Studio*. Available: http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html

[25] J. P. Calvez, O. Pasquier, D. Isidoro, and D. Jeuland, "CoDesign with the MCSE methodology," in *Proceedings of Twentieth Euromicro Conference. System Architecture and Integration, 5-8 Sept. 1994*, Los Alamitos, CA, USA, 1994, pp. 19-26.

[26] *Clang*. Available: http://clang.llvm.org/

[27] *Zedboard Specifications*. Available: http://zedboard.org/sites/default/files/product_briefs/PB-AES-Z7EV-7Z020_G-v12.pdf

[28] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, pp. 215-226, 2000.

[29] *ROSE compiler infrastructure*. Available: http://rosecompiler.org/

[30] P. Cantiello, B. Di Martino, and F. Piccolo, "Unimodular loop transformations with source-to-source translation for gpus," in *Algorithms and Architectures for Parallel Processing*, ed: Springer, 2013, pp. 186-195.

[31] *Unimodular Loop Transformation*. Available: http://link.springer.com/referencework/10.1007%2F978-0-387-09766-4

[32] R. Aloor and V. K. Nandivada, "Unique Worker model for OpenMP," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 47-56.

[33] C. Lattner. (2007). *New LLVM C front-end: "clang"*. Available: http://lists.llvm.org/pipermail/cfe-dev/2007-July/000000.html

[34] A. Kaushik and H. D. Patel, "Systemc-clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Specification & Design Languages (FDL), 2013 Forum on*, 2013, pp. 1-8.

[35] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and W. Zhanyong, "Large-Scale Automated Refactoring Using ClangMR," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 548-551.

[36] *ISO/IEC 9899:2011*. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853

[37] L. Daibin. (2009). *LibBmp*. Available: https://code.google.com/archive/p/libbmp/

ANNEXES

ANNEXE A – ARCHITECTURE DE C2SPACE

A-1.Paquetage gestionnaire

Ce paquetage a pour responsabilité d'assurer l'interface entre les données et le reste du programme. Trois classes sont implémentées dans ce paquetage, chacune gérant un aspect différent du stockage de données. Le point commun entre ces trois classes est qu'elles sont implémentées en respectant le patron de conception Singleton. Ainsi, il est impossible d'instancier plus d'une fois chacune de ces classes. Cela permet de s'assurer que les données ne sont pas segmentées par une erreur de manipulation. Le seul moyen d'accéder à l'instance est d'utiliser la méthode statique `getInstance()` présente dans chaque classe.

A-1.1.ModuleManagerSingleton

Cette classe est responsable du maintien des informations sur les éléments du code source destinés à être modularisés. Ses responsabilités spécifiques sont :

1. De stocker l'intégralité des instances de `ModuleInfos` créées et d'en permettre l'accès aux autres éléments du programme
2. De classifier les fichiers sources d'origine et de maintenir les informations sur les fichiers inclus par ces sources
3. D'imprimer des rapports détaillés sur l'état actuel des données stockées

Nous avons réalisé les objectifs 1. et 2. À l'aide d'un `std::map` liant respectivement le nom de la fonction et le nom du fichier au `ModuleInfos` et au vecteur d'`includes` correspondants. Cela nous permet de retrouver rapidement les informations qui nous intéressent à l'aide d'identifiants facilement récupérables via l'interface de programmation de Clang.

Des accesseurs permettent de récupérer la référence vers le `ModuleInfos` ou le vecteur d'`includes` qui nous intéresse.

Enfin l'objectif 3. est implémenté sous la forme d'une série de méthodes permettant l'affichage de différents rapports d'exécution.

A-1.2.TableMemoryManagerSingleton

Cette classe a pour but de maintenir les informations sur les tables de données utilisées par le code source dans le cas où celles-ci doivent être déplacées dans une mémoire externe. Ces informations ne peuvent être contenues dans les instances de la classe `ModuleInfos`, car elles sont transversales à celles-ci. Les responsabilités spécifiques de `TableMemoryManagerSingleton` sont :

1. Maintenir les informations permettant de relier le nom d'une table dans la déclaration d'une fonction, le nom de cette même table lors de l'appel de ladite fonction et l'emplacement réel en mémoire de ces données
2. Générer une table de décalage sommaire pour éviter que les données placées en mémoire externe ne se recouvrent

Le premier objectif est réalisé à l'aide d'une série de structures. La première de ces structures est `s_table`, qui représente une table de données définie dans la signature d'une fonction. La seconde est `s_callingTable`, qui représente la table de données utilisée en tant que paramètre d'appel d'une fonction. Elle possède une référence vers la `s_table` correspondante. Nous avons aussi implémenté une structure en arbre `callingTableTree` permettant de contextualiser les `s_callingTable`. Cette structure est basée sur une série de `std::map` imbriqués et permet de rechercher une table d'appel précise en fonction de l'appelant, l'appelé et la position dans la source de l'appel. Ces trois niveaux sont nécessaires pour éviter les ambiguïtés lors de la reconstruction des modules. L'extrait A-1-1 illustre l'utilisation de cet arbre.

De la même manière, nous avons créé une structure pour contextualiser les `s_table`, `m_calledTableList`, qui est un `std::map` mettant en relation le numéro de l'argument dans la signature de la fonction et le nom de la fonction d'appel.

```
1  //fileA.c
2  void funcNameA() {
3      int table1[100];
4      int table2[100];
5      //Some treatment
6      funcNameB(&table1); //We try to identify this callingTable
7      //Some more treatment
8      funcNameB(&table2); //Not this one
9  }
```

```
1  //Usage of callingTableTree :
2  s_callingTable *table;
3  table = callingTableTree[funcNameA][FuncNameB][239];
4  //funcNameA : Name of the function responsible for the call
5  //funcNameB : Name of the function that has been called
6  //239 : Raw position of the start of the call. Used to disambiguate
   in case of multiples calls in the same function.
```

Extrait A-1-1 : Utilisation de callingTableTree

Pour réaliser le second objectif, relier les noms de tables d'appel aux emplacements mémoires correspondants, nous avons défini un autre `std::map` reliant cette fois-ci fonction appelante, nom de table et emplacement en mémoire appelé `m_defineMap`. Il est alors facile, pour une table donnée, de retrouver l'emplacement mémoire approprié. L'extrait suivant récapitule les trois structures de données présentées ci-dessus :

```
1  Typedef
2  std::map<std::string /*callingModule*/,
3    std::map<std::string /*CalledFunction*/,
4      std::map<unsigned /*Raw location of the call*/,
5        std::list< s_newCallingTable >>>> callingTableTree;
6
7  std::map<std::string /*FunctionName*/,
8    std::map<unsigned /*argNum*/,
9      s_table>> m_calledTableList;
10
11 std::map<std::string /*CalledFunction*/,
12   std::map<std::string /*nameofthetable*/,
13     std::string/*defineOffset*/>> m_defineMap;
```

Extrait A-1-2 : Structures de données de TableMemoryManagerSingleton

`TableMemoryManagerSingleton` implémente plusieurs méthodes destinées à construire les fichiers d'entête nécessaires. Les fichiers d'entête générés contiennent une série de définitions préprocesseurs liant les noms de table aux décalages correspondants dans la mémoire. Cela permet de conserver les notations employées par les développeurs du code source original plutôt que de

les remplacer par des décalages opaques. Les fonctions de communications nécessaires pour interagir avec la mémoire externe sont générés dans les fichiers modules concernés.

A-1.3.ConfigFileManagerSingleton

Le gestionnaire de fichiers de configuration est chargé du contrôle des fichiers de paramétrage. Au cours de l'utilisation normale de C2Space, l'utilisateur doit à plusieurs reprises entrer des informations pour guider la reconstruction du code source. Toutefois, ce processus peut être long et nous avons donc implémenté un système de sauvegarde de paramètres. Ainsi, le développeur n'a pas besoin d'entrer ses paramètres plus d'une fois. Les responsabilités de cette classe sont :

1. Proposer une interface à l'utilisateur pour lui permettre la sauvegarde et le chargement des fichiers de configuration.
2. Générer les fichiers de configuration.
3. Permettre de substituer facilement les saisies utilisateur par les données enregistrées.

L'interfaçage avec l'utilisateur est assuré par une méthode appelée au démarrage du programme. L'utilisateur est sollicité pour fournir le nom du fichier qu'il désire charger ou bien le nom du nouveau fichier de configuration à créer. La classe s'assure que le fichier existe et que son ouverture se soit bien accomplie.

Les fichiers de configuration sont implémentés en enregistrant dans un fichier chaque saisie de l'utilisateur, en changeant de ligne après chaque donnée. Ce format a pour intérêt de simplifier la réalisation du troisième objectif.

En effet, afin de permettre de substituer simplement la saisie utilisateur par les données du fichier de configuration, nous utilisons une des fonctions de la bibliothèque standard C++, qui permet de substituer les tampons des flux standards. Ainsi, si l'utilisateur désire utiliser un fichier de configuration, nous remplaçons juste le tampon que consomme le flux standard `std::cin` par le tampon du flux du fichier de configuration. L'opérateur `<<` va alors récupérer les données du fichier, ligne par ligne, au fur et à mesure de l'exécution au lieu d'attendre une saisie utilisateur. Cette solution permet de limiter la quantité de code requise. Une référence vers le tampon original est maintenue au sein de la classe et est réattribuée au flux `cin` dès que le fichier de configuration est fermé.

A-2.Paquetage Structures de données

Le but du paquetage Structures de données et de fournir un ensemble de classes et de structures permettant le stockage et l'accès simple aux données récupérées lors de l'analyse du code source. Deux classes principales sont contenues dans ce paquetage : la classe `ModuleInfos` et la classe `OperationTree`.

A-2.1.ModuleInfos

La classe `ModuleInfos` est une classe permettant de contenir toutes les informations utiles d'un module. Ces informations sont stockées dans des structures propres à la classe. Une série de méthodes publiques est fournie pour interagir avec ces données. Ces méthodes sont directement utilisables par le développeur après avoir récupéré la référence vers une des instances de la classe `ModuleInfos` via le `ModuleManagerSingleton` vu plus haut. Les responsabilités spécifiques de cette classe sont :

1. Maintenir les informations nécessaires à la traduction d'un module de manière organisée
2. Maintenir une référence vers le constructeur approprié (selon le type de module à créer)
3. Permettre l'accès aux différentes informations par d'autres modules

Le maintien des informations dans un `ModuleInfos` est accompli grâce à une série de structures de données présentées dans les extraits suivants :

```
1  struct s_varDeclStruct {
2    std::string                        varName;
3    std::string                        varType;
4    std::string                        initString;
5    std::pair<unsigned, unsigned>      initRange;
6    std::pair<unsigned, unsigned>      varDeclRange;
7  };
```

Extrait A-2-1 : Structure représentant une déclaration de variable

```
1  struct s_interactionStruct {
2    std::string                                          interactionType;
3    std::string                                          interactorName;
4    std::string                                          returnType;
5    unsigned                                             binaryOperatorStartLoc;
6    unsigned                                             callStartLoc;
7    unsigned                                             callEndLoc;
8    unsigned int                                         nbArgs;
9    std::vector<bool>                                    isTableArg;
10   std::vector<std::pair<std::string, std::string>>     args;
11 };
```

Extrait A-2-2 : Structure représentant une interaction entre un module appelant et un module appelé

```
1  struct s_forStruct{
2    bool                                                 isRoot;
3    std::string                                          forVarName;
4    std::string                                          forVarType;
5    unsigned                                             childFor;
6    unsigned                                             parentFor;
7    std::vector<std::string>                             readVars;
8    std::vector<std::string>                             writtenVars;
9    std::vector<std::pair<std::string, std::string>>     writtenTables;
10   unsigned                                             initValue;
11   unsigned                                             endValue;
12   unsigned                                             nbIterations;
13   std::pair<unsigned, unsigned>                        initRange;
14   std::pair<unsigned, unsigned>                        condRange;
15   std::pair<unsigned, unsigned>                        incRange;
16   std::pair<unsigned, unsigned>                        bodyRange;
17   std::pair<unsigned, unsigned>                        globalForRange;
18 };
```

Extrait A-2-3 : Structure représentant une instance de boucle

```
1  //Spécifie la manière dont cette variable est utilisée au sein de la fonction
2  enum FlowDirection { in = 1, out, inout };
3   //Spécifie la manière dont les données vont transiter entre modules
4  enum TransferType { ModuleX, DeviceX };
5  struct s_arg
6  {
7    std::string                                  varName;
8    std::string                                  varType;
9    unsigned                                     ArgNbr;
10   unsigned                                     nbrOfElements;
11   TransferType                                 transferMethod;
12   FlowDirection                                flow;
13   s_arg(std::string argName, std::string argType, unsigned argNum, unsigned
   nbrOfElements, TransferType transferMethod, FlowDirection flow)
14   : varName(argName), varType(argType), ArgNbr(argNum),
   nbrOfElements(nbrOfElements), transferMethod(transferMethod), flow(flow) {}
15 };
```

Extrait A-2-4 : Structure représentant un des arguments de la fonction analysée

Les quatre structures de données présentent des similarités. Tout d'abord, ces structures représentent chacune une instance d'éléments que nous pouvons retrouver dans une fonction analysée, respectivement les déclarations de variable, les appels à des fonctions DAM, les boucles et les arguments. De plus, les trois premières structures contiennent des positions, sous la forme de portées, qui nous permettent à tout moment de retrouver l'emplacement dans le code source de l'instance en question. En plus de ces positions, chaque structure contient des informations spécifiques à l'élément qu'elle représente.

Pour les déclarations de variables de l'extrait A-2-1, nous gardons en mémoire le nom et le type de variable ainsi que la valeur de leur initialisation s'il y en a une.

Pour les interactions intermodules (extrait A-2-2), en plus de garder les positions de début et de fin d'appel à la fonction correspondante, nous enregistrons la position d'un potentiel opérateur binaire dans le cas où la fonction est utilisée au sein d'une expression mathématique ou d'une assignation. Nous enregistrons ensuite le type d'interaction (est-ce une interaction « appelant » ou « appelé ») ainsi que le nom de l'interacteur. Nous enregistrons enfin les informations concernant les paramètres de la fonction : le type du retour ainsi que le nombre, le nom et le type des arguments d'appel. Nous enregistrons enfin une table de correspondance avec les arguments qui précise si les arguments sont à considérer comme des tables.

En ce qui concerne les instances de boucles (extrait A-2-3), un nombre d'informations assez important est gardé en mémoire. Tout d'abord, nous enregistrons la position de la boucle dans un potentiel nid de boucle : est-elle à la racine, quel est son parent, quel est son enfant ? Puis nous enregistrons toutes les informations contenues dans l'appel de boucle. Nous considérons une structure de boucle classique (exemple : `for( int i = 0; i < 100; i++)`) et nous enregistrons donc le nom et le type de la variable d'incrément, son initialisation et sa valeur finale. Nous enregistrons aussi un certain nombre d'informations sur le corps de la boucle : les noms des variables écrites et lues ainsi que les noms et indices des tables lues.

Enfin, pour chaque argument dans la signature de la fonction analysée, nous gardons en mémoire plusieurs informations, dont son nom, son type, la quantité d'éléments qu'il contient, ainsi que sa position dans la signature. Deux informations sont stockées dans des énumérations, pour une possible extension et pour plus de clarté : la direction des données (entrante, sortante ou les deux) ainsi que le procédé utilisé pour communiquer cette donnée (pour l'instant limité aux interfaces

`ModuleRead/Write` ou `DeviceRead/Write`). Ces structures permettent de stocker les informations d'une seule instance. Puisque chaque `ModuleInfos` est susceptible de contenir plus d'une de ces instances, elles sont stockées dans des conteneurs standards listés ci-après.

```
1  std::vector<OperationTree*> m_assignmentList;
2  std::vector<std::pair<std::string, std::pair<unsigned, unsigned>>>
   m_returnLocationsList;
3  std::vector<s_interactionStruct> m_interactionsList;
4  std::map<unsigned, s_forStruct> m_forList;
5  std::vector<s_varDeclStruct> m_varDeclList;
6  std::map<std::string/*Name*/, s_arg* /*arg Occurence*/> m_args_by_name;
7  std::map<unsigned/*ArgNbr*/, s_arg* /*arg Occurence*/> m_args_by_nbr;
```

Extrait A-2-5 : Conteneurs utilisés dans la classe ModuleInfos

Les lignes 3, 4, 5, 6 et 7 sont les conteneurs utilisés pour les structures définies ci-dessus. Deux conteneurs supplémentaires sont utilisés pour stocker les données correspondantes aux assignements dans le corps de la fonction DAM ainsi que les positions des clauses de retour, respectivement `m_assignmentList` et `m_returnLocationsList`. Notez que le type `OperationTree` utilisé dans le conteneur d'assignements est expliqué dans la section suivante.

Le reste des données pertinentes sont stockées dans des variables simples, par exemple le nom de la fonction analysée ou encore le type de retour de celle-ci.

Afin de répondre à la seconde responsabilité énoncée au début de cette section, qui est de maintenir une référence vers le constructeur approprié (selon le type de module à créer), nous avons utilisé un patron de conception « Stratégie ». La stratégie de construction du module est définie dans une classe de construction spécifique. Nous attachons une instance de cette classe à notre classe `ModuleInfos`. Lorsque la construction doit être effectuée, la classe `ModuleInfos` fournit le constructeur approprié.

Enfin pour répondre à la dernière responsabilité, qui était de permettre aux autres classes l'accès à toutes ces données, nous mettons en place une série de méthode d'accès, soit élément par élément pour les processus ciblés, soit d'un conteneur complet pour les processus itératifs.

A-2.2.OperationTree

La classe `OperationTree` permet le stockage d'informations sur les opérations décrites dans la source, comme une opération arithmétique ou un assignement. Une instance de cette classe va

représenter un nœud de l'arbre d'opérations complet. On peut alors visiter l'arbre en partant du nœud racine. Cette classe a pour responsabilités :

1. Pouvoir stocker les informations utiles à chaque nœud et décrire tous les types potentiellement accessibles dans une opération
2. S'assembler en une structure d'arbre
3. Permettre des actions variées sur l'arbre

Afin de répondre à la première responsabilité de la classe, nous avons défini un certain nombre de variables membres contenant l'information sur le nœud actuel :

```
1  TreeNodeType                        m_typeOfNode;
2  std::pair<unsigned, unsigned>       m_range;
3  std::string                         m_data;
4  OptionalInfos                       m_optInfos;
```

Extrait A-2-6 : Variables membres contenant des données au sein de la classe OperationTree

Comme nous pouvons le constater, chaque nœud n'enregistre que peu d'informations. La première de celles-ci est le type du nœud. Nous avons créé une énumération simplifiée des types que propose Clang pour créer notre propre nomenclature (nous avons abandonné certains types peu intéressants comme les passages implicites de `RValue` à `LValue`). L'extrait suivant présente cette énumération.

```
1  enum TreeNodeType {
2  CallExpr, UnaryOperator, DeclRefExpr, BinaryOperator, ArraySubscriptExpr,
3  IntegerLiteral, FloatingLiteral, ParenExpr, Error
4  };
```

Extrait A-2-7 : Énumération des types de nœuds possibles dans le cadre d'une opération

La seconde des informations présentes dans l'extrait A-2-6 est la portée du nœud, c'est-à-dire les emplacements de début et de fin dans le code source que couvre ce nœud. La troisième information contenue dans un nœud est tout simplement la donnée que représente le nœud. Il faut noter que cette variable n'a pas toujours de sens selon le type de nœud (par exemple un nœud de type `ParenExpr` ne contient pas de donnée). Le dernier champ `m_optInfos` est réservé à certains types de nœuds qui nécessitent plus d'informations, par exemple l'opérateur unaire pour lequel on doit stocker s'il est postfixe ou préfixe.

Afin de permettre l'assemblage des nœuds en un arbre de données (responsabilité n°2), nous construisons des variables membres permettant de créer un arbre de références double chainé :

```
1  OperationTree                    *left;
2  OperationTree                    *right;
3  OperationTree                    *parent;
4  std::vector<OperationTree*>      optArgs;
```

Extrait A-2-8 : Variables membres permettant l'accès aux nœuds adjacents dans l'arbre de la classe OperationTree

Chaque nœud maintient une référence vers son parent et vers ses deux enfants (gauche et droite). Dans les rares cas où le nœud possède plus de deux enfants, nous utilisons la structure alternative `optArgs` (en réalité ce membre n'est utilisé que pour les nœuds représentant un appel de fonction). Une méthode `insertNode()` est aussi fournie afin de rajouter des nœuds dans un arbre en construction. L'arbre suivant est un exemple de comment cette classe permet de décomposer une opération.

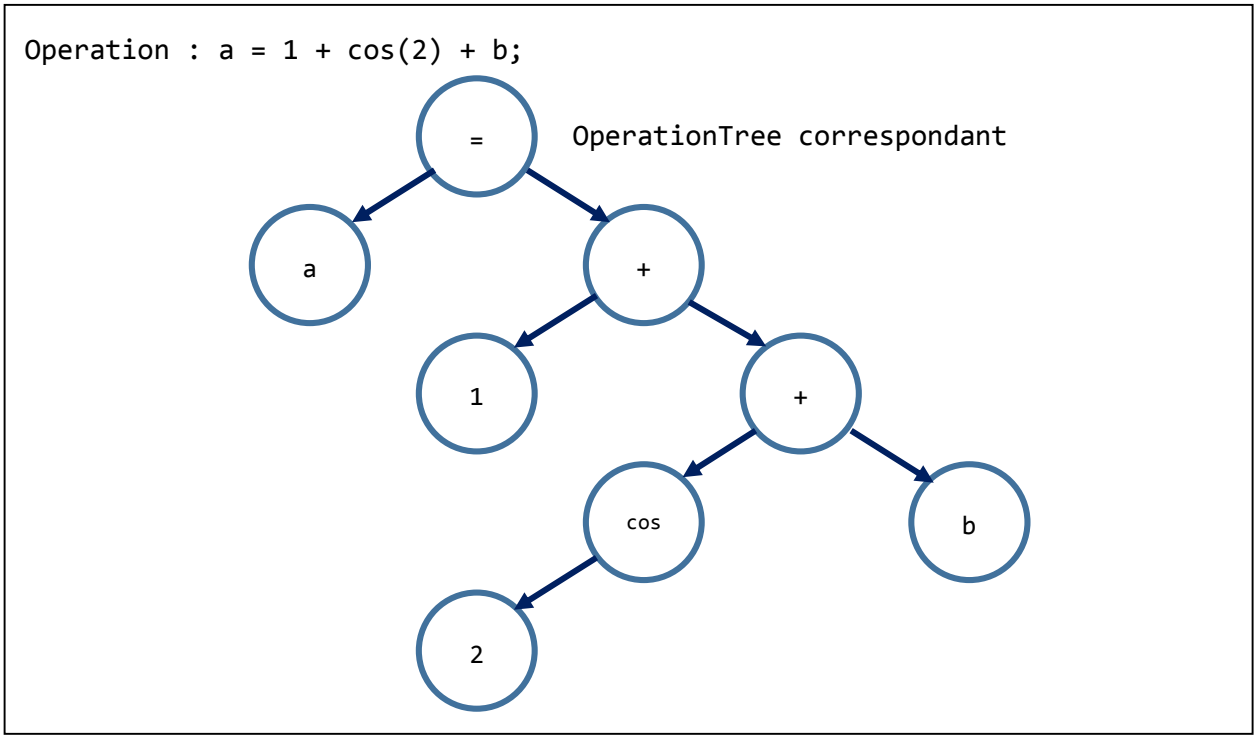


Figure A-2-1 : Exemple d'arbre créé avec la classe OperationTree

Enfin, afin de pouvoir effectuer des actions sur ces arbres d'opérations, nous implémentons une méthode appelée `visitTree()`, qui agit simplement comme une méthode générique appelant une fonction spécifique selon le type du nœud. Ces fonctions spécifiques sont encore une fois implémentées dans des classes séparées en suivant le patron « Stratégie » utilisé dans la classe

`ModuleInfos`. La méthode `acceptvisitor()` permet de sélectionner le type de visite à effectuer sur l'arbre d'opération. Cette méthode est statique pour que le visiteur soit appliqué à tous les nœuds. Une fois un visiteur choisi, lors de l'appel à `visitTree()`, le nœud utilisera le visiteur attaché afin d'appeler les méthodes spécifiques à chaque nœud. La plupart des visiteurs existants sont présentés sur la figure suivante ainsi que leurs interactions avec la classe `OperationTree`. Nous n'avons affiché que les méthodes et membres pertinents de la classe `OperationTree`.

Il existe actuellement 5 visiteurs possédant des compétences distinctes. Le visiteur `Print` permet d'afficher l'expression originale contenue dans l'arbre. Le visiteur `ReadVars` permet de générer la liste des variables lues et des tables lues avec leurs indices dans un arbre donné. Le visiteur `WrittenVars` génère le même type d'informations pour les variables écrites. Le visiteur `ReplaceCalls` permet de modifier la source dans le cas où un appel de fonction utilise une des variables enregistrées en mémoire externe. Enfin le Visiteur `ReplaceAssignment` permet la réécriture des assignements dans le cadre de la reconstruction d'un module à boucles (voir `LoopModuleBuilder` section A-4.3).

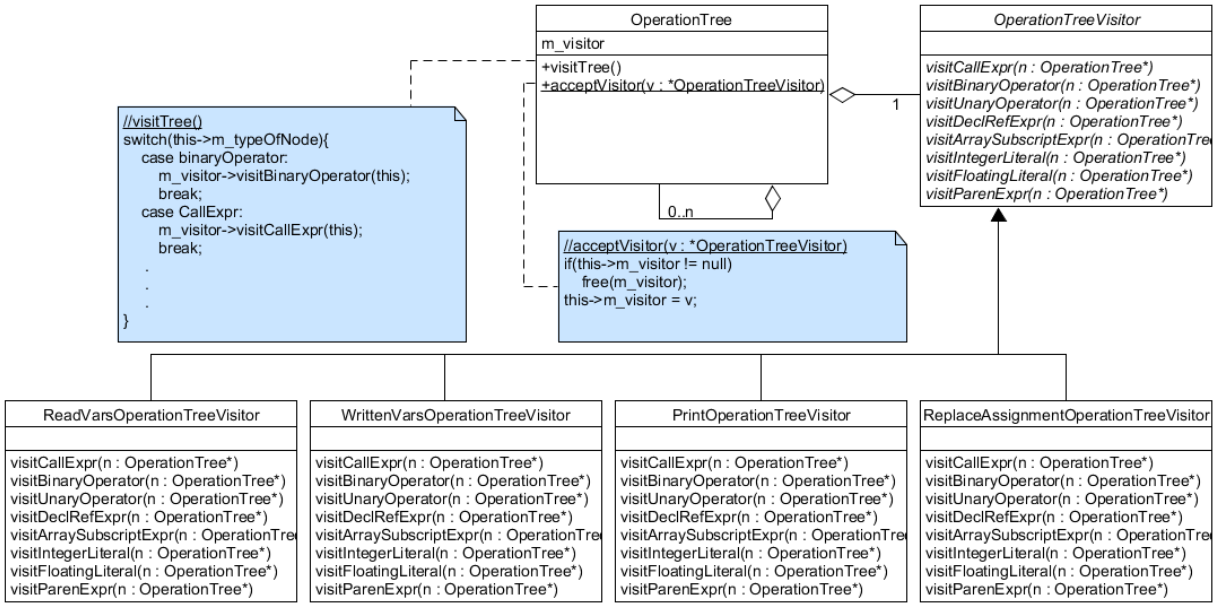


Figure A-2-2 : Diagramme de classe de la relation Arbre d'opérations/Visiteur

A-3.Paquetage action à la compilation

Le paquetage Action utilise extensivement les constructions de la bibliothèque Clang afin d'accomplir des tâches visant certaines parties de l'arbre de syntaxe abstraite produit par le compilateur. Il contient une série d'actions spécialement désignées pour récupérer ou utiliser les informations pertinentes au formatage du code source en modules SpaceStudio. À chaque action est associé un patron de correspondance permettant de viser certains éléments de l'arbre. Ces patrons seront présentés en même temps que l'action correspondante. Les actions sont présentées dans l'ordre dans lequel elles interviennent dans C2Space.

A-3.1.FindIncludesAction

Cette action est basée sur la combinaison `ASTFrontendAction`/`ASTConsumer` vue lors de la présentation de Clang. C'est la seule action construite de cette manière. Ce choix est motivé par le fait que les patrons de correspondances ne permettent pas de récupérer les directives de préprocesseurs. En effet, cette structure nous permet de rajouter une classe exécutant une action sur le code avant le préprocess, la classe `clang::PPCallbacks`. Nous basons donc notre classe `FindIncludesAction` sur cette classe.

La responsabilité de cette classe est de traverser toutes les directives `#include` du fichier source et de les rajouter aux données maintenues par la classe `ModuleManagerSingleton`. L'intérêt de maintenir ces informations est de pouvoir facilement réinsérer les `#include` pertinents dans les modules.

A-3.2.FindFunctionsAction

Cette classe a pour but d'aller rechercher les fonctions DAM définies par l'utilisateur au sein des fichiers sources. Pour ce faire, nous utilisons le patron de correspondance suivant :

```
1  clang::ast_matchers::DeclarationMatcher FunctionsMatcher =
2  functionDecl(
3    isExpansionInMainFile()
4  ).bind("functionDecl");
```

Extrait A-3-1 : Patron de correspondance de fonction

Ce patron nous permet de repérer toutes les définitions de fonctions présentes directement dans le fichier source. La spécification `isExpansionInMainFile()` rejette tous les résultats ne faisant pas partie du fichier source avant expansion des directives de préprocesseur. Le point d'attache de ce

patron est la déclaration de fonction (`clang::FunctionDecl`). Nous profitons de ce point d'attache pour comparer le nom de la fonction avec la liste des fonctions DAM. Si la fonction repérée fait partie des fonctions DAM, nous créons un fichier séparé contenant la fonction complète afin de simplifier les traitements suivants.

Nous profitons aussi de cette action pour récupérer les arguments de la fonction afin de remplir les champs pertinents dans l'instance de `ModuleInfos` correspondante. Si certains des paramètres sont de type pointeur, nous demandons à l'utilisateur de fournir des informations supplémentaires sur ceux-ci : le statut de communication du paramètre (entrant sortant ou les deux) ainsi que la quantité de données que le pointeur référence. Ces données sont enregistrées dans la classe `TableMemoryManagerSingleton`.

À partir de la prochaine action, les fichiers sources originaux ne sont plus utilisés et nous utilisons l'ensemble de fichiers générés contenant les fonctions DAM isolées.

A-3.3.AnalyzeForAction

La classe de recherche de boucles for doit retrouver toutes les boucles contenues dans une des fonctions DAM. Pour répondre à ce besoin, nous utilisons le patron de correspondance présenté ci-après :

```
1  clang::ast_matchers::StatementMatcher ForMatcher =
2  anyOf(
3    forStmt(
4      hasAncestor(
5        forStmt().bind("forStmtAncestor")
6      )
7    ).bind("forStmt"),
8    forStmt(
9      unless(
10       hasAncestor(
11         forStmt().bind("forStmtAncestor")
12       )
13     )
14   ).bind("forStmtWithoutParent")
15 );
```

Extrait A-3-2 : Patron de correspondance de boucles

Ce patron, bien qu'un peu complexe, a l'avantage de nous donner des informations supplémentaires sur le contexte de la boucle en une seule expression. Le patron va donner lieu à deux sortes de résultats. Le premier type de résultat est une boucle sans parent (ce résultat est pris en charge par

l'ancre `forStmtWithoutParent`). Le second résultat est une boucle contenue dans le corps d'une autre boucle. Dans le cadre de ce résultat, nous avons deux ancres : `forStmtAncestor` et `forStmt`.

Lors de l'action, nous examinons la descendance dans l'arbre de syntaxe abstraite de nos points d'ancrage afin de récupérer des informations pertinentes sur chacune des boucles présentes dans la fonction analysée. Nous récupérons ainsi l'initialisation, la condition ainsi que l'incrément de la boucle. Nous profitons aussi des informations de contexte pour créer une hiérarchie au sein des boucles imbriquées. Toutes ces informations sont stockées dans des structures `s_forStruct` (voir section A-2.1) puis ajoutées à l'instance de `ModuleInfos` correspondante.

A-3.4.FindSplitLoopParametersAction

Cette action a pour but de modifier la source en amont des constructeurs afin de permettre la séparation d'un corps de boucle du reste d'une fonction. Cette action a besoin de deux patrons de correspondance différents :

```
1  clang::ast_matchers::DeclarationMatcher  varDeclMatcher =
2  varDecl(isExpansionInMainFile()).bind("varDecl");
3  clang::ast_matchers::StatementMatcher AssignmentMatcher =
4  binaryOperator(
5    anyOf(
6      hasOperatorName("="),
7      hasOperatorName("*="),
8      hasOperatorName("/="),
9      hasOperatorName("%%="),
10     hasOperatorName("+="),
11     hasOperatorName("-="),
12     hasOperatorName("<<="),
13     hasOperatorName(">>="),
14     hasOperatorName("&="),
15     hasOperatorName("^="),
16     hasOperatorName("|=")
17   )
18 ).bind("assignmentOperation");
```

Extrait A-3-3 : Patrons de correspondance pour l'action de séparation de boucle

Le premier patron, `varDeclMatcher` nous permet de récupérer le nom de toutes les fonctions déclarées dans le corps de la boucle que l'utilisateur a sélectionnée, tandis que le second nous permet de récupérer une entrée sur tous les assignements dans la boucle sélectionnée. Pour chaque assignement, nous créons un `OperationTree` que nous visitons ensuite avec les visiteurs `WrittenVars` et `ReadVars` afin de générer les listes des variables écrites et lues dans le corps de

boucle. Une fois en possession de nos trois listes : variables déclarées, lues et écrites, nous pouvons passer à la suite.

Le traitement suivant se produit dans le callback `onEndOfTranslationUnit()` qui ne s'exécute qu'une fois à la fin de l'analyse syntaxique d'une unité de traduction. Durant ce traitement, nous comparons les trois listes créées précédemment afin de déterminer les variables qui sont hors de la portée de la boucle (ces variables sont destinées à devenir des paramètres dans la fonction encapsulante). À noter que nous comparons aussi les variables destinées à devenir des paramètres à la liste des variables étant destinées à la mémoire externe. Si une variable externe est repérée, elle est retirée de la liste des paramètres, car elle est déjà mise en mémoire externe par un autre module.

Une fois la liste définitive créée, nous créons le nouveau fichier accueillant la fonction encapsulant le corps de boucle à séparer en générant la signature adéquate. Nous remplaçons aussi le corps de boucle par un appel à la fonction nouvellement créée dans la fonction originale. Une fois ces transformations effectuées, nous modifions les informations contenues dans le `ModuleInfos` du module segmenté afin de refléter son nouvel état. Nous créons aussi un nouveau `ModuleInfos` représentant le module encapsulant le corps de boucle segmenté.

Une fois sortis de cette action, nous procédons à une passe des actions précédentes sur le module nouvellement créé afin qu'il soit dans un état propice à la suite du traitement.

A-3.5.FindReturnStmtAction

Cette action a pour but de retrouver les positions des clauses `return` dans une fonction. Le patron de correspondance utilisé est très simple :

```
1  clang::ast_matchers::StatementMatcher returnStmtMatcher =
2  returnStmt().bind("returnStmt");
```

Extrait A-3-4 : Patron de correspondance de clause de retour

Aucune composition n'est nécessaire pour ce patron de correspondance. Nous utilisons simplement la fonction de correspondance de nœud `returnStmt()` pour récupérer une référence vers chaque occurence de `return`. Ces occurrences sont enregistrées dans le `ModuleInfos` correspondant. Nous nous servons de ces informations pour reformater la méthode d'action des modules boucles.

A-3.6.AnalyzeAssignmentsAction

L'action d'analyse d'opération d'assignement permet d'analyser toutes les opérations d'assignement dans le corps d'une fonction DAM. Le patron de correspondance permet de retrouver toutes les opérations d'assignement existantes :

```
1  clang::ast_matchers::StatementMatcher AssignmentMatcher =
2  binaryOperator(
3    anyOf(
4      hasOperatorName("="),
5      hasOperatorName("*="),
6      hasOperatorName("/="),
7      hasOperatorName("%%="),
8      hasOperatorName("+="),
9      hasOperatorName("-="),
10     hasOperatorName("<<="),
11     hasOperatorName(">>="),
12     hasOperatorName("&="),
13     hasOperatorName("^="),
14     hasOperatorName("|=")
15   )
16 ).bind("assignmentOperation");
```

Extrait A-3-5 : Patron de correspondance d'opération d'assignement

Une fois une opération d'assignement trouvée, l'action génère un nœud racine de la classe `OperationTree` vue plus haut. Une fois ce premier nœud créé, nous visitions les descendants du nœud de l'AST de Clang en utilisant les fonctions de transtypage dynamique de LLVM (exemple pour passer d'une `Expr` générique à une `CallExpr` : `llvm::dyn_cast<const clang::CallExpr>()`) afin de retrouver le type concret de chaque descendant. Nous testons de cette manière tous les types possibles de chaque nœud descendant et effectuons des actions spécifiques pour chaque type de nœud afin de construire notre propre arbre `OperationTree`.

A-3.7.FindCallsAction

Cette action effectue une passe sur toutes les fonctions DAM afin de retrouver dans celles-ci des appels à d'autres fonctions DAM. Afin d'effectuer ce traitement, nous utilisons un patron de correspondance nous permettant de trouver deux types d'appels distincts : Les appels au cœur d'une assignation et les appels directs (voir l'extrait de code A-3-6). Dans le cas du premier type, l'appel lui-même est récupéré par l'ancre `callExprWith`, mais nous récupérons aussi une ancre sur la fonction appelante (`functionDecl`) ainsi que sur l'opération d'assignement elle-même (`operator`). Dans le second cas, nous récupérons juste une ancre sur l'appel et la fonction

appelante (resp. `callExpr` et `functionDecl`). Les informations récupérées pendant cette action nous permettent par la suite de remplacer ces appels par des protocoles de communication appropriés pour les modules. Nous utilisons la structure s`_interaction_struct` définie dans la classe `ModuleInfos` pour stocker ces informations.

```
clang::ast_matchers::StatementMatcher CallsMatcher =
anyOf(
  binaryOperator(
    hasOperatorName("="),
    hasDescendant(
      callExpr().bind("callexprWith")
    ),
    hasAncestor(
      functionDecl().bind("functionDecl")
    )
  ).bind("operator"),
  callExpr(
    unless(
      hasAncestor(binaryOperator())
    ),
    hasAncestor(
      functionDecl().bind("functionDecl")
    )
  ).bind("callexpr")
);
```

Extrait A-3-6 : Patron de correspondance d'appel de fonction

A-3.8.CreateModulesAction

L'action Création de modules est l'action qui permet d'enclencher l'écriture des fichiers représentant les modules SpaceStudio. Nous utilisons à nouveau une action de compilation pour lancer l'écriture afin d'avoir accès aux données du code source (car nous n'enregistrons que les positions dans la source la plupart du temps). Nous utilisons un simple patron de correspondance de définition de fonction (le même qu'utilisé dans l'extrait A-3-1) afin de récupérer un point d'ancrage sur chacune des fonctions analysées. Pour chaque résultat de notre fonction de retour d'action, nous appelons le constructeur de module spécialisé contenu dans l'instance `ModuleInfos` correspondante. Aucun traitement n'est réellement effectué dans cette action, toutes les actions de construction de module sont de la responsabilité du paquetage constructeur de source.

A-4.Paquetage constructeur de source

Le paquetage constructeur de source a pour responsabilité de créer les fichiers sources et entêtes représentant les modules SpaceStudio tirés des fonctions analysées. Ce paquetage tire parti des informations générées par toutes les actions à la compilation. Il utilise ensuite les fonctionnalités de la classe `Rewriter` (voir section 3.1.4.4) de Clang afin de récupérer les éléments de la source originale nécessaires à la construction de la source modifiée. Ce paquetage contient différents constructeurs selon le comportement que doit exhiber le module en sortie.

Tous les constructeurs de modules héritent d'une classe abstraite appelée `ModuleBuilder`. Les développeurs peuvent alors augmenter les capacités de C2Space en créant de nouveaux constructeurs de source basés sur cette classe. Actuellement, C2Space contient 2 constructeurs distincts : un constructeur standard et un constructeur de module à boucle.

A-4.1.ModuleBuilder

La classe `ModuleBuilder` est la classe abstraite sur laquelle sont basés tous les constructeurs de modules de C2Space. Cette classe définit deux méthodes virtuelles pures qui sont les points d'entrée d'un utilisateur de la classe : `createHFile()` et `createCppFile()`. Un développeur qui souhaiterait ajouter de nouveaux constructeurs devra s'assurer de définir minimalement ces deux méthodes. En plus de ces deux méthodes, une série de méthodes communes aux deux constructeurs déjà présents sont définies dans cette classe abstraite, par exemple les fonctions de génération de communications (`generateReadSend()`, `generateAck()`, `generateCommStarters()`) qui analysent les interactions intermodules afin de générer toutes les méthodes de communication dont aura besoin le module ou encore la fonction `replaceCall()` qui remplace les appels à des fonctions DAM par le protocole de communication intermodule en prenant en compte toutes les informations de l'interaction (variables entrées sorties, variables externes, multiples modules appelants, etc.).

Enfin, deux méthodes statiques sont définies dans `ModuleBuilder` qui permettent de définir et d'accéder au module défini comme étant le module principal. Les règles d'écriture étant légèrement différentes pour le module principal, celles-ci sont définies dans la méthode `printMainThreadC()`.

A-4.2.StandardModuleBuilder

Le constructeur de module Standard permet l'écriture de fichiers modules tels que présentés dans la section 3.2.2.2. Ce constructeur définit les deux fonctions virtuelles pures de la classe ModuleBuilder. Ces deux fonctions agissent comme des guides pour la création des modules. Ainsi, elles ne contiennent que des appels à des méthodes de plus bas niveau. L'extrait suivant illustre cela.

```
1  void StandardModuleBuilder::createHFile(clang::Rewriter& rewriter,
   clang::FunctionDecl* context) {
2    this->rewriter = rewriter;
3    td::string moduleName = context->getNameInfo().getName().getAsString();
4    std::ofstream headerFlux("temp/Module_" + moduleName + ".h",
   std::ios::trunc);
5      if (headerFlux) {
6      //Entête du fichier
7      PrintHeader(headerFlux, moduleName, true);
8      //guarding blocks
9      PrintGuardingBlockH(headerFlux, moduleName);
10     //Includes
11     PrintIncludes(headerFlux, moduleName, true);
12     //Déclaration des éléments récurrents d'une classe module
13     PrintClassDeclH(headerFlux, moduleName);
14     //Ajout de la fonction action qui sera le cœur du thread
15     PrintActionDeclH(headerFlux, context);
16     //Ajout des fonctions nécessaires à la communication intermodules
17     ProcessInteractionsH(headerFlux, moduleName, context);
18     //Fermeture du fichier
19     PrintEndFile(headerFlux, true);
20   }
21 }
```

Extrait A-4-1 : Fonction de création du fichier d'entête d'un module standard

Comme nous pouvons le constater, le constructeur fait appel à différentes méthodes, chacune responsable de la création d'une partie du fichier. Nous avons choisi cette approche afin de compartimenter le code et d'isoler les erreurs dans des méthodes ne faisant pas plus d'une centaine de lignes de code. Cela a permis de faciliter le débogage du constructeur.

Les méthodes de création de fichiers prennent en paramètre un flux de fichier. Ce flux représente le fichier module nouvellement créé. Chacune des méthodes va rajouter du code dans ce flux jusqu'à arriver à un fichier source complet.

A-4.3.LoopModuleBuilder

Le constructeur de module à boucle permet la création de modules suivant le modèle présenté dans la section 3.2.2.4. Le constructeur possède une architecture très semblable au constructeur standard. En effet, les deux fonctions virtuelles pures de la classe parente ont été implémentées exactement de la même manière. Les différences se trouvent dans les fonctions de réécriture de source spécifiques comme par exemple `printThreadC()`.