Vers un partitionnement automatique d'applications en codelets spéculatifs pour les systèmes hétérogènes à mémoires distribuées

Eric Petit

N° d'ordre : 4021 ANNÉE 2009

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Éric Petit

préparée à l'unité de recherche INRIA, UMR6074
Institut National de Recherche en Informatique et Automatique
IFSIC

Vers un partitionnement automatique d'applications en codelets spéculatifs pour les systèmes hétérogènes à mémoires distribuées

Thèse soutenue à Rennes le 3 décembre 2009

devant le jury composé de :

Matthieu Martel / président
Maitre de conférences à l'Université de Perpignan

Nathalie Drach / rapporteur
Professeur à l'Université Pierre et Marie Curie (Paris 6)

William Jalby / rapporteur
Professeur à l'Université de Versailles Saint-Quentin-en-Yveline

Serge de Paoli / examinateur
Chef de projet chez ST-Microelectronics

André Seznec / examinateur
Directeur de recherche à l'INRIA-Rennes

François Bodin / directeur de thèse
Directeur Technique à CAPS-entreprise

" Par tous les pépins de la pomme de Newton, que s'est-il passé ici !?!! "
Pr. Emmett Brow, 1985 bis

Remerciements

Je remercie Matthieu Martel qui m'a fait l'honneur de présider ce jury.

Je remercie Nathalie Drach et William Jalby pour avoir accepté de rapporter cette thèse

Je remercie Serge de Paoli et André Seznec pour avoir accepté d'être membres du jury.

Je remercie François pour avoir dirigé ma thèse. Son recul et sa vision fine sur le sujet de ma thèse m'ont permis de prendre une direction claire dans mes recherches et a grandement contribué à l'aboutissement de mes travaux. Je le remercie tout particulièrement pour l'opportunité de valoriser mes recherches par l'industrialisation du prototype d'Astex.

Je tiens à remercier toute l'équipe CAPS, pour son soutien, mais aussi pour les discussions animées, et parfois enrichissantes, lors des pauses. Une mention spéciale pour l'humour de Pierre qui est sans égal.

J'adresse un remerciement particulier à Isabelle, Erven, André, Pierre et Matthieu qui m'ont apporté leur aide dans la préparation de ma soutenance.

Merci à Guillaume et Julien pour leur amitié, leur patience et leur dévouement.

Je remercie ma famille, mes amis pour leur soutien et leur patience.

Je remercie ma femme, Solène, avec qui j'ai partagé la vie de doctorant pendant toute la durée de nos thèses respectives.

Table des matières

Introduction

De l'informatique enfouie au calcul haute performance, la demande de puissance des systèmes informatiques va toujours croissante. Depuis des années, les architectures n'ont cessé de progresser au niveau de l'intégration, de la consommation, de la puissance de calcul, des mécanismes pour la performance... Tous ces progrès ont permis de suivre jusqu'à aujourd'hui la loi de Moore [68]. Cependant, les limites de ces techniques semblent proches. Nous sommes aujourd'hui à une période charnière dans l'évolution des systèmes informatiques. Le coût en développement, en consommation, en surface de silicium nécessaire aux nouvelles optimisations et les bénéfices de plus en plus faibles ou restreints, semblent avoir dépassé un seuil critique de rentabilité qui freine leur mise en œuvre et limite la rapidité de croissance des systèmes informatiques. C'est dans ce contexte que l'on a pu assister au retour en force du parallélisme et des coprocesseurs spécialisés dans les architectures. Cette technique semble en effet apporter le meilleur compromis entre puissance de calcul élevée et utilisation optimale des ressources disponibles.

Il existe aujourd'hui tout un éventail varié d'architectures multiprocesseurs : parallèles ou non, homogènes ou hétérogènes, à mémoire partagée ou distribuée, génériques ou spécialisées, sur une même puce ou non... toutes les combinaisons étant possibles. Il existe pourtant au moins une constante entre toutes ces architectures : il faut partitionner le code en tâches, éventuellement parallèles avant de les distribuer aux différentes unités de calcul. Ce partitionnement est loin d'être trivial, l'espace des solutions est vaste. La mise au point manuelle par un programmeur d'un tel partitionnement est longue, fastidieuse, et source d'erreur. Il est donc essentiel de développer des outils d'automatisation efficaces pour le partitionnement du code séquentiel.

Produire un compilateur capable de partitionner l'application et d'en garantir l'exécution correcte n'est pas tout, il faut que le résultat soit efficace sur l'architecture cible. Le compilateur doit être optimisant. Pour l'aider dans sa tâche, le processus de partitionnement doit en plus de la construction des partitions apporter un maximum d'information sur le code en vue de son optimisation.

Le temps de développement d'un compilateur complet pour une architecture donnée peut prendre plusieurs années. Le rythme de succession des générations de processeurs allant croissant, les industriels cherchant à optimiser le *time-to-market*, les outils de compilation développés aujourd'hui doivent être reciblables et les applications garantir leur *portabilité* et *adaptabilité*.

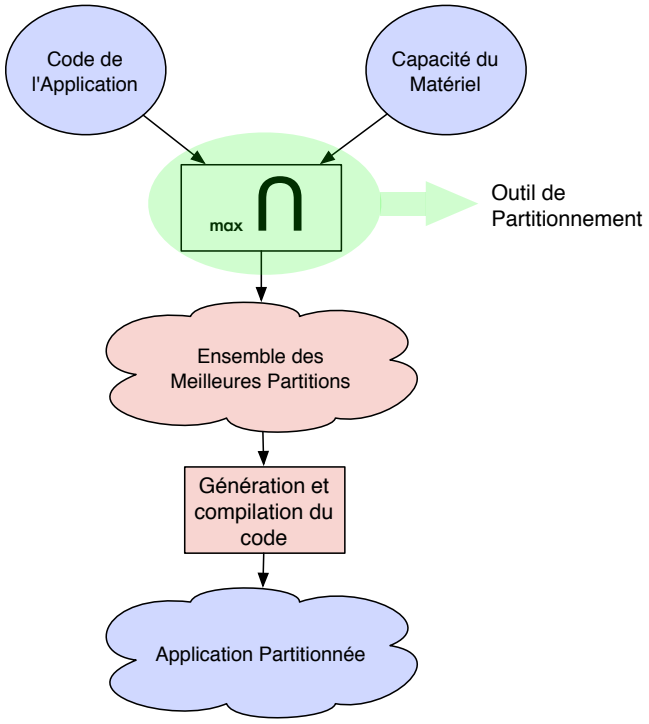

FIG. 1 – Vue générique situant l'outil de partitionnement dans la chaîne de compilation.

Proposition

L'objectif de mon travail de thèse est la mise au point prospective d'une méthode de partitionnement de code et l'étude de ses implications et applications. Ce travail tend à démontrer tant la pertinence que les gains en performances d'une solution basée sur la spéculation.

L'étape de partitionnement du code est un préalable à la compilation d'une application pour un système distribué [79]. Comme il est figuré sur le schéma 1, l'objectif du processus est de maximiser l'intersection entre la couverture en temps d'exécution du code contenu dans les éléments de la partition et le code exécutable sur les unités matérielles en présence, ceci afin de permettre une utilisation quasi optimale des ressources du système. Une fois les partitions calculées, le code correspondant à l'application partitionnée est généré puis compilé.

Un processus de partitionnement du code doit, pour être efficace, répondre à un grand nombre de problèmes. Ces problèmes peuvent être liés au type d'architecture cible, ou bien au code à traiter.

Tout d'abord, il doit être capable de traiter des systèmes à mémoires distribuées, c'est-à-dire où chaque processeur et/ou coprocesseur a son propre espace d'adressage. En effet, eu égard au coût en terme de silicium, de performance et de consommation électrique des mécanismes matériels de cohérence mémoire, les systèmes à mémoires distribuées sont courants dans le domaine de l'embarqué. Enfin, les systèmes de type processeur-coprocesseur, comme le couple CPU-*GPU* devenus incontournables dans la recherche sur le calcul haute performance, forment des systèmes à mémoires distribuées.

Le mécanisme de partitionnement doit aussi pouvoir gérer des systèmes hétérogènes. En effet, le compromis entre performance et coût a conduit à l'émergence de coprocesseurs spécialisés (*SIMD*, *VLIW*, *DSP*, *GPU*...). Après le partitionnement du code, les différentes tâches créées sont distribuées sur les processeurs ou coprocesseurs où leur exécution sera la plus efficace.

De plus, afin d'obtenir les meilleurs résultats possibles pour un large spectre d'applications, il faut impérativement réduire les contraintes sur le code à traiter. Gérer les accès par pointeurs et les problèmes d'*alias* semble être un minimum.

Bien que certains programmes de partitionnement d'applications existent déjà, aucun ne semble répondre efficacement aux problèmes sus-cités. Mon travail se focalise donc sur la mise au point d'une méthode efficace de partitionnement de code principalement destinée aux architectures hétérogènes à mémoires distribuées.

Première Contribution

L'objet principal de ma thèse est la mise au point d'un processus de partitionnement de codes séquentiels en codes multitâches. Mon approche se base sur l'usage de la spéculation. En effet, les informations nécessaires au calcul des partitions et à l'optimisation du code et des communications sont calculées à partir des données de profils d'exécution. Les propriétés déduites de ces profils ne sont donc pas nécessairement vraies pour toutes les exécutions.

L'implémentation du processus automatique de partitionnement d'application, nommé Astex, prend en entrée un code en langage *C*, séquentiel, et retourne en sortie une description quantitative et qualitative des meilleures partitions possibles. Elle consiste en une liste de tâches qui contiennent la portion de code concernée mais aussi des données spéculatives telles que la couverture en temps d'exécution des tâches extraites, un résumé spéculatif des accès mémoire, des données spéculatives sur le chemin dans le code suivi à l'exécution. Astex constitue un frontal regroupant toutes les informations nécessaires à la génération d'un code multitâche à partir du code original. Une implémentation "industrielle" du logiciel est en cours au sein de Caps-Entreprise.

Le processus d'extraction détecte les candidats potentiels à l'extraction. En accord avec la loi d'Amdahl, le partitionnement se focalise sur les parties de calcul intensif du code, c'est-à-dire avec un ratio calcul/longueur du code le plus élevé possible. Par instrumentation et exécution, on calcule le comportement spéculatif du code et de la

mémoire. Une fois la détection des candidats achevée, on sélectionne une partie des portions de code candidates pour les implémenter sous forme de tâches indépendantes appelées "*codelets*". Les critères de sélection portent autant sur le potentiel de gain en performance que sur la compatibilité de la nature du code et son implémentation sur un coprocesseur spécialisé. Si les résultats sont satisfaisants, on a alors obtenu une partition du code. Afin d'optimiser le choix des *codelets*, il est possible, suite à l'évaluation du partitionnement proposé, d'explorer des solutions alternatives afin d'optimiser la partition en sortie. On parle alors de compilation itérative.

La description des tâches ainsi obtenues permet leur implémentation sous forme de *codelets* et leur exécution sur des unités distantes. En spéculant sur les entrées sorties et le code contenu dans la fonction, il est possible de spécialiser les *codelets* et ainsi d'ouvrir la voie à de nouvelles optimisations. Il est alors possible de générer, à la compilation, des versions différentes d'un même *codelet* suivant les données à traiter mais aussi suivant le coprocesseur choisi pour les traiter. Plusieurs versions d'un même *codelet* peuvent coexister. Le choix de la version à exécuter se fait alors dynamiquement en fonction du chemin d'exécution et/ou des ressources disponibles. Les premières applications pratiques d'Astex portent sur l'utilisation d'accélérateur matériel tel que les processeurs graphiques programmés par l'intermédiaire de *HMPP* [42] et CUDA [61, 67]

Seconde Contribution

Lors de la construction d'une partition avec Astex, un grand nombre de propriétés du code est calculé à partir des profils d'exécution. On obtient, en sus de la partition, une modélisation fine du comportement de l'application, tant au niveau du contrôle que des données. L'utilisation de ce modèle, basé sur des informations spéculatives, complète avantageusement l'analyse statique pour l'application d'optimisations sur le code : soit en limitant les indéterminations statiques qui rendaient impossible une optimisation donnée ; soit en ouvrant la voie à de nouvelles optimisations. La seconde contribution de ma thèse est l'étude de la mise au point d'une telle optimisation se basant sur les données spéculatives. Il s'agit de l'optimisation des communications en entrée et sortie de *codelet*.

Lors de l'exécution d'un code partitionné dans un système à mémoires distribuées intervient un grand nombre de communications. C'est le cas par exemple dans le *GPGPU*, i.e. un processeur graphique est utilisé comme coprocesseur. Comme le montre la figure 2, lors de l'exécution distante d'un noyau de calcul sur un coprocesseur, il arrive que tout le gain en temps d'exécution dû à l'accélération du code soit réduit à néant par le coût des communications. C'est le cas dans la deuxième colonne. La dernière colonne de la figure présente un schéma d'exécution du même code, mais avec les communications optimisées. Il s'agit dans ce cas d'un simple préchargement après la dernière définition de la variable ou ensemble de variables à communiquer. Suivant les architectures, le préchargement peut se passer en parallèle de l'exécution du programme principal, c'est

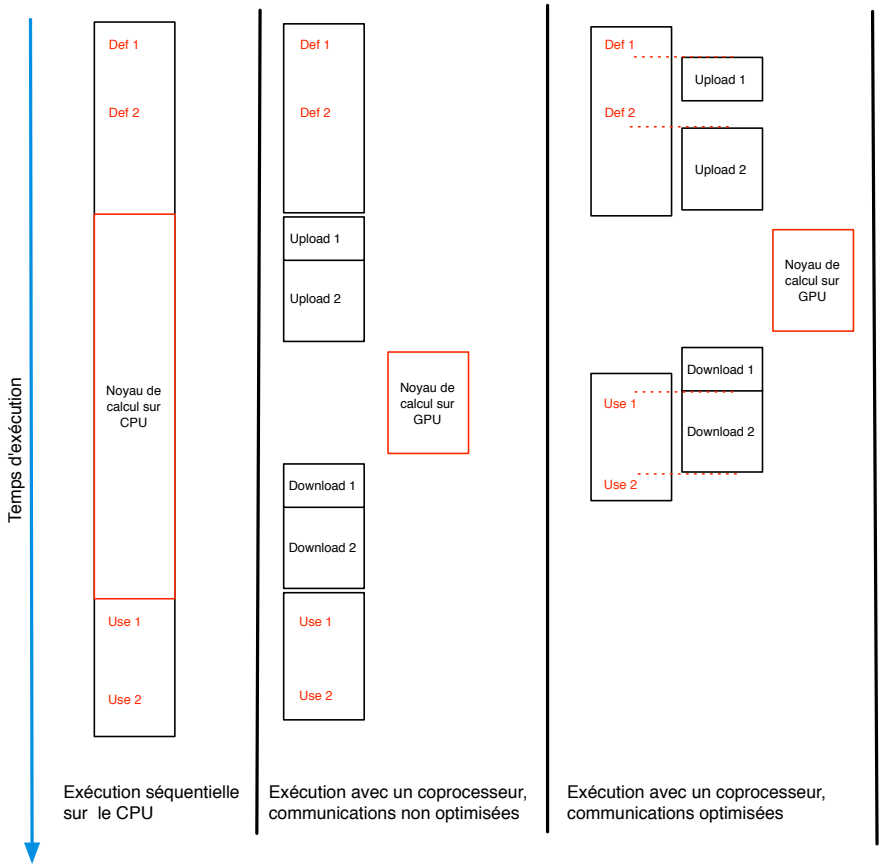

FIG. 2 – Schéma d'exécution d'un programme contenant un noyau de calcul sur un seul processeur, puis avec un coprocesseur sans optimisation des communications et enfin avec un coprocesseur et les communications optimisées.

le cas sur le processeur graphique de dernière génération chez NVIDIA : le GT200 [67].

La méthode exposée dans cette thèse vise à effectuer les communications le plus tôt possible dans le flot d'exécution, tout en limitant le nombre de communications, leur volume, et les possibles redondances. Le rapatriement sur l'unité principale des résultats des calculs effectués par le *codelet* sur le coprocesseur vers le processeur principal constitue un problème symétrique.

L'optimisation de ces communications est un problème, proche de celui de minimisation du placement des barrières de synchronisation, NP-complet. Il est l'une des nombreuses optimisations nécessaires et possibles faisant usage de la spéculation afin d'accélérer les programmes sur des coprocesseurs spécialisés [58, 59].

Présentation du Document

Ce document de thèse se compose de trois chapitres : un état de l'art, l'étude détaillée de la méthode de partitionnement mise au point durant la thèse et enfin la présentation d'une méthode d'optimisation des communications dans le cadre d'une exécution distribuée.

L'état de l'art introduit les notions nécessaires, tant logicielles que matérielles, à la bonne compréhension des tenants et aboutissants du problème de partitionnement de code. Cette partie définit ensuite l'unité du partitionnement et le partitionnement lui-même. Dans la section suivante, je présente un panel représentatif des méthodes déjà existantes de construction de programmes partitionnés et en explique le fonctionnement. Je poursuis par la description des modèles d'exécution possibles d'un programme partitionné, ce qui me permet d'introduire les différentes sources de surcoûts d'une exécution distribuée et/ou parallèle. Ces derniers sont essentiels dans l'analyse des performances des différentes approches. Enfin, cet état de l'art termine sur l'exposé des techniques proches de l'optimisation des communications mises au point durant ma thèse.

La deuxième partie présente la première contribution de cette thèse. Il s'agit de la méthode spéculative de partitionnement d'application ainsi que de son implémentation : Astex. Se basant sur les deux premières sections présentant respectivement le modèle des représentations des partitions et le modèle d'exécution du code partitionné, cette partie décrit dans sa troisième section le processus de construction des partitions. Suit logiquement la présentation de l'implémentation d'Astex ainsi que l'analyse des résultats expérimentaux.

Enfin la troisième et dernière partie présente la seconde contribution de ma thèse. La première section formalise le problème d'optimisation des communications entre un coprocesseur et son/ses coprocesseurs. Vient ensuite la description d'une solution sous forme d'algorithme. Enfin, je développe le cas particulier de l'optimisation des communications dans le cadre de l'utilisation d'un processeur graphique comme coprocesseur. Je décris les premiers résultats obtenus avec mon approche sur ce problème particulier.

Cette thèse conclut sur le bilan des contributions apportées par rapport à la problématique de départ et présente différentes perspectives possibles pour améliorer la

méthode de partitionnement proposée mais aussi les applications possibles des résultats d'ASTEX dans l'optimisation des programmes partitionnés et de leur environnement d'exécution.

Chapitre 1

État de l'art du partitionnement d'application

L'utilisation du parallélisme et/ou d'accélérateurs matériels pour améliorer les performances d'exécution de programme est une solution depuis longtemps prospectée par la recherche et l'industrie [1, 51, 60, 62, 63, 64, 65, 66, 67]. Les domaines d'application sont variés, des systèmes enfouis au calcul haute performance en passant par les machines dites de bureau. Les types d'architectures le sont donc tout autant.

L'utilisation efficace de ces nouvelles ressources mises à disposition dans les architectures passe par le partitionnement et/ou la parallélisation du code (dénommé processus de partitionnement par la suite) [6, 7]. L'exploration de l'espace des solutions de partitionnement est très coûteuse et réduite dans la conception manuelle de programmes parallèles et/ou distribués [7]. Il est critique de développer des procédés automatiques de partitionnement de code. Ce partitionnement ne se limite pas aux boucles ou *nid de boucles* et traite aussi les portions de codes irréguliers (dits "non numériques") [3].

Une fois l'application partitionnée, il est nécessaire de mettre en place des mécanismes de gestion de l'exécution pour en assurer le bon déroulement. Ces mécanismes, parfois complexes, sont une source importante de surcoûts. L'obtention de bonnes performances passe par l'optimisation du déroulement de l'application, en particulier au niveau des communications, surcoût majeur dans les systèmes distribués [7].

La première partie de cette section présente une vue logicielle et matérielle des systèmes parallèles. Après avoir introduit les modèles de parallélisme à grain fin puis à gros grain, je décris de manière générale les différentes caractéristiques des architectures parallèles puis me focalise sur l'utilisation d'accélérateur matériel, objectif central de mon approche du partitionnement de code.

Suite à une partie définissant les éléments principaux d'une partition, en particulier spéculative, je décris les méthodes de construction automatique de partition me basant

sur une sélection représentative des approches présentes dans la littérature.

Une fois le code partitionné construit et généré, il nécessite une gestion particulière de son exécution, en particulier au niveau des communications et de la gestion des erreurs. La gestion de l'exécution est l'objet de la quatrième section de cette partie qui conclut sur l'analyse des surcoûts engendrés à l'exécution.

La cinquième partie présente l'état de l'art relatif au problème d'optimisation des communications. Après une introduction au problème théorique du MinCut, je présente un problème proche et déjà bien traité dans la littérature : le problème du placement des barrières de synchronisation. Enfin je conclus sur les techniques existantes d'optimisation des communications.

Après un bilan des différents résultats quantitatifs et qualitatifs des approches existantes, la conclusion situe notre approche et introduit les problèmes que j'ai contribué à résoudre.

1.1 Parallélisme et architecture

Afin d'obtenir le meilleur compromis entre puissance de calcul élevée, coût minimal et consommation électrique réduite, la conception de systèmes enfouis s'oriente vers les systèmes distribués [6]. Ils possèdent donc plusieurs processeurs, éventuellement hétérogènes, une hiérarchie mémoire propre dont l'objet est de permettre l'exécution de multiple tâches sur des unités différentes, en parallèle ou non.

Le compromis entre difficulté d'intégration, performance et dissipation thermique ayant eu raison des monocœurs complexes et des fortes augmentations de fréquence d'horloge, les processeurs superscalaires "classiques", comme ceux d'AMD et Intel, ont adopté récemment des solutions similaires [51, 66].

Lors de l'exécution d'une application, le parallélisme peut se situer à plusieurs niveaux. Dans le schéma de la figure 1.1 la partie en vert met en valeur le parallélisme aux niveaux des tâches, c'est-à-dire que ce sont des portions complètes de l'application qui sont exécutées en parallèle sur des unités généralement différentes. On parle de parallélisme à gros grain. Les parties notées en bleu pointent sur des zones où le parallélisme est interne à une tâche exécutée sur un seul processeur ou coprocesseur. On parle de parallélisme à grain fin.

Dans un premier temps, cette partie décrit les principaux mécanismes du parallélisme, du parallélisme d'instruction au parallélisme de tâche.

Dans un second temps, je présente les principales caractéristiques architecturales des systèmes parallèles et/ou distribués et leur influence sur la nature et la forme de l'application.

1.1.1 Parallélisme à grain fin *vs* parallélisme à gros grain

Lors de l'exécution d'un programme, le partitionnement en tâches d'un programme peut intervenir à plusieurs niveaux. On parle de *granularité*. Plus la *granularité* d'une

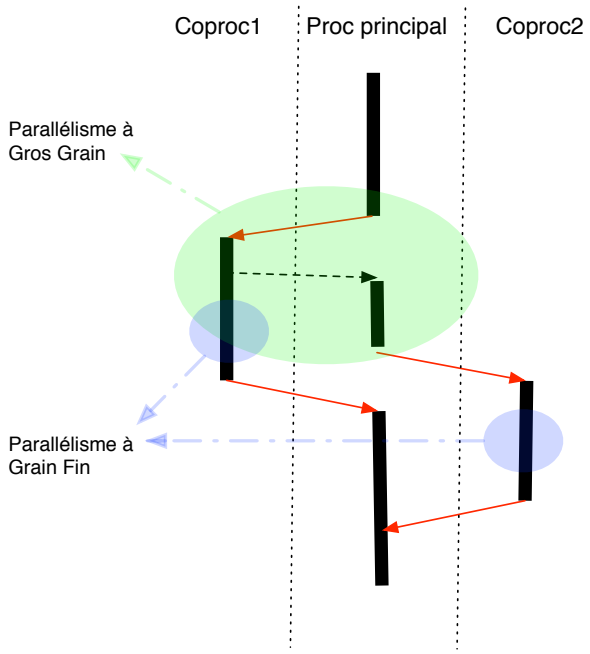

FIG. 1.1 – Schéma d'exécution d'un programme sur un système avec un processeur et deux coprocesseurs. Le parallélisme entre les tâches est dit à gros grain. Le parallélisme au sein d'une tâche est dit à grain fin.

tâche est élevée plus sa taille est grande. Suivant la *granularité* et les architectures, c'est au matériel ou à l'utilisateur d'exhiber ce parallélisme.

Dans cette section, je présente successivement les méthodes et modèles du parallélisme à grain fin puis du parallélisme à gros grain.

1.1.1.1 Le parallélisme à grain fin

Le parallélisme à grain fin est l'exécution en parallèle d'une instruction ou d'un groupe d'instructions en parallèle. Le parallélisme à grain fin apparaît naturellement dans plusieurs types d'applications : le calcul itératif sur des tableaux, le calcul en flux (*stream computing*) comme la compression, l'encryption, le filtrage... On peut distinguer deux cas de figure dans l'utilisation du parallélisme à grain fin :

- L'exécution en parallèle d'instructions différentes et indépendantes d'un même programme : c'est le parallélisme d'instruction.
- L'exécution d'une même instruction (ou même groupe d'instructions) sur plusieurs données simultanément. On parle alors d'exécution vectorielle ou *SIMD*.

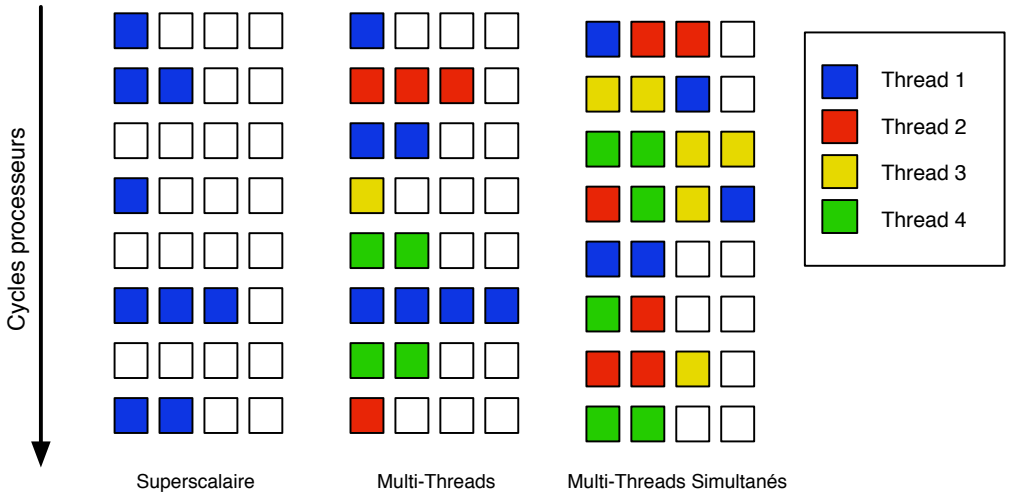

FIG. 1.2 – Trois types de processeurs superscalaires à 4 voies, i. e. on peut sortir 4 instructions par cycle. Le premier exécute un seul codelet à la fois, le second plusieurs mais ne mélange pas les instructions de chaque codelet, le dernier dit smt exécute plusieurs codelets en même temps sans restriction [35, 36, 68].

Le parallélisme d'instruction, dénommé *ILP* (Instruction Level Parallelisme), est le grain le plus fin du partitionnement. L'*ILP* est utilisé en particulier dans les processeurs superscalaires [36, 68]. Quand deux instructions sont sans dépendance logique ou physique (problème d'allocation de ressources) , elles sont exécutées en même temps.

La partie de gauche de la figure 1.2 présente le schéma d'exécution d'instructions sur un processeur superscalaire 4 voies. Les cases pleines représentent une instruction, les cases vides un "slot" libre, c'est-à-dire un emplacement non occupé par une instruction. Quand les instructions arrivent dans le processeur elles sont réparties dans les 4 voies en fonction de leurs interdépendances. Dû aux latences, il est possible qu'aucune instruction ne soit lancée pendant un ou plusieurs cycles, c'est le cas pour l'attente de résultats d'instructions longues ou de chargements en mémoire. Dans cet exemple, au premier cycle n'est exécuté qu'une instruction, deux au deuxième, aucune au troisième... Bien que le nombre d'instructions traitées par cycle, appelé IPC, puisse être supérieur à 1, il est aisé de constater que le nombre d'emplacements libres est élevé, les ressources du processeurs ne sont pas exploitées au maximum.

Lors de l'exécution répétée d'une même instruction ou d'un même groupe d'instructions sur des données différentes, il est possible de les paralléliser. Dans les processeurs superscalaires, on utilise des unités dites vectorielles, l'AltiVec dans le PowerPC d'IBM [46, 47, 49] ou le SSE dans les processeurs Intel [66, 68, 71]. Ces unités exécutent une instruction vectorielle sur tous les éléments d'un vecteur pour sortir un résultat

sous forme de vecteur ou de scalaire.

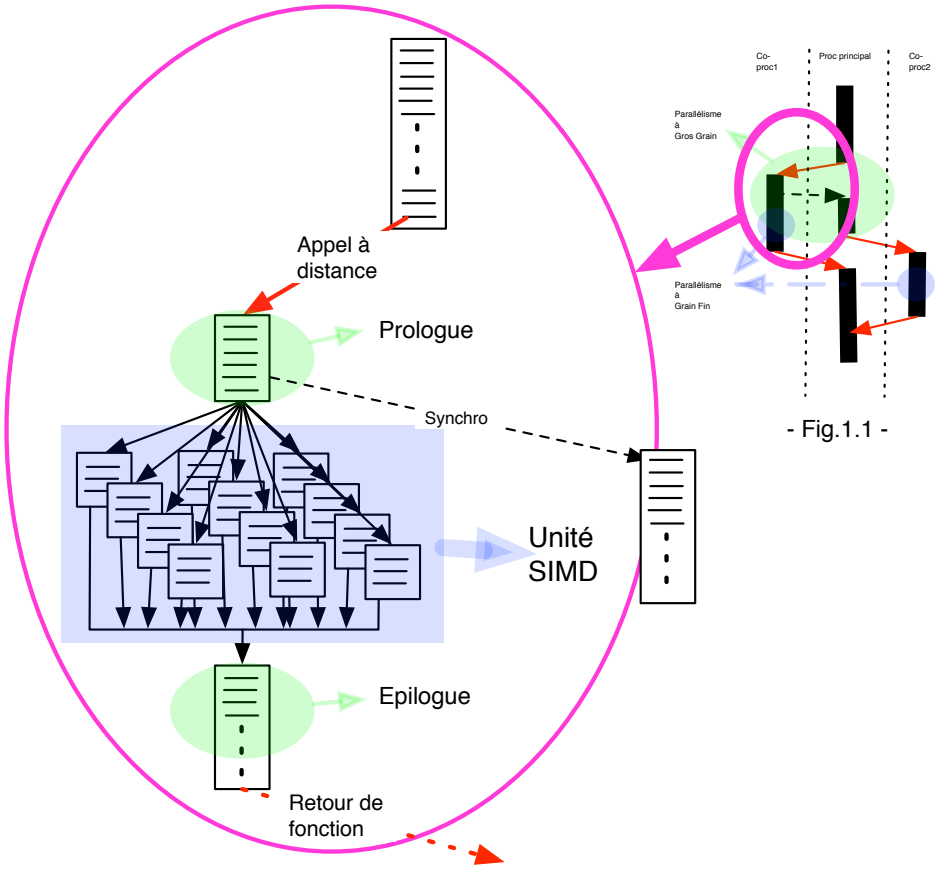

FIG. 1.3 – Vue détaillée possible de la figure 1.1. Le coprocesseur 1 est de type *SIMD*, il exécute le même petit groupe d'instructions en parallèle sur de multiples données.

Certaines architectures sont spécifiquement conçues pour les opérations vectorielles, on parle d'architecture *SIMD*, *Single Instruction Multiple Data*. C'est le cas des SPE du processeur Cell d'IBM [44, 45], des "pixel shaders" des processeurs graphiques [61, 67], du TriMédia de Phillips [68], et d'autres encore. Dans ces architectures, toutes les unités exécutent le même flot d'instructions mais sur des données différentes. La figure 1.3 reprend la figure 1.1, elle présente une vue possible de l'utilisation de parallélisme à grain fin dans le coprocesseur 1. Il s'agit ici d'un schéma *SIMD* qui pourrait correspondre à l'utilisation d'un *GPU* comme coprocesseur. Une fois l'appel lancé au coprocesseur, celui-ci effectue l'exécution :

– d'un prologue : par exemple pour calculer des variables scalaires fonctions des données d'entrée, pour effectuer des tests préalables, éventuellement pour envoyer des signaux de synchronisation disant qu'il va bien effectuer la tâche demandée, etc. ;

– du noyau de calcul : dans la partie avec le fond bleu, on exécute en parallèle le même noyau de calcul sur des ensembles de données différents ;
– de l'épilogue : il se charge d'assurer le retour propre de la fonction et des résultats calculés.

Ce type d'architecture est particulièrement adapté au calcul de type matriciel comme le traitement d'image.

1.1.1.2 Le parallélisme à gros grain

On parle généralement de parallélisme à gros grain quand les tâches exécutées en parallèle dépassent plusieurs centaines d'instructions et contiennent elle-même éventuellement du parallélisme (à grain fin). Chaque tâche effectuée en parallèle pourrait être considérée comme une "mini-application" autonome. Chacune de ces tâches autonomes peut-être effectuée en parallèle comme représenté dans la partie sur fond vert de la figure d'introduction 1.1.

Dans la section précédente, j'ai détaillé le fonctionnement de l'*ILP* dans les processeurs superscalaires. Il apparaît clairement, cf. fig 1.2, que ce système seul sous-exploite les ressources du système et nécessite un fort parallélisme implicite dans les applications chargées, ce qui n'est pas toujours le cas.

Afin de combler les emplacements vides dans le pipeline d'exécution du processeur, on peut mettre en place un procédé de *multithreading* [36]. On lance l'exécution de plusieurs threads sur le même processeur. À chaque cycle, il est possible de changer de flot d'instructions à exécuter en passant d'une thread à une autre. Les cycles de latence peuvent ainsi être recouverts par l'exécution d'instructions d'autres threads.

Ce procédé est illustré dans la partie centrale du schéma 1.2. On constate que les "slots" libres se font plus rares. Cependant, il n'est pas souvent possible de remplir toutes les voies du processeur à chaque cycle, ceci implique la présence de 4 instructions indépendantes dans le même flot d'instructions.

Afin d'optimiser au maximum l'utilisation des ressources du processeur, il est nécessaire de mélanger les instructions de plusieurs threads en les exécutant de manière simultanée. On parle de processeur *SMT*, *Simultaneous MultiThreading* [28, 31, 35]. Cette solution est illustrée dans la partie gauche de la figure 1.2.

Les solutions de *multithreading* sur un processeur monocœur ont tendance à saturer ou interférer avec les mécanismes améliorant les performances des processeurs [31, 35, 68] (TLB, cache, registre,...). Ce partage de ressources entre les threads crée des points de congestion dans le processus et limite l'accélération potentielle que l'on pourrait obtenir. L'adoption des architectures multicœurs répond en partie à ce problème [47]. Dans ces derniers, chaque cœur exécute de manière autonome le groupe de tâches qui lui a été assigné. Le nombre de ressources partagées est bien plus faible (généralement limité au cache de niveau 2), l'accélération des programmes est donc potentiellement meilleure.

L'architecture StepNP de type SoC (System on Chip) et l'outil de développement MultiFlex développés par STMicroelectronics [6] sont un exemple existant de plate-forme d'exécution parallèle de threads. L'architecture est composée de processeurs *RISC* reliés par un réseau configurable, le tout sur une seule puce.

Cette plate-forme permet deux modèles de programmation parallèle : un modèle distribué synchronisé par messages ou un modèle basé sur une mémoire partagée. L'article [6] présente en détail l'architecture, le modèle de programmation et deux exemples d'applications de routage de paquets IPv4. Bien que très performante, l'utilisation des ressources de calcul de cette architecture demande à l'utilisateur de développer des applications spécifiques en exhibant manuellement le parallélisme.

L'utilisation efficace de telles architectures avec un faible coût de programmation nécessite l'utilisation préalable d'un processus automatique de partitionnement. L'intégration de ces architectures serait alors beaucoup plus facile, rapide et donc moins coûteuse.

1.1.2 Description de modèles d'architecture pour les exécutions distribuées

De la même manière qu'il existe plusieurs types de parallélisme et donc de parallélisation, il existe de nombreuses topologies possibles pour les architectures parallèles et les coprocesseurs.

Cependant, quelque soit le type et l'utilisation du processeur ou groupe de processeurs, il existe un certain nombre de caractéristiques architecturales globales communes à tous. Ces dernières nous intéressent dans le cadre de l'étude menée dans cette thèse. Il s'agit de la topologie de la mémoire et de la nature des composants.

1.1.2.1 La topologie de la mémoire

Tout d'abord, il existe deux types de systèmes, ceux à mémoire partagée et ceux à mémoires distribuées comme représentés sur la figure 1.4.

Dans le cas de la mémoire partagée, il n'existe qu'un seul espace d'adressage. Par conséquent, tout le monde accède physiquement aux mêmes données, elles sont partagées. Dans un système supportant le parallélisme, les accès en lecture-écriture deviennent concurrents, des systèmes matériels doivent donc être mis en place pour s'assurer de la cohérence des données [51]. Le schéma topologique d'une architecture à mémoire partagée est illustré dans la partie gauche de la figure 1.4.

La complexité de tels systèmes est quadratique avec le nombre de processeurs et la taille des mémoires [68]. En multipliant le nombre de processeurs pouvant accéder à la mémoire, outre la taille, la complexité et la consommation électrique, la latence va exploser et les performances chuter. L'hypothèse d'une seule mémoire partagée de manière matérielle est trop limitative pour beaucoup de processeurs et systèmes actuels (par exemple SoC et NoC [6]) et encore plus pour les systèmes à venir si la tendance à la multiplication des cœurs se poursuit.

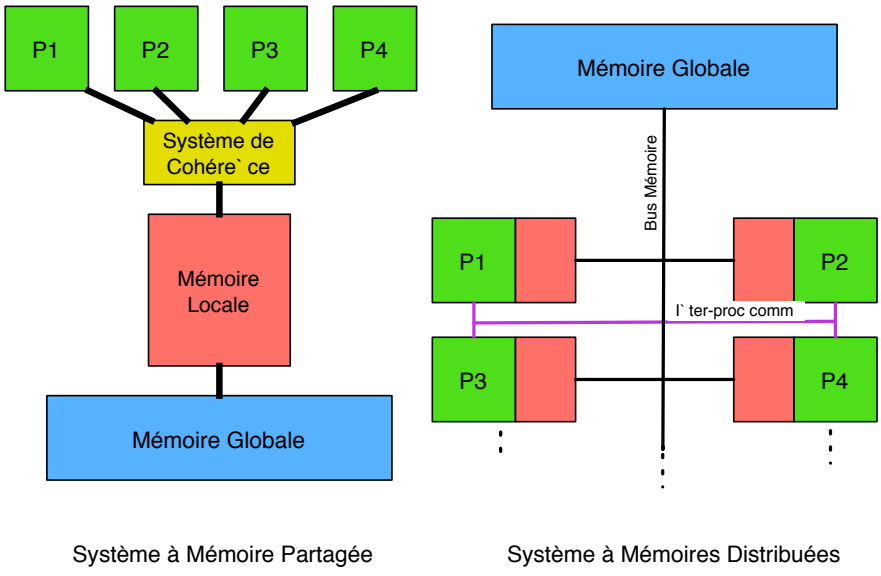

FIG. 1.4 – Schéma de système à mémoire partagée et à mémoires distribuées.

Dans le cas d'un système à mémoires distribuées, chaque processeur possède son propre espace d'adressage et par conséquent, il existe autant de copie de la donnée que d'utilisateur. Au niveau de l'architecture, la solution est "simple" à mettre en oeuvre et passe facilement à l'échelle. Elle facilite la conception modulaire des architectures. Cependant, la multiplication du nombre d'unités augmente la complexité et le besoin en bande passante du réseau d'interconnection. Un schéma générique d'une architecture à mémoire partagée est illustré dans la partie droite de la figure 1.4.

Il n'y a pas de surcoût matériel de gestion de la cohérence mémoire, celle-ci est laissée au système ou à l'utilisateur. Gérer au mieux l'utilisation des ressources mémoire, en particulier éviter les congestions dans le réseau d'interconnexion est une tâche essentielle pour obtenir de bonnes performances sur ce type d'architecture [43, 44, 45]. Cette tâche est très complexe et la recherche de solutions optimales, sauf cas particulier, est NP-complète [19, 21, 24].

Il existe des solutions intermédiaires aux hiérarchies mémoires complexes mélangeant mémoire partagée et distribuée. L'illustration 1.5 est un processeur multicœur Intel à 4 cœurs [67]. Dans ce processeur, chaque cœur a son propre cache de niveaux 1, le cache de niveau 2 est partagé entre les deux cœurs d'un même "die", enfin ils partagent tous la même mémoire centrale. Des topologies mémoires similaires se retrouvent dans beaucoup de processeurs superscalaires actuels.

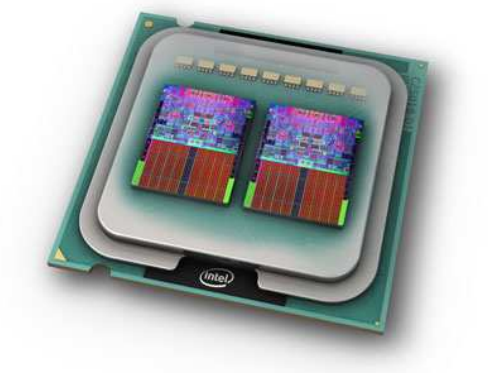

FIG. 1.5 – Le processeur Intel core2 quad core, il est composé de deux "dies" de deux cœurs chacun.

1.1.2.2 Architectures hétérogènes

L'autre caractéristique majeure des systèmes multiprocesseurs dans le cadre de mon étude porte sur la nature des processeurs composant l'architecture matérielle. Ils peuvent être homogènes ou hétérogènes.

Un système est homogène si tous ses composants sont de même nature. C'est le cas par exemple des processeurs multicœurs d'AMD et Intel [67]. La photo 1.5 représente un multicœur homogène. Ce type d'architecture est favorable à l'utilisation de mémoire partagée car tous possèdent le même mode d'adressage. Les tâches sont le plus souvent distribuées aux cœurs par le matériel ou le système d'exploitation en fonction des ressources disponibles. Les gains ne se font qu'en exploitant le parallélisme, tous les cœurs ayant les mêmes performances.

Un système peut aussi être hétérogène, c'est-à-dire composé de processeurs de nature différente. Dans ce cas, la configuration courante est la présence d'un processeur principal de type *RISC* et un ou plusieurs coprocesseurs spécialisés de type vectoriel, *DSP*, *VLIW*, *FPGA*, ... [1, 7, 29] . La photographie 1.6 représente le processeur Cell d'IBM [44, 45]. Ce processeur est composé d'un cœur principal de type *RISC* PowerPC et de 8 SPE qui sont des petits processeurs spécialisés de type *SIMD*, très rapides pour les calculs parallèlisables.

L'utilisation de systèmes hétérogènes permet d'exploiter les différentes architectures là où elles sont les plus efficaces (performance, consommation électrique,....). Le parallélisme n'est plus une nécessité sachant que l'exécution peut être accélérée en exploitant le coprocesseur le plus adapté. De tels systèmes peuvent être contenus sur un seul composant physique, c'est le cas des NoC, SoC [4, 29] ou du Cell [44, 45] par exemple. Il est aussi possible d'intégrer au système des coprocesseurs, sur des composants distincts, comme dans le cas du ClearSpeed CSX [55], du *GPGPU* [54], des accélérateurs

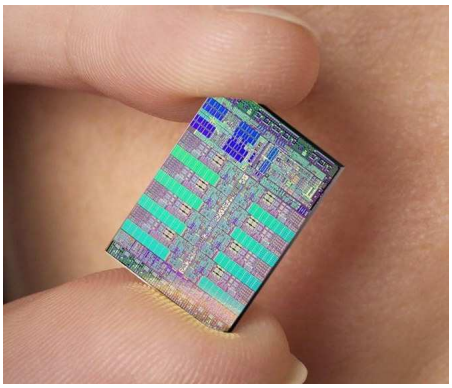

FIG. 1.6 – Le processeur Cell d'IBM.

FPGA [62]... La communication se fait alors à travers des interfaces externes au processeur tel que le PCIe [68, 72]. La bande passante disponible pour les communications avec le processeur principal est alors bien plus réduite et les communications deviennent donc un point prépondérant dans la conception et l'optimisation des programmes [21, 41, 54].

La complexité des systèmes hétérogènes réside dans leur programmation. En effet, le partitionnement du code, la génération des communications et la compilation des éléments du programme sur chaque type d'architecture en présence doivent être fait avant l'exécution. Une compilation à l'exécution de type *JIT* comme dans [5] n'étant pas envisageable en terme de performance.

Toutes ces différentes caractéristiques architecturales peuvent se combiner afin d'adapter au mieux l'architecture à son domaine d'application : le multimédia, l'embarqué, le temps réel, la 3D, le calcul scientifique... La conception de l'architecture dépend alors du ou des programmes à traiter, c'est ce qu'on appelle la conception conjointe matériel-logiciel ou encore "codesign" [7, 55].

1.1.2.3 L'utilisation d'accélérateur matériel

Afin d'accélérer certaines applications spécifiques, il est possible d'effectuer une partie des tâches d'un programme sur un coprocesseur spécialisé. Cette exécution ne se fait pas nécessairement en parallèle du programme principal étant donné que l'accélération du programme est basée sur les performances du coprocesseur et non sur le parallélisme des tâches.

L'utilisation d'un *GPU* en complément du CPU "classique" est un bon exemple d'utilisation de coprocesseur spécialisé pour l'accélération des portions de calcul intensif des applications. Cette solution fait l'objet de recherches approfondies dans un grand nombre de domaines scientifiques "gourmands" en puissance de calcul [55].On parle de *GPGPU*, *General Purpose (computing) on Graphic Computing Unit*.

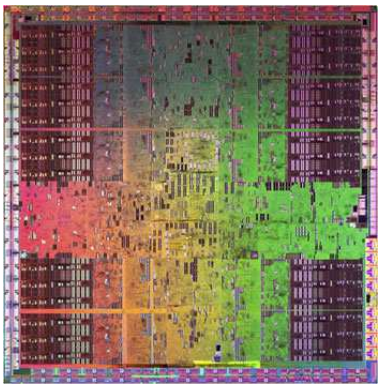

FIG. 1.7 – Photographie du "die" du processeur graphique gt200 de Nvidia. Il possède 1,4 milliard de transistors et exécute jusqu'à 7680 threads simultanément (de manière *SIMD*).

En effet, la performance en pic des architectures de type *GPU* est bien supérieure à celle des processeurs superscalaires classiques. La figure 1.8 montre l'évolution des performances des *GPU* et des CPU d'une génération à l'autre. Non seulement la performance en pic des *GPUs* est supérieure, mais sa croissance est bien plus rapide que celle des CPUs (Intel). Cependant, le gain en performance est principalement la résultante d'architectures de plus en plus massivement parallèles dont les nombreuses unités identiques sont difficilement exploitables. La performance en pic doit être considérée comme une donnée théorique indicative dans le cadre du *GPGPU*.

La plupart des développements d'applications spécialisées sur *GPU* se font manuellement et les résultats, quand les applications s'y prêtent bien, sont souvent excellents. Les accélérations sont d'un facteur 2 à plus de 100 [55].

L'apparition de processeurs graphiques de plus en plus puissants aux capacités de calcul étendues (float 32 et 64 aux normes IEEE, structures de contrôles améliorées,...) a permis la démocratisation de l'usage des *GPUs* comme coprocesseurs. Cela s'est grandement accéléré avec l'apparition de CTM [70] d'ATI et aujourd'hui CUDA [62] de Nvidia. CUDA est une interface de programmation (*API*) proche du *C++* pour le calcul générique sur les *GPUs* Nvidia. Son successeur OpenCL [66] est supposé devenir un standard de programmation pour les *GPUs* et éventuellement autre accélérateurs matériel de type proche du *SIMD* et ce quelque soit leur origine.

Des approches pour la programmation automatique ou assistée sur les *GPUs* sont en cours de développement [61, 65]. Les résultats sont encourageants mais ces approches ne sont pas encore totalement fonctionnelles.

Processeurs et coprocesseurs étant bien souvent de natures différentes, chacun possède son propre espace d'adressage. Dans la majorité des cas les systèmes coprocesseur-processeur forment donc des systèmes à mémoires distribuées (cf 1.1.2). L'utilisation d'un coprocesseur implique des communications en sus de la synchronisation pour assurer les transferts de mémoire entre les différents espaces d'adressage. Les médiums de communication dans les solutions processeurs-coprocesseurs ayant une bande passante

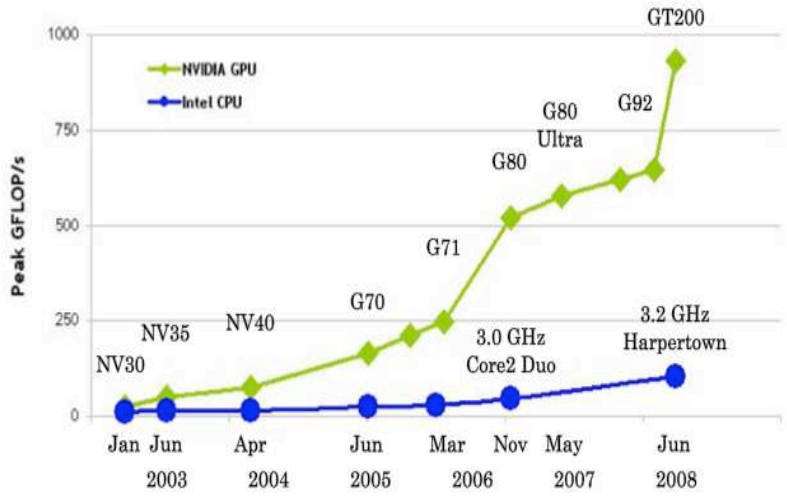

FIG. 1.8 – Courbes d'évolution de la performance en pic des processeurs Intel et des processeurs graphiques NVidia.

limitée, il est nécessaire d'avoir des threads d'une *granularité* suffisante pour masquer les latences dues aux communications [9]. Il est possible sur certaines architectures d'effectuer les chargements au plus tôt en parallèle du programme principal. La mise en place d'une telle optimisation est loin d'être triviale [21, 41, 54], elle fait l'objet de la deuxième contribution de ma thèse.

1.2 Le partitionnement d'application

Afin d'exploiter les mécanismes de parallélisme à grain fin ou à gros grain présents dans les architectures telles que décrites dans la section 1.1.2, il est nécessaire d'adapter les applications séquentielles classiques, les "*legacy codes*" afin d'extraire des parties de code pour les distribuer et/ou les paralléliser sur les différentes unités de calcul disponibles.

L'objectif est de maximiser l'intersection entre le code extrait et les capacités des différentes unités sans avoir à reprogrammer intégralement toute la bibliothèque de programmes spécifiquement pour chaque architecture.

L'architecture matérielle étant généralement fixe, c'est au niveau du logiciel, plus flexible, que l'on peut travailler sur les propriétés du code pour l'adapter à l'architecture. L'une des manières de procéder est de spéculer sur des propriétés du code comme le chemin suivi, l'*aliasing*, l'*alignement des données*, la taille des variables... D'autres approches basées sur des architectures reprogrammables [7, 55], comme les *FPGA*, inversent les rôles et adaptent l'architecture aux applications en ne faisant que

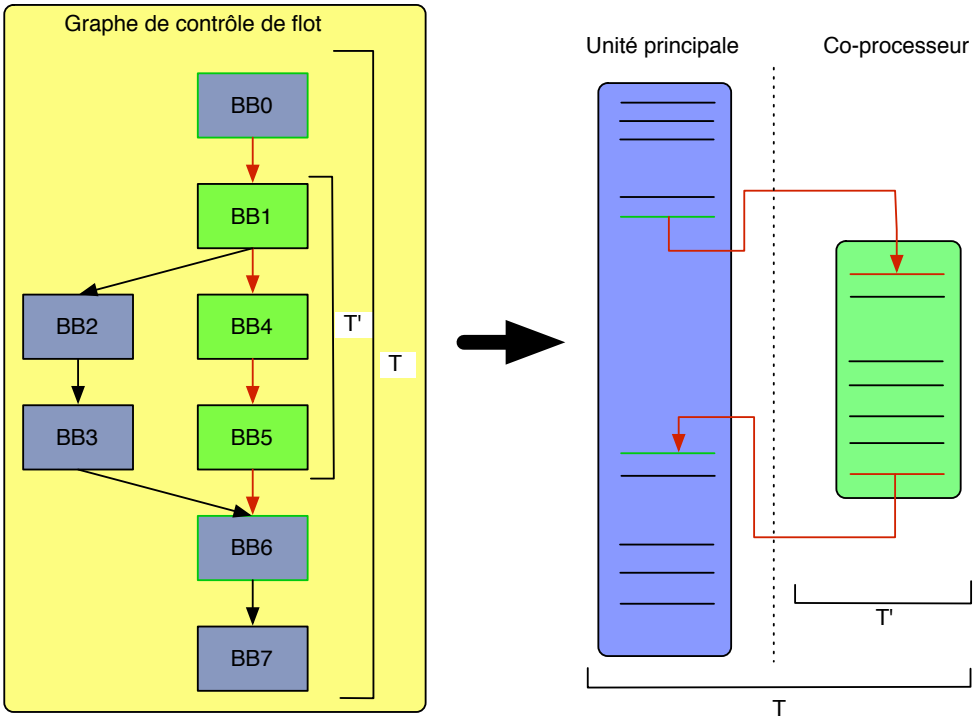

FIG. 1.9 – *CFG* et trace d'exécution associés à une procédure contenant un codelet.

des remaniements mineurs du code.

Lors de la phase de partitionnement, les parties de code à extraire ou paralléliser, sont identifiées. Elles sont alors mises sous forme de *codelets* éventuellement spéculatifs.

Le premier point de cette partie définit la notion de *codelet*. La seconde explique où et comment intervient la spéculation.

1.2.1 Définition d'un codelet

Un programme peut se diviser en *Bloc de Base*, *BB*. Un *Bloc de Base* est un groupe d'instructions consécutives sans instruction de saut sauf éventuellement en dernière position. En reliant chaque *BB* à ses successeurs, on obtient le Graphe de Contrôle de Flot, *CFG* (*Control Flow* Graph). Lors d'une exécution, la séquence des *BBs* décrit une trace T d'exécution, elle correspond à un chemin dans le *CFG*. Une sous-partie d'instructions consécutives de T correspond à une partie du chemin dans le *CFG*, elle constitue une trace T' (Fig. 1.9). On appelle contexte C de T' l'ensemble des mots mémoires et variables d'état utilisés par les instructions de T'. L'ensemble T'+C forme ce que l'on appelle un codelet. Ce codelet peut-être extrait du programme et exécuté de manière indépendante. On parle aussi de *fonction pure*.

Lors d'une exécution parallèle, le contrôle de flot du code séquentiel impose une

relation d'ordre sur les codelets [9] . On dit que deux codelets sont indépendants si ils ne partagent pas de dépendance de données. Si deux codelets sont indépendants leur ordre d'exécution peut être interverti ; une fois cet ordre fixé par le système, ils sont soumis à la relation d'ordre comme tous les autres codelets. La notion d'ordre et l'indépendance ne sont donc pas directement liés. Un codelet T sera un prédécesseur de T' si les résultats de T sont attendus avant ceux de T' ; dans le cas contraire, il s'agit d'un successeur. La relation d'ordre établie interviendra dans certaines approches [4] dans la gestion de l'exécution et/ou le calcul des surcoûts et donc dans l'étape de génération de code.

Afin de réduire les dépendances de données entre les codelets et ainsi réduire les contraintes d'*ordonnancement*, il est possible de spéculer sur le contexte des codelets. C'est l'objet du point 1.2.2.

1.2.2 La spéculation

Il est possible, afin d'élargir les possibilités d'extraction, d'optimisation et de parallélisation du code de spéculer sur les propriétés de ce dernier. Cette section présente une partie des sources et applications possibles de la spéculation dans les applications de partitionnement et parallélisation de code. Le déroulement global de l'exécution distribuée et la gestion de la spéculation font l'objet de la section 1.3.

La spéculation peut être faite de manière arbitraire puis affinée au fil des exécutions. De telles techniques sont utilisées dans des approches dynamiques [4, 5, 7, 8], c'est-à-dire quand l'extraction et/ou la parallélisation se passe à l'exécution.

Dans les approches statiques [1, 2, 3], il peut aussi être fait usage de la spéculation. Deux solutions sont alors possibles :

- Par analyse statique du code : on retrouve des "patrons" de code connus dont on connaît de manière quasi certaine le comportement dynamique. On ajoute alors au code les tests nécessaires qui permettront de s'assurer que l'on a fait le bon choix.
- L'autre solution, celle que nous avons développée, se base sur des données issues de profil d'exécution, donc dynamique, pour inférer les propriétés du code. Là encore, il est possible de générer à la compilation des tests validant les choix effectués. Ces approches statiques ne font donc pas l'hypothèse de la présence nécessaire de mécanismes matériels de gestion de la spéculation.

Il existe deux sources de spéculation possibles pour les codelets : elles peuvent être spéculatives par construction, c'est-à-dire que le partitionnement se base sur des informations spéculatives, ou alors à l'exécution, c'est-à-dire que l'appel au codelet peut être lancé sans garantie sur l'utilité et/ou les éventuelles dépendances. Par ailleurs, cette spéculation peut porter sur les données ou sur le contrôle.

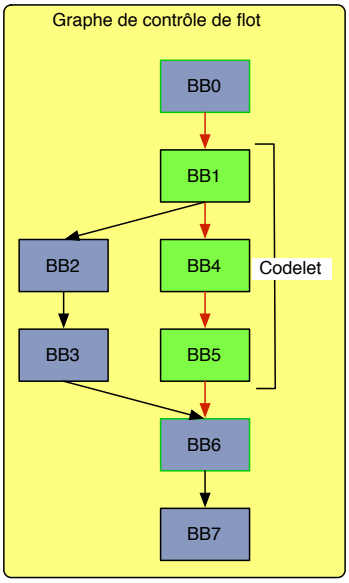

FIG. 1.10 – *CFG* associé à une procédure contenant un codelet. Ce codelet ne contient qu'un seul chemin issu de BB1.

1.2.2.1 Spéculation sur le contrôle

Lors de la détection du codelet, il est possible de ne pas choisir d'extraire tous les chemins possibles d'une section de code. Dans l'exemple de la figure 1.10, {BB2 ;BB3} sont exclus du codelet. Si le codelet comporte des instructions de saut dont la destination n'est pas unique, celle de BB1 dans l'exemple, il faut alors prévoir le retour du codelet en cas de mauvaises prédictions et l'exécution du chemin valide [4] : {BB2 ;BB3} dans l'exemple. Lorsqu'on choisit d'exécuter ce codelet, on spécule sur le contrôle.

L'intérêt de la spéculation sur le contrôle peut intervenir à plusieurs niveaux.

Limiter le nombre de chemins possibles peut diminuer l'espace de travail du code, c'est-à-dire que la mémoire à copier et à gérer dans le cadre d'un système à mémoires distribuées sera plus faible. De plus, la ressource mémoire, tant en terme de quantitée et de débit est souvent limitée dans les coprocesseurs [44, 45].

On diminue par ailleurs la taille du code et le nombre de branchement. Ceci a pour effet d'améliorer le fonctionnement de mécanisme matériel pour la performance comme les caches d'instruction et les préchargements, les prédicteurs... Le comportement du code est plus prévisible.

Enfin, la simplification des structures de contrôle présente plusieurs avantages. Le premier est un gain de performance parfois très grand sur les architectures spécialisées qui ont une gestion peu performante des branchements, la plupart des *GPUs* [54, 67], par exemple. De plus, supprimer les branchements facilite un grand nombre d'optimi-

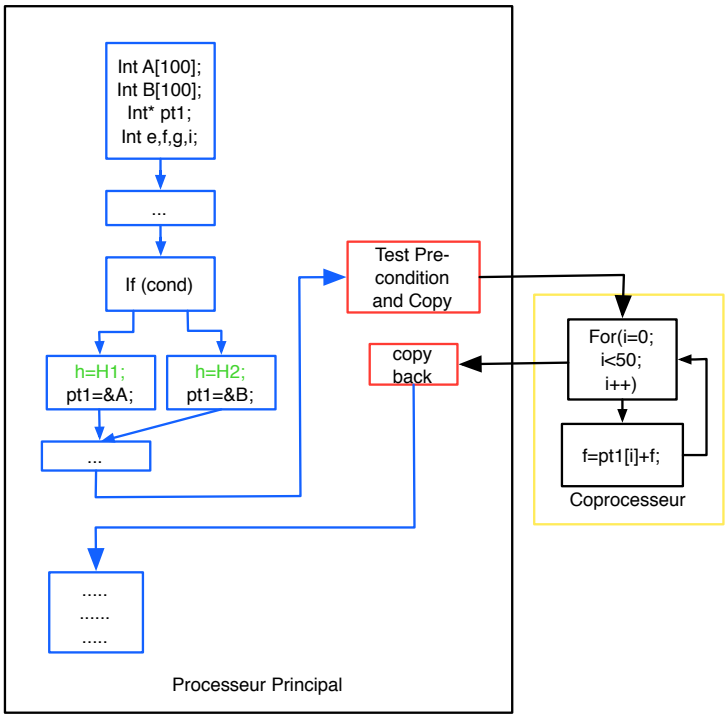

FIG. 1.11 – *CFG* associé à une procédure contenant un codelet exécuté sur un coprocesseur distant (encadré jaune). Suivant le chemin d'exécution suivi, le codelet travaille exclusivement sur la zone A ou exclusivement sur la zone B.

sation du code à la compilation ; en particulier au niveau du "*scheduling*", c'est-à-dire l'*ordonnancement* des instructions, problème récurrent sur les architectures "*in order*" comme le sont la plupart des coprocesseurs *VLIW* et *SIMD* [59].

1.2.2.2 Spéculation sur les communications

Lors d'une exécution sur un système distribué, afin que le coprocesseur ait accès aux données nécessaires pour effectuer la tâche qui lui a été confiée, on copie une partie des données du programme principal dans la mémoire locale du coprocesseur. Afin de réduire les communications et de réduire l'espace mémoire utilisé, il convient de transférer le minimum de données. Il n'est pas toujours possible statiquement de déterminer avec précision quelles zones mémoire et quelles portions de ces zones seront accédées. Il peut alors être fait usage de la spéculation pour minimiser la taille et le nombre des zones mémoire accédées [75]. Cette spéculation est calculée à partir de profils d'exécution résumant les accès effectués en fonction du contexte d'exécution. Une fois le contexte identifié, on effectue une copie uniquement des données que l'on suppose accédées. Il n'existe cependant pas nécessairement de preuve statique que le choix effectué à partir des profils est tout le temps vrai. Lorsqu'on exécute un codelet en minimisant les

communications avec la méthode sus-citée, on spécule sur les données.

L'exemple 1.11 présente le *CFG* d'un programme contenant l'appel à un codelet sur un coprocesseur distant. Ce codelet calcule son résultat à partir des données du tableau pointé par $pt1$. Suivant si $pt1$ pointe sur A ou B, le coprocesseur travaillera uniquement sur A ou sur B. Par conséquent, il est possible de ne copier qu'une des deux zones *alias* de $pt1$ en ajoutant des tests judicieux aux programmes.

D'autres optimisations basées sur des données spéculatives sont possibles pour les communications. C'est l'objet de la section 1.5 de l'état de l'art mais aussi de la deuxième contribution de cette thèse présentée au chapitre 3.

1.2.2.3 Spéculation sur les dépendances de données

Afin d'exploiter au mieux le parallélisme, il est possible, sans vérification préalable ou en se basant sur des données spéculatives, d'assumer l'indépendance d'un codelet avec le reste du code [33]. On peut alors faire intervenir le lancement des codelets au plus tôt et en parallèle, certaines opérations de lecture/écriture deviennent concurrentes. Si l'on peut résoudre de manière statique la cible des lectures et des écritures, les contraintes sur l'ordre d'exécution sont déterminées statiquement [77]. Dans le cas contraire, on ne peut résoudre les dépendances qu'à l'exécution. Si une lecture spéculative, c'est-à-dire potentiellement effectuée avant une écriture dans le code initial et sur la même zone mémoire, n'est pas valide alors il faut restaurer le programme dans un état valide [1].

Lorsque l'on exécute en parallèle plusieurs codelets sans s'être assuré de leur indépendance, on dit que l'exécution a un statut spéculatif. Quand tous les autres codelets exécutés dans le même temps ont rendu leurs résultats et que l'on a constaté qu'il n'y avait effectivement pas de dépendances de données, le codelet quitte son statut spéculatif [1, 2, 4, 5].

1.3 Gestion des codelets à l'exécution

Durant leur exécution, les codelets nécessitent des mécanismes de gestion répondant aux contraintes induites par l'exécution déportée de programme et éventuellement la spéculation sur les données et le contrôle.

En premier lieu, la mémoire distribuée impose la recopie des données et la translation des adresses afin de correspondre au nouvel espace d'adressage du code. C'est la première partie des communications. La deuxième partie des communications vient de la synchronisation des codelets entre eux lors d'une exécution parallèle afin de répondre aux contraintes de dépendances de données. La gestion des communications et des synchronisations constitue le premier point de cette section.

Ensuite vient le problème de détection des erreurs d'exécution. Elles peuvent être dues à des erreurs de spéculation sur les données, le contrôle, mais aussi liées, par exemple, à des manques de ressources à allouer au codelet. Enfin, une fois détectées, ces erreurs doivent être traitées. Plusieurs méthodes et leurs variantes sont présentées dans la littérature. La détection et la gestion des erreurs sont présentées dans le point 1.3.2 et 1.3.3.

Les surcoûts induits par l'exécution déportée des codelets, en particulier liés à la gestion des codelets, dépendent de l'approche choisie. Les sources principales de surcoût sont détaillées dans la partie 1.3.4.

1.3.1 Communication et synchronisation

Lors de l'exécution parallèle de différentes sections d'un programme, les accès aux données peuvent devenir concurrents. Les dépendances de données ainsi générées imposent un ordre d'exécution pour les codelets. Il est donc nécessaire de se munir à l'exécution de mécanisme de synchronisation pour garantir la bonne exécution des codelets.

Dans le cas des systèmes à mémoires distribuées, chaque coprocesseur a sa propre copie à jour des données nécessaires à ses calculs. Cette recopie des données impose des communications volumineuses et complexes aux systèmes pendant l'exécution.

Dans un premier temps, cette section détaille le problème des synchronisations, les mécanismes matériels et logiciels pour les gérer mais aussi ses implications, en particulier sur l'*ordonnancement* des codelets. Dans un second temps, j'aborde le problème des communications dans les systèmes à mémoires distribuées, leur génération et leur mise en oeuvre.

L'optimisation des communications et synchronisations est abordée dans la partie 1.5.

1.3.1.1 Dépendances et synchronisations

Le schéma 1.12 présente l'exécution distribuée et parallèle d'un programme. Les arcs rouges mettent en évidence les dépendances de données qu'il peut exister entre les codelets. Elles sont de deux sortes [74] :

- Les dépendances de type lecture après écriture, dénommées *RAW* pour "Read After Write". Lorsque dans l'ordre d'exécution séquentielle du programme, une lecture intervient après une écriture, il faut s'assurer lors d'une exécution parallèle que la donnée utilisée pour le calcul soit bien à jour, c'est-à-dire que l'écriture soit bien achevée avant la lecture. Dans la figure 1.12, si une lecture intervient au point 4 avant l'écriture sur la même donnée au point 1 alors il y a violation d'une dépendance de type RAW. Cette dépendance impose donc l'exécution séquentielle dans l'ordre des instructions au point 1 et 4.
- Les dépendances de type écriture après lecture, dénommées *WAR* pour "Write After Read" ou encore anti-dépendance. Lors d'une exécution parallèle sur un système à mémoire partagée, il faut s'assurer que la lecture intervient bien avant l'écriture,ie. l'écrasement de la donnée. Dans la figure 1.12, si une lecture intervient au point 4 après l'écriture sur la même donnée au point 6, alors la donnée lue ne sera pas la bonne. Il faut donc que l'instruction au point 6 soit exécutée après l'instruction 4.

Ces dépendances imposent un ordre naturel pour l'exécution des codelets. Elles sont donc des contraintes fortes pour l'*ordonnancement* des tâches sur les différents

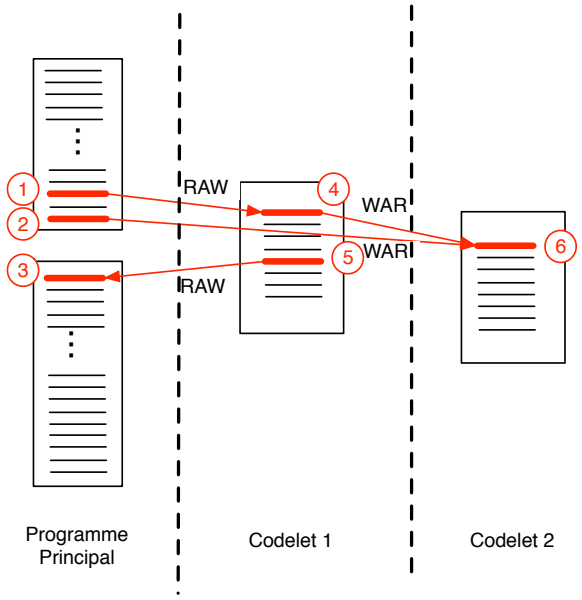

FIG. 1.12 – Graphe d'exécution d'un programme contenant deux codelets. En rouge figure de multiples dépendances de type lecture après écriture (*RAW*) ou anti-dépendances, *i.e.* écriture après lecture (*WAR*). Ces dépendances imposent un ordre dans l'exécution des différentes parties du programme.

processeurs d'un système distribué. Afin de pouvoir exécuter en parallèle des codelets inter-dépendants, il faut mettre en place des mécanismes de synchronisation. Il peut s'agir de communication unitaire ou bien encore de mécanismes telles que les barrières de synchronisation [18, 19]. Afin de permettre la synchronisation, certains codelets sont obligés d'interrompre leur exécution en attendant les résultats d'un autre. Il s'agit d'un surcoût d'exécution, il figure dans la partie 1.3.4 qui répertorie les sources de surcoût possible dans l'exécution parallèle de programme.

Certaines approches, comme [1, 2, 3], spéculent sur la présence ou non de dépendances. Le code est alors exécuté en parallèle et des mécanismes vérifient si d'éventuelles violations de dépendance ont eu lieu. Les parties 1.3.2 et 1.3.3 traitent de la détection et correction des erreurs, en particulier sur la violation des dépendances de données.

1.3.1.2 Gestion des communications

Le schéma 1.13 représente une vue schématique d'un système processeur coprocesseur. Il peut, par exemple, correspondre au système formé par un CPU et un *GPU* comme coprocesseur. Lors de l'exécution, avant que le processeur ne fasse appel au coprocesseur, il est nécessaire de transférer les données utiles aux calculs dans la mémoire locale du coprocesseur. À la fin de l'exécution, il faut les rapatrier sur l'unité principale. Certains systèmes, comme le CELL d'IBM sont hétérogènes et ont pourtant un espace d'adressage global, *i.e.* les données sont virtuellement accessibles par tous. Dans ce sys-

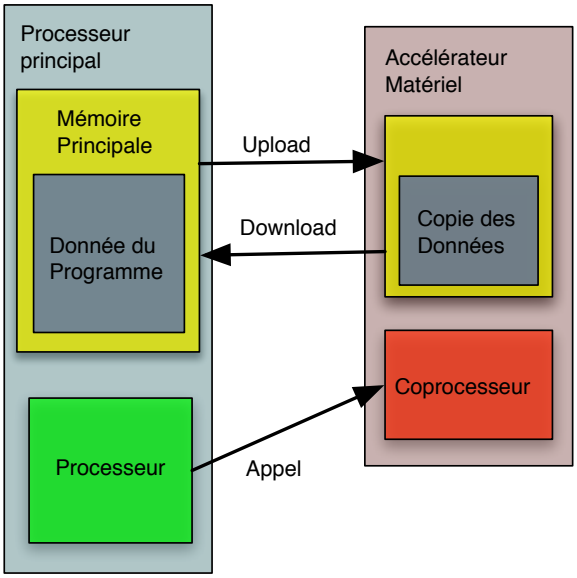

FIG. 1.13 – Schéma d'un système processeur coprocesseur avec modélisation des communications.

tème, les communications sont de grain plus fin mais quasi permanentes. La localité des données et la position physique du coprocesseur auquel un codelet est alloué a une influence sur les performances [76].

Lors d'une exécution distribuée interviennent deux problèmes principaux, sources de surcoûts importants, dans la gestion des communications :

- Le choix des données à copier.
- Quand et comment copier les données.

Lors de l'exécution sur l'unité distante toutes les données qui peuvent être accédées doivent être copiées. Du fait de l'*aliasing*, des tailles de zones mémoire inconnues et des bornes de boucle non déterminées statiquement, cet ensemble est largement surévalué. Pour limiter les coûts et ne pas multiplier les copies inutiles on peut en sus d'une analyse statique fine ajouter une dimension spéculative au choix de copier ou non une zone mémoire.Il s'agit d'une première voie d'optimisation pour les communications.

Il est possible en fonction des capacités du système et de l'architecture hôte d'utiliser des techniques de préchargement afin de masquer les latences due aux communications. Ces techniques pour être efficaces doivent dépasser les problèmes liés aux *alias*. En effet ces derniers sont source de fausses dépendances de type lecture après écriture dans le code à recouvrir entraînant la redondance obligatoire ou le retard des communications concernées par ces faux accès. L'usage de la spéculation pour placer les communications au plus tôt est traité dans le chapitre 3. Masquer les latences en effectuant un maximum de communications en parallèle de l'exécution est la deuxième voie d'optimisation dans la gestion des communications.

1.3.2 Détection des erreurs

Il y a trois types d'erreurs répertoriés dans la littérature [3], elles sont liées aux trois types de spéculation possibles cf.1.13. Ils n'apparaissent pas nécessairement dans toutes les implantations.

- **Erreur de dépendance de données** : En cas d'exécution parallèle, il est possible de voir apparaître des erreurs de type *RAW* (Read After Write). Les anti-dépendances (*WAR* : Write After Read) ne sont pas des cas d'erreurs dans les systèmes à mémoire partagée [3].

- **Erreur d'accès aux données** : Si un codelet accède à une zone de mémoire (par pointeur) qui lui est non allouée, alors il y a violation. Si un codelet provoque un overflow de la mémoire qui lui est allouée, alors il y a erreur.

- **Erreur de contrôle** : un codelet est un ensemble de chemins du *CFG*, par conséquent si l'exécution quitte cet ensemble de chemins alors il y a une erreur de spéculation sur le contrôle. De même, si par anticipation et de manière spéculative on a commencé l'exécution d'un codelet alors qu'elle n'aurait pas dû l'être, il y a erreur de spéculation.

Afin de pouvoir garantir une exécution conforme à celle du programme séquentiel original, il est nécessaire de pourvoir le modèle d'exécution des codelets de mécanismes de détection des erreurs ainsi que du moyen de les corriger. Les mécanismes de détection d'erreurs ont deux origines : les mécanismes matériels et les mécanismes logiciels.

Dans le cas du matériel les accès mémoire sont monitorés en parallèle de l'exécution afin de vérifier leur validité. Deux méthodes sont possibles. Soit on arrête immédiatement le codelet quand une erreur est détectée et on envoie un code de correction de l'erreur. Soit on attend la fin du codelet, on vérifie alors l'intégrité de la mémoire ; si tout va bien, il quitte son statut spéculatif, sinon on le réexécute [1, 2, 3]. On peut aussi ajouter à la mémoire des bits de validité pour réaliser ce contrôle d'accès [1]. Concernant la détection d'erreur de spéculation sur le contrôle, l'une des solutions expérimentées est celle des instructions gardées [4]. Si la garde est vraie on continue le codelet, sinon on reprend la trace originale.

Dans le cas du logiciel, on instrumente le code de manière à détecter les erreurs de contrôle. Lorsqu'une telle erreur est détectée, le codelet quitte le cours de son exécution et on applique un code correcteur. Pour la gestion logicielle des dépassements de zone mémoire et le contrôle des violations de dépendance mémoire, on instrumente le code et on ajoute du code en entrée et sortie du codelet pour contrôler la validité de l'utilisation de la mémoire [2]. Au vu des articles, il n'y a apparemment pas de solution entièrement logicielle. La motivation semble venir d'un surcoût du *monitoring* logiciel [5].

D'autres approches, qualifiées d'hybrides, utilisent les résultats d'analyses statiques et dynamiques. Celle adoptée par Rauchwerger dans [77] par exemple génère de manière statique les tests à vérifier pour la parallélisation automatique à l'exécution des boucles.

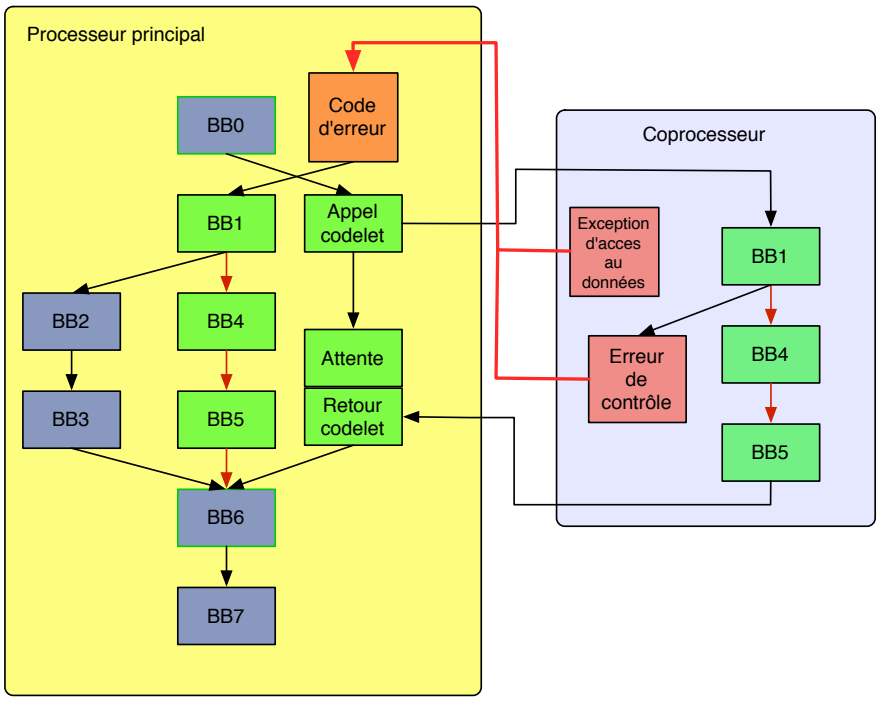

FIG. 1.14 – *CFG* d'un programme contenant un codelet et un dispositif de détection et correction d'erreurs.

Ce procédé présente de fortes similitudes avec la méthode de génération des gardes dans l'approche traitée au cours de ma thèse et détaillée dans le chapitre 2.

1.3.3 Correction des erreurs

Lorsqu'une erreur est détectée par l'environnement d'exécution, il est nécessaire de rétablir le statut valide de l'exécution, *i.e.* conforme à l'exécution séquentielle du programme. Suivant les erreurs et suivant les approches, plusieurs solutions sont possibles.

Une première solution consiste à tuer le codelet ou les codelets en cours et reprendre l'exécution normale du programme. Elle semble plus appropriée aux méthodes d'extraction statique [2, 3]. La figure 1.14 montre la mise en place d'une telle politique de gestion d'erreur. Dans cet exemple, les erreurs d'accès aux données et les erreurs de contrôle renvoient immédiatement un code d'erreur. Chacun de ces codes renvoie sur l'exécution séquentielle du code.

Une deuxième solution consiste en la réexécution du code. Le codelet fautif est arrêté, ses résultats ignorés et les variables rétablies (*roll back*) suivant la politique de gestion de la mémoire (partagée ou non). Le codelet est alors relancé [1, 4]. Dans la figure 1.15, il y a une dépendance de type *RAW* qui n'a pas été respectée entre le codelet 3 et le codelet 1 ; le codelet 3 et ses successeurs ne sont donc plus valides. On

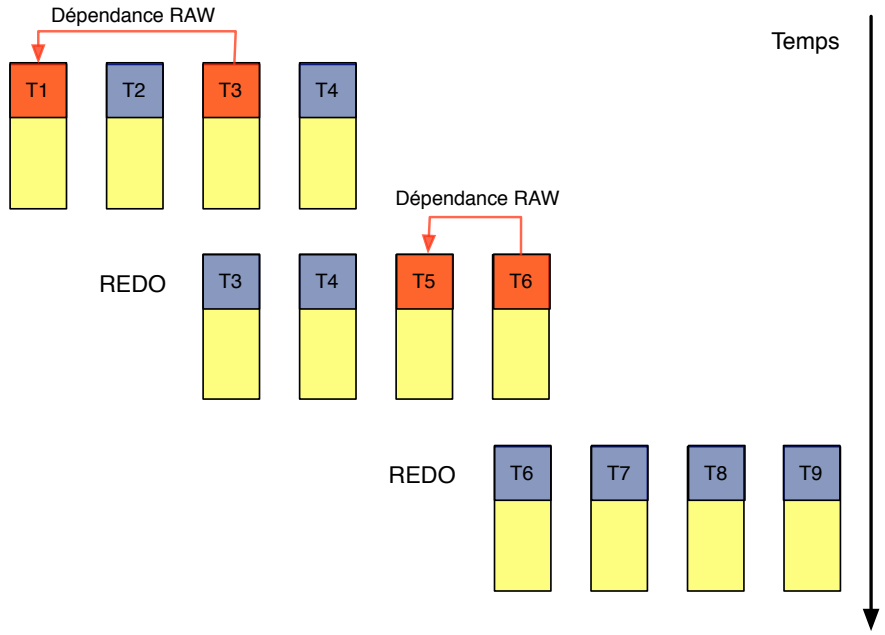

FIG. 1.15 – Schéma d'une politique de *roll back and redo* sur 4 processeurs.

reprendra donc l'exécution de 3 et 4 a la fin de l'exécution de 1 et 2.

Il existe bien sûr différentes variantes de ces solutions ou bien encore des solutions intermédiaires. Pour TEST [5] par exemple, on emploiera un système de *roll back* comme il est décrit figure 1.15 pour les premières exécutions. Une fois ces premières exécutions passées, on ordonnance les codelets de manière à ce que toutes les dépendances qui ne varient pas d'une exécution à l'autre n'aient plus lieu. Pour les dépendances non détectables, on conserve le premier système. Les premières exécutions jouent donc le rôle du *profiling* dans les approches statiques qui font usage de la spéculation.

1.3.4 Analyses des surcoûts d'exécution

Cinq sources majeures de surcoût dans le cadre d'exécution de codelets spéculatifs ont été identifiées [9] : la mise en place et le retour du codelet, les erreurs de prédiction, les overflows de la mémoire, la communication entre codelets, la répartition des charges sur les processeurs. À ces cinq surcoûts vient s'ajouter le surcoût lié à l'implantation de la spéculation dans les codelets.

– **Le surcoût dû à la mise en place et le retour des codelets** dépend en partie de l'architecture. Pour lancer le codelet, la première partie de ce coût est fixe, il s'agit du prologue d'invocation du codelet. Avant d'invoquer le codelet, si la mémoire n'est pas partagée ou si l'unité traitant le codelet a sa propre mémoire,

il faut migrer le contexte vers l'unité distante ; le coût est alors proportionnel à la taille des données en entrée. En sortie du codelet, toutes les données modifiées sont rapatriées dans la mémoire de l'appelant, l'instruction de fin de codelet est exécutée. On a donc ici encore un coût fixe et un coût proportionnel à la taille des données en sortie. Quand architectures et systèmes le permettent, la partie variable du coût peut être optimisée grâce au préchargement des données en parallèle de l'exécution.

– **Le surcoût dû aux erreurs de prédiction**, dépend de la politique de gestion des erreurs (cf. 1.3.3). De manière générale, le surcoût se compose du temps perdu pour la création, l'exécution et la destruction du codelet devenu inutile ainsi que du coût de restauration de la mémoire quand cela est nécessaire. La probabilité que ce surcoût intervienne est la probabilité qu'une erreur de prédiction au moins se produise lors de l'exécution.

– **Le surcoût dû à un dépassement de la mémoire allouée** correspond au dépassement de zone mémoire par le codelet. Lors de l'exécution d'un codelet, une certaine quantité de mémoire lui est allouée et ce codelet produit une quantité pas toujours prévisible de données. Si la mémoire n'est pas suffisante, alors il y a dépassement et l'exécution ne peut plus continuer. Il faut alors rétablir un état valide, en réallouant de la mémoire, ou en exécutant à nouveau le code sous forme de codelets ou dans le programme séquentiel. Le choix est fait en fonction des possibilités de l'architecture. La probabilité de ce surcoût est difficile à estimer car les seules données pertinentes pour la calculer sont celles du *profiling*.

– **Le surcoût dû aux synchronisations entre codelets** vient du temps passé par un codelet pour attendre le résultat d'un autre codelet. À cela vient s'ajouter le temps de migration des données produites par le codelet attendu vers le codelet qui attend. Ce surcoût n'intervient que dans les architectures permettant ce genre de communications entre codelets (ex : mémoire partagée).

– **Le surcoût dû à la répartition des charges** [9] ("load imbalance") correspond au temps durant lequel les processeurs exécutant les codelets restent inactifs après la fin de leur exécution et ce jusqu'à ce qu'ils puissent rendre leurs résultats à l'appelante, c'est-à-dire après ses prédécesseurs. Ce surcoût dépend essentiellement de la différence de temps d'exécution entre les codelets. L'estimation de ce surcoût ne peut ce faire que sur une sélection de codelets (cf. 1.4.2.1).

– **Le surcoût de profilage** [5] est lié à l'instrumentation du code nécessaire dans de nombreuses implantations de détection dynamique des codelets, la prédiction des données, les tests de spéculations,... L'évaluation de ce surcoût est propre à chaque approche.

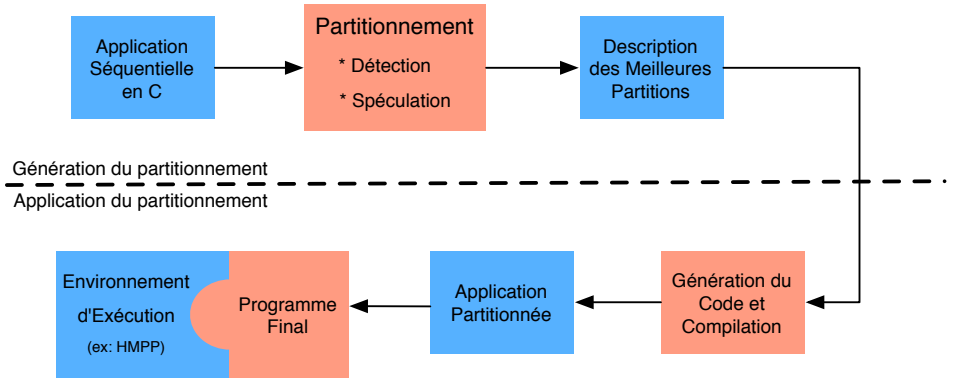

FIG. 1.16 – Vue schématique de l'implantation d'un pipeline de partitionnement statique. La phase de partitionnement est effectuée avant la compilation et l'exécution du code.

Des six surcoûts cités dans cette partie, les deux plus significatifs semblent être la mise en place et le retour des codelets (en particulier la copie des données), et la répartition de charge. Les évaluer avec précision est l'un des points critiques pour estimer le temps d'exécution des codelets et ainsi pouvoir chiffrer l'efficacité d'un partitionnement. Une bonne évaluation de ce surcoût est essentielle pour la sélection des codelets (cf. 1.4.2.1).

1.4 Construction automatique de programmes partitionnés

Le partitionnement manuel d'applications étant trop coûteux et restrictif dans l'exploration de l'espace des partitions possibles [7], il est critique de développer des méthodes automatiques de partitionnement.

Les deux grandes familles de méthodes de partitionnement sont les méthodes statiques [1, 2, 3] et les méthodes dynamiques [4, 5, 7, 8].

La figure 1.16 est une vue générique du processus de partitionnement statique. Bien que chaque approche ait sa subtilité, toutes les approches statiques commencent par la détection, parfois triviale [7], des portions de code à traiter. Tous les éléments relatifs à la génération de la partition, la spéculation et les tests sont alors calculés. À partir de cet ensemble, code plus éléments caractéristiques, on peut effectuer la phase de génération et compilation de l'application partitionnée. Il est alors possible, en s'appuyant sur un environnement d'exécution matériel [1, 2, 3] et/ou logiciel [42, 5] d'exécuter l'application partitionnée.

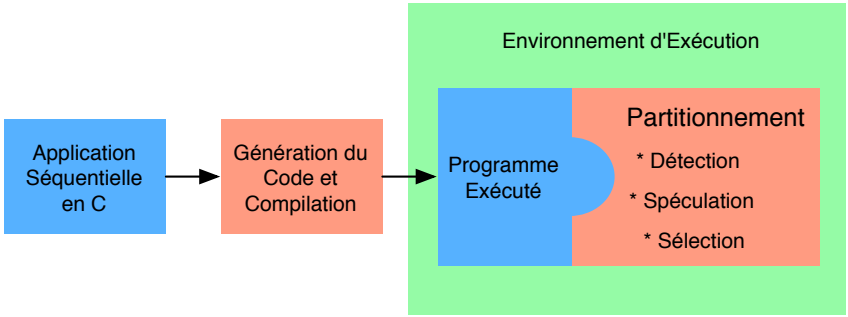

FIG. 1.17 – Vue schématique de l'implantation d'un pipeline de partitionnement dynamique. La détection et la génération du partitionnement se fait à l'exécution et le plus souvent avec l'assistance de mécanismes matériels.

La figure 1.17 présente le schéma d'un processus de partitionnement dynamique. Avant l'exécution, certaines approches [77] permettent aux programmeurs de signaler les points d'intérêts au mécanisme de partitionnement. Le programme est compilé puis exécuté. L'environnement d'exécution, principalement matériel pour des raisons de performance [5], se charge de détecter, délimiter, extraire et parfois recompiler pour le coprocesseur qui est ciblé, les portions de codes à implémenter sous forme de codelets. Le code de l'application est donc modifié à la volée durant l'exécution.

Cette partie présente dans ces deux sections les principales étapes du partitionnement, à savoir respectivement : la détection du code pouvant constituer un codelet et la génération de l'application partitionnée, en particulier la sélection des codelets à implémenter mais aussi les problèmes liés à leur implantation.

1.4.1 La détection des portions de code candidates

Le partitionnement d'un code en codelets commence par la détection des portions de code potentiellement extractibles. Cette étape de détection peut se faire par deux approches, l'une statique, l'autre dynamique.

L'approche la plus communément exploitée pour la détection de codelets est celle de la recherche des portions de code les plus usitées lors de l'exécution [1, 3, 4, 5]. Ces portions de code sont des "*hotspots*". D'autres approches comme PICO-NPA [7], considèrent comme candidates toutes les zones de code constituées par les nids de boucles. Le but est, en accord avec la loi d'Amdhal [11, 68], de cibler les portions consommatrice de temps et de ressource dans les programmes afin de les optimiser.

En complément de cette détection sur les "*hotspots*" se trouve parfois une analyse des "*hotpaths*" [4], c'est-à-dire les chemins les plus usités dans le code lors de l'exécution. Cette analyse permet de faire des hypothèses sur les chemins les plus employés et par conséquent de spéculer sur les branchements.

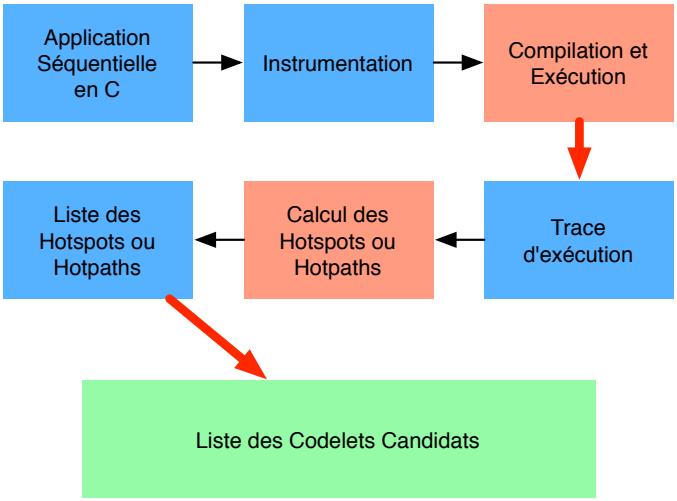

FIG. 1.18 – Schéma du pipeline de calcul des *hotpaths* d'un programme par instrumentation et exécution.

Une fois l'ensemble des codelets formé, on sélectionne le sous-ensemble qui a potentiellement les meilleures performances afin de générer l'application partitionnée.

Cette section présente successivement la détection des codelets dans les approches statiques de partitionnement puis dans le cadre des approches dynamiques.

1.4.1.1 La détection de codelets dans les approches statiques

Pour calculer les "*hotspots*" de manière statique, le programme est instrumenté, puis exécuté. Cette instrumentation fournit, après exécution, les données nécessaires à la localisation des "*hotspots*", c'est-à-dire le nombre d'occurrences de chaque instruction ou groupe d'instructions du programme [1, 3].

Dans le but d'obtenir des informations sur les chemins les plus suivis, on peut instrumenter le code de manière à pouvoir distinguer dans la trace d'exécution obtenue les différents chemins. On peut alors calculer les "*hotpaths*". Il s'agit d'un travail de détection de séquence, un algorithme de Karl, Miller et Rosenberg dit algorithme KMR [56], est très performant. C'est cette méthode qui a été adaptée par G. Pokam [11] que j'ai utilisée dans l'implantation d'ASTEX.

La figure 1.18 montre le processus de détection des codelets dans le cadre d'une approche statique. Le code original est instrumenté aux endroits stratégiques puis compilé et exécuté. On obtient alors une trace d'exécution à partir de laquelle on peut calculer la liste des *hotspots* ou des *hotpaths* dans le programme. Il est alors possible de les mettre sous forme de codelets.

Dans le cadre d'un partitionnement statique, la qualité du partitionnement dépend de la qualité des tests. En effet, le calcul des *hotspots* et/ou *hotpaths* étant effectué à

partir de traces d'exécution, les jeux de tests doivent être représentatifs de l'exécution moyenne du programme. Quel est alors le coût et la faisabilité de tels jeux de données ? Ce problème ne semble pas abordé dans la littérature concernant la détection statique de codelets.

1.4.1.2 La détection de codelets dans les approches dynamiques

Dans le cadre d'un partitionnement dynamique, les codelets sont formés et extraits pendant l'exécution du code. Le choix se fait par un *monitoring* du code qui tient à jour une table de vérification des dépendances de données et des tables de prédictions. Ici la forme d'un codelet n'est pas nécessairement fixe au cours du temps et la sélection des instructions extraites s'affine au fil des exécutions [4, 5].

Il existe deux mises en oeuvre possibles pour la détection dynamique : une matérielle et une logicielle. La première exclusivement matérielle [4] demande beaucoup de transistors pour l'analyse du code en temps réel, pour la mémoire nécessaire à la table de prédiction et l'analyse de dépendance de données. La seconde est une solution logicielle, avec une machine virtuelle régissant l'exécution du programme. Il en existe une implantation en JAVA [5], mais cette dernière nécessite quand même un mécanisme matériel de *monitoring*. La même approche avec un *monitoring* logiciel est 100 fois plus lente.

L'implantation TEST [5] propose une solution mixte, c'est-à-dire que le *profiling* se fait en monitorant les premières exécutions ; par la suite, on fige la décomposition et on poursuit l'exécution comme si la parallélisation avait été statique. Ceci se base sur l'hypothèse que le partitionnement du code par ce procédé converge rapidement vers une solution optimale au sens de Pareto [7]. Cependant, cette approche est inadaptée aux systèmes hétérogènes car elle permet pas l'exécution de code d'*ISA* différentes.

L'approche très préliminaire[85] basée sur une JVM JAVA et le *run-time* RapidMind [60] traite contrairement à [5] de l'utilisation d'architectures hétérogènes à partir d'un programme JAVA, en particulier du cas des GPU. Cette approche présente plusieurs inconvénients : la détection se limite aux nids de boucles détectés sans dépendances, le surcoût de profilage est bien présent et diminue l'accélération maximale pouvant être atteinte pour l'application et enfin les communications ne sont pas optimisées. Ce travail de par son étude du *run-time* et des mécanismes associés ainsi que du problème d'adaptabilité présente un axe intéressant d'étude. Il peut à terme, sous réserve d'optimisation des communications, représenter une alternative sérieuse à HMPP [42] pour le langage JAVA.

Dans le cadre d'architectures hétérogènes, les solutions dynamiques sont difficilement adaptables car elles nécessitent une recompilation de type *JIT*, c'est-à-dire à la volée, du code avant de pouvoir l'exécuter sur l'unité distante en plus du profilage de l'exécution. De plus, des communications et synchronisations avec un minimum d'optimisation sont nécessaires pour arriver à accélérer l'application [41].

1.4.2 La génération de code

Dans le cadre d'une détection dynamique des codelets, les codelets détectés sont les codelets extraits. Étant donné que ce type de sélection utilise des dispositifs matériels à mémoire, les codelets ont une *granularité*, *i.e.* une longueur en nombre d'instructions, limitée par la taille du dispositif. Pour ce qui est de l'extraction statique, le problème est plus complexe. Les codelets détectés ont des *granularités* différentes et peuvent se recouper. Il se pose donc le problème du choix des codelets à extraire.

Le premier point de cette partie évoque les différentes approches de sélection de la partition à implémenter. La seconde soulève les problèmes liés à l'implantation effective des codelets dans le programme partitionné.

1.4.2.1 La sélection du code

Après une pré-sélection basée sur la compatibilité entre la nature du code et les ressources matérielles à disposition, la sélection repose essentiellement sur la maximisation de la couverture en temps d'exécution dans le programme séquentiel du code à extraire, mais aussi sur la minimisation des surcoûts d'exécution. La mise au point d'une fonction d'objectif telle que la formulation de l'accélération potentielle du programme en fonction de la *granularité* des codelets et des surcoûts, est une manière d'approcher le compromis optimal.

Concernant le paramètre de *granularité*, on peut noter que plus le codelet est long plus les surcoûts sont absorbés ; cependant, la latence est plus forte en cas d'erreur sur la fin du codelet. Si le codelet est court, les surcoûts prennent une part importante du temps total d'exécution et donc l'accélération potentielle est réduite. Enfin, le surcoût de répartition des charges est directement proportionnel à la différence de *granularité* entre les différents codelets ; en cas d'exécution parallèle il est donc préférable de choisir un ensemble de codelets de *granularité* homogène.

Le second facteur de sélection est la présence de portions de code non parallélisables. Ces portions de code détectées statiquement sont caractérisées par la présence de structures de données complexes, d'*aliasing* entre les pointeurs et/ou d'un fort taux de dépendances de données, de type lecture après écriture [5]. Enfin, certains chemins et/ou données des *hotspots* ne sont pas, ou difficilement, prédictibles. Le code est donc dans ces cas là impossible à extraire avec une forte probabilité sur le gain en temps d'exécution.

1.4.2.2 L'implantation des codelets

Dans le cadre des approches statiques, tous les éléments de contrôles des codelets sont inclus dans le code de l'application. On y trouve donc les tests de détection d'erreur et les codes correcteurs. Lorsque c'est nécessaire il faut aussi généré dans l'application les synchronisations (dépendances) mais aussi les communications (dans le cadre des mémoires distribuées) ou tous autres mécanismes nécessaires à la gestion de l'exécution.

Ces éléments doivent être générés statiquement et éventuellement évalués dynamiquement (c'est le cas des tests d'erreur par exemple). Ces modifications du code augmentent sa taille et handicapent sa probabilité et son adaptabilité. Des approches basées sur un ajout d'expressivité pour le parallélisme dans le langage de haut niveau comme *HMPP* limitent les facteurs négatifs des approches statiques en laissant l'application portable et en laissant la possibilité de revenir à la version initiale du code par simple désactivation du partitionnement.

Dans le cadre des approches dynamiques, le code de l'application reste inchangé. La complexité du processus de partitionnement est repoussée dans l'architecture matérielle. Cette dernière est donc complexifiée par l'intégration de nouveaux mécanismes pour la performance ; il faut intégrer l'évaluation des tests, les communications, les stockages d'information pour les accès et les portions de code à extraire,.... Surface de silicium, consommation électrique et complexité des circuits en pâtissent.

Il existe des solutions hybrides comme [5] ou [77]. Dans cette dernière, Rus and co. introduisent un système hybride de parallélisation automatique. Il s'agit de générer statiquement les tests qui décident de l'exécution parallèle ou non de sections de code signalées comme intéressantes à extraire. Le partitionnement effectif lui, reste dynamique. Il s'agit cependant d'un compromis ; l'impact sur le code source est limité, mais le côté dynamique a perdu de son adaptabilité : on ne peut extraire que les parties prévues statiquement. Par ailleurs, cette approche ne traite pas des mémoires non cohérentes et/ou distribuées.

1.5 Optimisation des communications

Dans la section 1.3.1.2 a été abordé le problème de la gestion des communications et ses voies possibles d'optimisation :

- Limiter le volume et la fréquence des communications.
- Masquer au maximum les latences en préchargeant au plus tôt les données.

Trouver le meilleur compromis entre fréquence, volume, et l'influence que peut avoir le préchargement sur ces deux paramètres est un problème complexe d'optimisation.

La limitation de volume peut s'effectuer grâce à une analyse statique fine du code et l'ajout de tests et/ou mécanismes d'historique de l'exécution pour affiner les zones que l'on doit considérer comme potentiellement accédées par le codelet. Il est aussi possible de faire appel à la spéculation quand les données statiques ne suffisent plus.

Quand l'architecture le permet, afin de masquer les latences, il est possible de précharger les données. Là encore, il faut analyser le code de manière à obtenir les informations sur les accès aux données. Les problèmes d'*aliasing* ou de structures de contrôles complexes limitent le déterminisme de l'analyse statique du code et le résultat obtenu par ces seules informations peut ne pas être satisfaisant. L'usage de la spéculation permet de pousser plus loin les optimisations. Il faut cependant s'assurer que quelque soit le chemin du code exécuté toutes les communications nécessaires à l'exécution du codelet

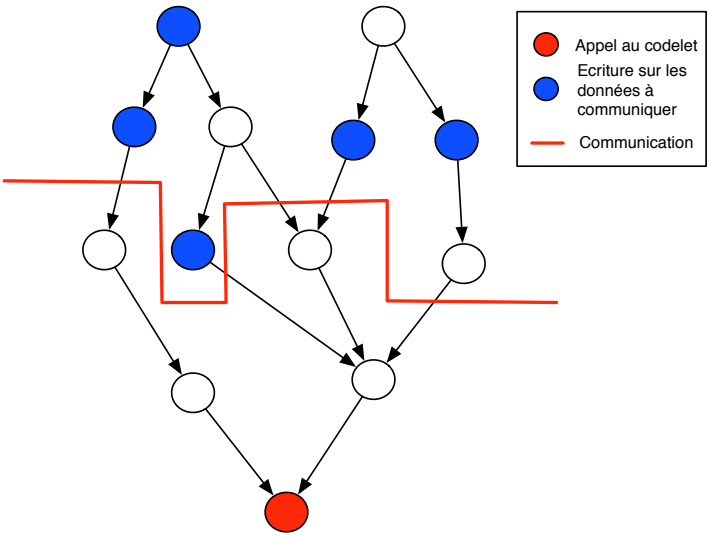

FIG. 1.19 – *CFG* d'une application. Le nœud rouge est le point d'appel au codelet. Les nœuds bleus modifient les zones mémoire nécessaires au codelet ; il faut donc que les communications des données correspondantes interviennent avant l'appel. Ces communications interviennent sur les arcs coupés par la ligne rouge. Le *CFG* est "coupé" en deux parties par cette ligne.

ont été exécutées.

L'optimisation du préchargement des données est guidée par la minimisation du coût de communication. Se basant sur une fonction de coût, l'objectif de l'optimisation est de trouver la coupure du *CFG* qui assurera la présence des données lors de l'exécution du codelet avec le coût minimal, et ce pour chaque zone à communiquer. La figure 1.19 représente une telle coupure dans un *CFG*. C'est le tracé de la ligne rouge qu'il faut optimiser. Ce problème est très proche dans sa formulation de celui du MaxCut [16, 25] et du MinCut/MaxFlow. C'est respectivement l'objet des deux premières sections de cette partie. La troisième discute du rapport de ces deux problèmes à celui d'optimisation des communications.

L'idée de *couper* le graphe de contrôle de flot à des endroits stratégiques pour minimiser un coût est aussi celle employée dans les méthodes de minimisation du placement des barrières de synchronisation [18, 19]. Ces dernières ont pour rôle dans les systèmes *SPMD* de casser les dépendances de données entre les différentes tâches. Présentant des similitudes avec le problème de l'optimisation des communications dans sa formulation, j'introduis brièvement ce problème et ses solutions dans la deuxième section de cette partie. Je conclus sur les différences qu'il comporte avec notre problème.

Enfin, la troisième section aborde plus directement les différentes techniques d'optimisation des communications que l'on peut déjà trouver dans les systèmes "multithread" issus de processus de partitionnement d'application.

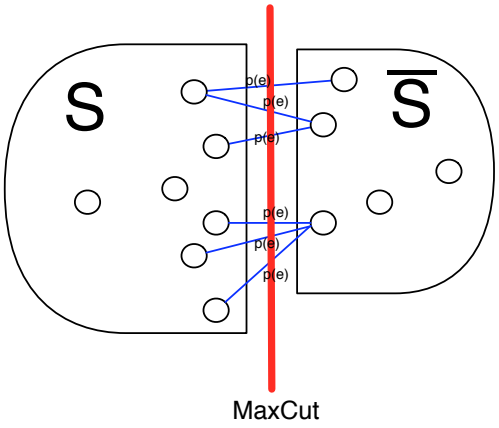

FIG. 1.20 – Représentation graphique du problème du MaxCut faisant référence à la définition 1.1

1.5.1 Le problème du MaxCut

Définition 1.1
Soit $G(V, E)$ un graphe non orienté G ayant comme ensemble de nœuds (vertices) V et E comme ensemble d'arcs (edges).
Soit p une fonction de E dans $\mathbb{R}^+$ définissant le poids des arcs.
La coupe maximale (MaxCut) est l'ensemble S des nœuds du graphe tel que la somme du poids des arcs ayant une extrémité dans S et l'autre dans $\overline{S}$ soit maximale.

Le problème de la coupe maximale dans un graphe est connu pour être NP-complet dans le cas général [16, 25, 78]. La figure 1.20 est la représentation graphique de la définition 1.1

Il y a deux façons d'aborder ce problème :

- La première consiste à limiter la portée du problème avec des hypothèses restrictives sur la nature du graphe,
- La seconde approche consiste à mettre en place des heuristiques afin d'orienter les choix et ainsi de diminuer la complexité.

La première solution peut conduire à des solutions optimales en un temps polynomial, comme dans [17] pour les arbres de degré borné.

L'autre a pour objectif de donner une solution jugée "bonne" avec une complexité jugée raisonnable à un problème resté NP-complet, c'est le cas dans [78] ou la solution est au minimum 87,8% de la solution optimale.

Les domaines d'applications liés au MaxCut sont nombreux [78, 80, 81]. Il est en particulier lié au problème de la minimisation du placement des barrières de synchronisation [18, 19] et à l'*ordonnancement* des tâches sur les machines parallèles [82].

1.5.2 Le MinCut et l'algorithme de Ford-Fulkerson

Le problème de l'optimisation des communications est la minimisation du coût de la coupure du CFG du programme entre les accès au variable accédé dans le codelet et les éléments constitutifs du codelet. Considérant les accès comme une source et le codelet comme un puits, l'énoncé de notre problème devient similaire à celui du MinCut/Maxflow.

Cette partie présente donc dans ses deux sections respectivement : le problème du MinCut/MaxFlow et l'algorithme de Ford-Fulkerson permettant de le résoudre en un temps polynomial.

1.5.2.1 Le MinCut/MaxFlow

Définition 1.2
Soit $G(V, E)$ un graphe non orienté G ayant comme ensemble de nœuds (vertices) V et E comme ensemble d'arcs (edges).
Soit p une fonction de E dans $\mathbb{R}^+$ définissant le poids des arcs.
Soit Src et $Dest$ éléments de V représentant respectivement la source et la destination d'un flux. La coupe minimale (MinCut) entre Src et $Dest$ est l'ensemble S des nœuds du graphe tel que :

- *$Src \in S$ et $Dest \in \overline{S}$*
- *la somme du poids des arcs ayant une extrémité dans S et l'autre dans $\overline{S}$ est minimale.*

La recherche du MinCut dans un graphe trouve son utilité dans de nombreux problèmes de flôt. En effet, elle est équivalente au problème du calcul du flot maximum dans un graphe : le MaxFlow. Cette équivalence est énoncée dans le théorème 1.1

Théorème 1.1 *Théorème flot-max/coupe-min (P. Elias, A. Feinstein et C.E. Shannon, et L.R. Ford, Jr. et D.R. Fulkerson, 1956)*

Pour tout graphe, tout couple (s,t) de sommets du graphe, et pour toute pondération positive, la valeur maximum du flot de s à t est égale à la capacité minimum d'une coupe séparant s de t.

Contrairement au MaxCut, une solution optimale au problème du MinCut peut être obtenue en un temps polynomial. Un des algorithmes possibles est celui de Ford-Fulkerson [83], basé sur l'existence d'une série de chemins augmentants, qui permet de construire la solution.

1.5.2.2 L'algorithme de Ford-Fulkerson

Les notations de cette section reprennent celle de la définition 1.2.

Le poids d'un arc dans la définition est la capacité c de l'arc. Un flux de valeur n passant par l'arc diminue sa capacité de n. Il a alors une capacité résiduelle égale à $c - n$. Si la capacité résiduelle devient nulle, l'arc est dit saturé. Le graphe privé de ses arcs saturés est le graphe résiduel.

L'algorithme se base sur le principe suivant : tant qu'il existe un chemin de la source au puit non saturé, on envoie un flux sur ce chemin. Un chemin non saturé est un chemin dit augmentant. En pseudo-code, l'algorithme s'énonce ainsi :

```
TantQue il existe un chemin augmentant de s vers p Faire
   Pour tous les arcs appartenant au chemin Faire
      mettre à jour la capacité résiduelle
   finPour
   chercher un chemin augmentant
finTantQue
```

L'algorithme du plus court chemin de Dijkstra [84] permet de trouver, si il existe, le chemin le plus court de la source au puit en $(O)(n^2)$, n étant le nombre de sommets du graphe. Ce chemin est un chemin augmentant.

Le calcul du flot maximum se termine lorsqu'il n'existe plus de chemin augmentant. Si sur chaque chemin trouvé on sature un arc, le nombre d'arcs étant fini, le nombres de chemin augmentants qui ne contiennent pas d'arc saturé converge donc vers 0.

Théorème 1.2 *L'algorithme de Ford-Fulkerson trouve un flot maximal avec une complexité* $(O)(n^2M)$ *où* M *est la capacité maximale d'une arête et* n *le nombre de sommet du graphe.*

Il existe des raffinements de l'algorithme de Ford-Fulkerson permettant d'améliorer le facteur M. Citons entre autres l'algorithme de Edmonds-Karp.

Le passage du MaxFlow trouvé par Ford-Fulkerson au MinCut est trivial. Le MinCut est l'ensemble des arcs sortant du graphe résiduel, *i.e.* les arcs dont la source est dans le graphe résiduel et leur destination non. Mis en rapport avec les notations de la définition 1.2, le graphe résiduel est l'ensemble S.

1.5.3 Le MaxCut / MinCut et l'optimisation des communications

Le problème d'optimisation est celui d'une coupe minimale sur une restriction du *CFG* d'une application. Cette restriction est l'ensemble des chemins allant des accès hors codelet de la variable à communiquer au codelet. Une définition formelle du problème et de la restriction du CFG est donnée dans le chapitre 3. La redondance des communications introduit un surcoût. L'introduction d'un arc dans la coupe minimale peut potentiellement influer sur le poids de tous les autres.

La représentation graphique du problème est donnée dans la figure 1.21.

Le problème portant sur la construction d'une coupe minimale, il est logique de le rapprocher de celui dit du MinCut donné par la définition 1.2. Il existe une équivalence entre notre problème et celui de la coupure minimale dit MinCut dans le cas d'une fonction de coût invariante. En effet, une construction simple par ajout de sommet permet de retrouver la définition 1.2. La figure 1.22 illustre cette construction. Soit Src et $Dest$ deux nouveaux sommets reliés par des arcs de coût infini respectivement à tous

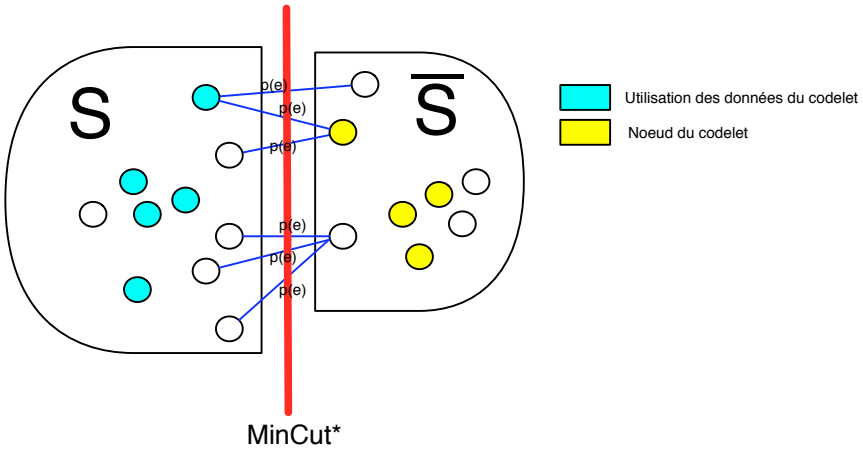

FIG. 1.21 – Trouver le placement des communications de coût minimal entre un codelet et les accès aux variables accédées par le codelet revient à trouver l'ensemble S qui contient l'ensemble des accès mais aucun nœud du codelet tel que le poids cumulé des arcs entre S et $\overline{S}$ soit minimal.

les sommets d'où sont issues les communications et tous les sommets composant les utilisations.

Lors de la construction du MinCut entre *Src* et *Dest*, tous les sommets d'où sont issues les communications seront nécessairement dans S et tout les sommets composant les utilisations seront dans $\overline{S}$. En effet, le coût des arcs concernés étant infini, ils ne peuvent trivialement pas être dans la coupe minimale.

La valeur de coût des arcs, dans notre problème d'optimisation, est réentrante. Ceci implique qu'il n'existe pas de sous-problème optimal permettant de construire une solution globale : la suite convergente des chemins augmentants de Ford-Fulkerson n'existe plus. En effet, une capicité résiduelle nulle pour un arc peut redevenir non nulle à cause de la redondance. Rien ne peut alors garantir la terminaison de Ford-Fulkerson.

Une solution simple au problème d'optimisation du placement des communications peut être d'utiliser l'algorithme de Ford-Fulkerson sans remise à jour des arcs déjà saturés. Il s'agit d'une heuristique garantissant la convergence. Les cas de dysfonctionnement de Ford-Fulkerson nécessitant l'utilisation de l'heuristique sont detaillés dans le chapitre 3.

Une autre approche possible au problème d'optimisation du placement des communications est celle d'un algorithme dédié faisant une combinaison de restriction du problème et d'heuristique pour arriver à la solution. Ce type d'approche utilise des principes similaires à ceux employées dans la résolution du MaxCut.

La discussion sur le choix de l'algorithme à adopter et les différentes heuristiques possibles pour le placement sont l'objet du chapitre 3.

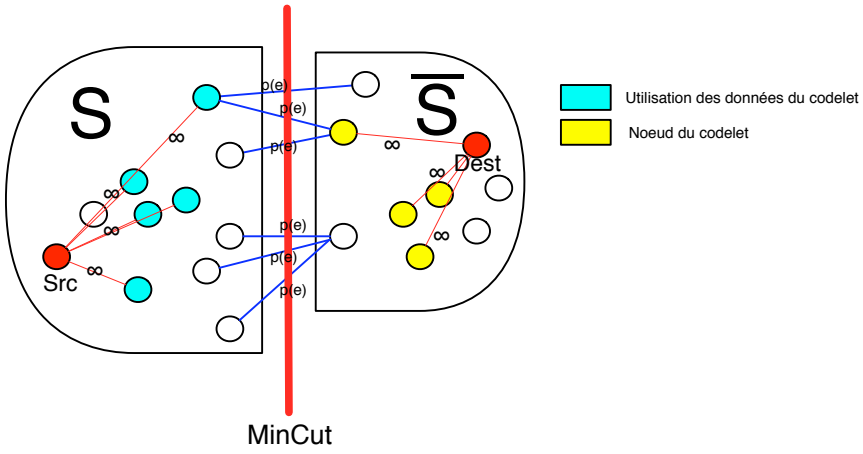

FIG. 1.22 – Transformation du problème d'optimisation des communications dans la forme de celui du MinCut par l'ajout des nœuds *Src* et *Dest*.

1.5.4 Le problème du placement des barrières de synchronisation

La figure 1.23 représente l'exécution de tâches éventuellement parallèles dont les dépendances sont représentées par les arcs reliant les blocs. Afin d'assurer la validité des valeurs des variables utilisées par les différentes tâches lors de leur exécution, il faut que chaque dépendance ait été *cassée*. C'est dans ce but que sont utilisées les barrières de synchronisation [18, 19].

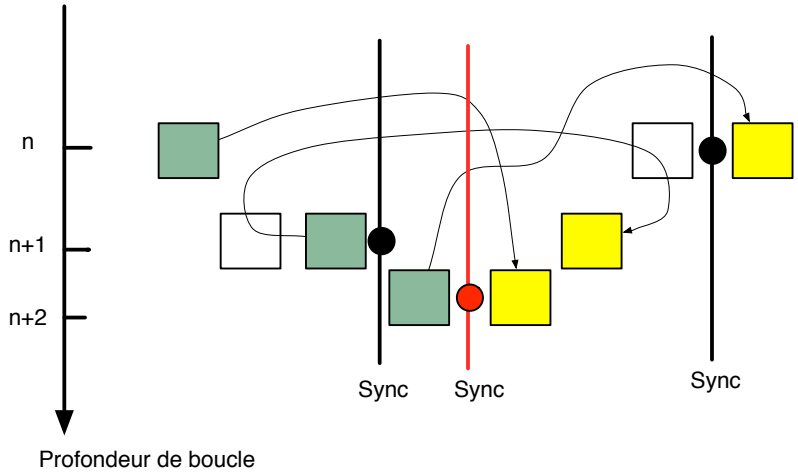

FIG. 1.23 – Placement de barrières de synchronisation dans un programme. En noir et en rouge figurent deux solutions possibles.

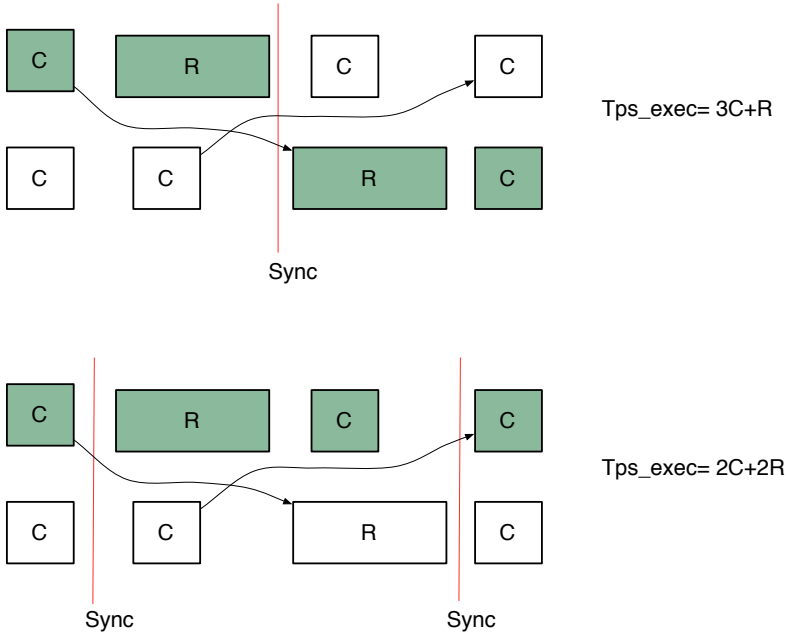

FIG. 1.24 – Deux solutions du placement des barrières de synchronisation pour un même code afin de casser les dépendances de données figurées par les flèches. Le temps d'exécution pour le code à deux barrières est plus court qu'avec une seule. Cet exemple est tiré de la référence bibliographique [19]

Dans l'exemple de la figure 1.23, une seule barrière figurée par un trait rouge suffit à couper toutes les dépendances de données. La solution figurée en noir en utilise deux. Cependant à l'exécution, la solution en rouge donnera lieu à plus d'exécution que la solution en noir. L'optimisation du placement des barrières de synchronisation est donc équivalente à la minimisation du nombre de barrières exécutées.

Les algorithmes comme dans [18, 19] parle de solution optimale pour la minimisation du nombre de barrières de synchronisation. Cependant, ce n'est pas nécessairement la solution optimale pour le temps d'exécution du programme parallèle. L'exemple de la figure 1.24 est issu du travail présenté par A. Darte et R. Schreiber dans [19]. La solution à une barrière de synchronisation présente un temps d'exécution de deux carrés et deux rectangles. Or la solution à deux barrières possède un temps d'exécution égal à trois carrés et un rectangle. Considérant un coût de barrière inférieur à celui d'un carré, le temps d'exécution de la solution avec deux barrières est inférieur.

Le problème global de génération du code synchronisé optimal est donc issu du couplage synchronisation/ordonnancement. Le problème conjoint de diminution des dépendances de données et de minimisation du nombre de barrières de synchronisation à l'exécution est NP-Difficile dans le cas général [18] mais linéaire dans le cas des

boucles imbriquées [19]. C'est le problème de génération de code parallèle optimal qui est indécidable [79].

Comme pour le MaxCut, les solutions proposées dans la littérature sont basées soit sur des heuristiques soit sur une restriction du problème. Dans le problème de la gestion de dépendance de données dans les systèmes *SPMD*, l'utilisation d'heuristiques donne de très bonnes solutions [18]. Encore une fois, dans certaines situations, une solution optimale à un sous-problème peut être trouvée [19].

Les solutions adoptées dans le cadre de la minimisation des barrières de synchronisation ne s'appliquent pas au problème de l'optimisation des communications traité dans mon travail de thèse.

La fonction de coût des barrières de synchronisation est simple, c'est une fonction du nombre de barrières, de leur fréquence respective d'exécution et éventuellement du volume de données à synchroniser. La redondance, *i.e.* une barrière casse deux fois la même dépendance, n'a pas d'influence. Enfin, les temps d'exécution des blocs entre les synchronisations n'entrent pas en compte alors qu'ils sont essentiels dans notre problème pour masquer les latences de communication.

1.5.5 Les techniques d'optimisation des communications

Il est possible d'utiliser pour l'optimisation des communications des approches similaires à celles des barrières de synchronisation sur les graphes de *flot de données* [21, 22]. Fahinger dans [22] insiste sur l'importance du compromis entre masquer la latence en préchargant au plus tôt et réduire le nombre d'occurrence des communications. Étant donnée la complexité du problème d'optimisation des communication, ce compromis semble un paramètre important pour l'exploration de l'espace des solutions.

D'autres approches comme celle de Banerjee dans [20] sont basées sur le graphe de contrôle de flot. L'approche [20] utilise la forme SSA du code et combine un placement statique et dynamique du code ; c'est-à-dire que des tests sont générés statiquement pour déterminer quel sera à l'exécution le point ou s'effectue la communication. C'est ce dernier point et son surcoût associé que notre approche tente de résoudre. Ceci grâce à l'utilisation des données spéculatives fournies par Astex. L'approche [20] est elle aussi utilisée dans le contexte de la génération automatique de code parallèle à partir de code séquentiel. Elle utilise un algorithme à heuristiques. Cependant, elle ne traite pas des regroupements de communications ni de l'*aliasing*. Chaque variable du programme est considérée indépendamment des autres.

Enfin, il existe des approches comme [26, 59], qui utilisent un processus en parallèle de l'exécution du programme principal pour calculer les préchargements nécessaires. Ces techniques font un calcul spéculatif du chemin suivi par l'exécution et en déduisent les données à précharger. Cette technique se retrouve couramment sous la dénomination *helper thread*.

Notre travail diffère des précédentes approches car il se base sur un système hybride

de données statiques et de données spéculatives produites par ASTEX. Ces informations facilitent un placement optimisé des communications par rapport à une approche complètement statique.

La spéculation intervient principalement à deux niveaux :

- La détermination entre vrai accès et faux accès : Les accès probables et/ou sûrs sont optimisés en priorité. Les accès peu ou pas probables ont une politique d'optimisation différente.
- Les données à copier : quelles zones mémoire et quelles tailles des segments de données à copier.

La combinaison des deux permet par ailleurs de déterminer des zones de mémoires résidentes, c'est à dire qui peuvent rester sur la mémoire hôte entre deux appels de codelet.

1.6 Conclusion

Au vu de l'actualité récente en architecture et des recherches publiées actuellement dans la littérature, la conception d'architecture s'oriente vers les systèmes multicœurs hétérogènes et à mémoires distribuées, souvent avec un espace d'adressage global mais pas nécessairement cohérent. Bien que n'étant pas absolue, et soumise à évolution, c'est cette tendance qui semble guider la conception des systèmes aussi bien dans l'embarqué que dans le *HPC* en passant par les machines dites de bureau.

Afin d'adresser efficacement ces architectures, on observe un développement rapide d'outils pour exploiter les nouvelles ressources offertes par les systèmes. L'exemple le plus représentatif étant CUDA de NVIDIA pour les *GPUs* et son successeur OpenCL qui est destiné à être une référence commune entre les principaux acteurs de la conception et programmation d'accélérateurs matériels.

HMPP de *CAPS entreprise* est un autre modèle de programmation pour les architectures hétérogènes, il fournit aussi un environnement d'exécution logiciel des codelets. Il répond efficacement aux contraintes de portabilité, d'hétérogénéité des architectures, des techniques spéculatives, et reste évolutif. C'est le support qui, eu égard aux relations étroites de mon travail de thèse avec les activités de *CAPS entreprise*, s'est naturellement imposé comme support à mes expérimentations.

Il n'existe pas encore de solution aux problèmes du partitionnement et/ou parallélisation du code ; cependant, il ressort des approches étudiées dans ce chapitre des résultats encourageants en terme de performances. Bien évidemment, tous les systèmes ne sont pas équivalents mais il est toutefois possible de noter les deux cas suivants :

- Dans le cadre d'une extraction statique de codelets le gain minimum est de l'ordre de 15-20% dans la quasi-totalité des cas [1, 3]. Le gain peut se situer aux alentours de 400% pour certains programmes de compression. [3].

– Dans le cadre d'une extraction dynamique du code le gain se situe dans une fourchette de 15 à 25% pour la grande majorité des programmes et atteint 75% dans certains cas [5].

Toutes les approches précédemment citées, statiques ou dynamiques, sont basées sur l'exploitation du parallélisme.

Cependant, la plupart des approches sont fondées sur l'utilisation de dispositifs matériels pour la gestion des codelets. Ces différentes approches sont pour la plupart difficilement adaptables aux systèmes enfouis. En effet, ces approches impliquent un coût matériel élevé, ce qui induit nécessairement une augmentation de la surface des puces, donc de leur consommation, de leur dissipation de chaleur ainsi que de leur coût de fabrication [7]. Cette omniprésence de matériel est le fruit d'hypothèses fortes sur la gestion de la mémoire, ceci afin d'en assurer le partage et la cohérence [1, 2, 4, 5, 8]. Ces hypothèses n'étant pas toutes les mêmes suivant les approches [2, 5, 6], la détection des codelets et leur gestion varient elles aussi. Ces procédés ne sont donc pas applicables d'une manière générale, ils dépendent de l'architecture à laquelle ils sont destinés. À l'heure actuelle il ne semble pas exister de véritable mécanisme d'extraction de codelets indépendant de l'architecture. Une approche générique pourrait fournir un bon préambule à la création à moindre coût de code partitionné multi-plateforme. L'apparition d'architectures spécialisées, permettant des exécutions déportées de codelets (avec ou sans parallélisme) telles que les SoCs [6], le Cell [44] et le *GPGPU* [54, 67],... crée une véritable demande de ce côté. C'est le problème auquel Astex, la première contribution de ma thèse, propose une solution.

Dans l'exécution distribuée de programmes sur des accélérateurs matériels, les communications sont un enjeu majeur pour l'obtention des performances attendues. Ce problème n'étant devenu critique que relativement récemment, la littérature sur l'optimisation des communications dans le cadre de l'utilisation d'accélérateurs matériels est assez limitée. Un parallèle avec les techniques d'optimisation du placement des barrières de synchronisation, et les aspects formels du problème du MinCut, sont une bonne base pour l'élaboration d'une technique d'optimisation des communications dans le cadre spécifique du partitionnement automatique d'application pour les systèmes à mémoires distribuées. La technique proposée dans le troisième chapitre utilise et complète les informations fournies par l'approche hybride de partitionnement d'Astex pour optimiser de manière globale les communications propres à chaque codelet.

Chapitre 2

ASTEX : Automatic Speculative Threads EXtractor

Les dernières générations de processeurs et coprocesseurs ont des performances potentielles très élevées [43, 44, 46, 48, 49, 50, 54, 67, 69]. Elles sont atteintes grâce à l'utilisation de systèmes massivement parallèles. Cependant, afin de tirer parti efficacement de ces architectures, il est nécessaire de faire apparaître au sein des nouvelles, mais aussi anciennes applications, les portions de code susceptibles d'exploiter au mieux les performances et les nouvelles ressources disponibles.

La mise au point d'un processus automatique de partitionnement évite aux programmeurs la réécriture manuelle de tous les programmes pour chaque génération/type d'architecture.

Dans les parties 1.4 et 1.3 du premier chapitre de cette thèse, plusieurs méthodes de partitionnement automatisées ont été proposées. Elles se focalisent sur le parallélisme, font souvent l'hypothèse de mémoire partagée, de système homogène et de mécanismes matériels pour la détection et l'exécution des codelets. Aucune de ces approches, sans modifications majeures, n'adresse efficacement les problèmes posés par les architectures actuelles, en particulier l'utilisation d'accélérateurs matériels comme coprocesseurs. Partitionner automatiquement et efficacement les applications dans le cadre du parallélisme et/ou l'utilisation d'accélérateurs matériels reste donc un problème ouvert.

La méthode exposée dans ce chapitre et implantée par ASTEX repose sur l'utilisation de la spéculation en complément des données statiques d'un programme pour la détection, la caractérisation, mais aussi l'exécution des codelets.

L'objectif du projet est la mise au point d'un procédé automatique de partitionnement de programme en codelets spéculatifs. Afin de garantir la portabilité du résultat, nous ne faisons aucune hypothèse sur l'environnement d'exécution des codelets, logiciel ou matériel. À partir de sa description, la partition de l'application est construite directement dans le programme non compilé, c'est-à-dire dans son langage de haut niveau. L'implantation actuelle d'ASTEX travaille sur le langage C. Par ailleurs, les données

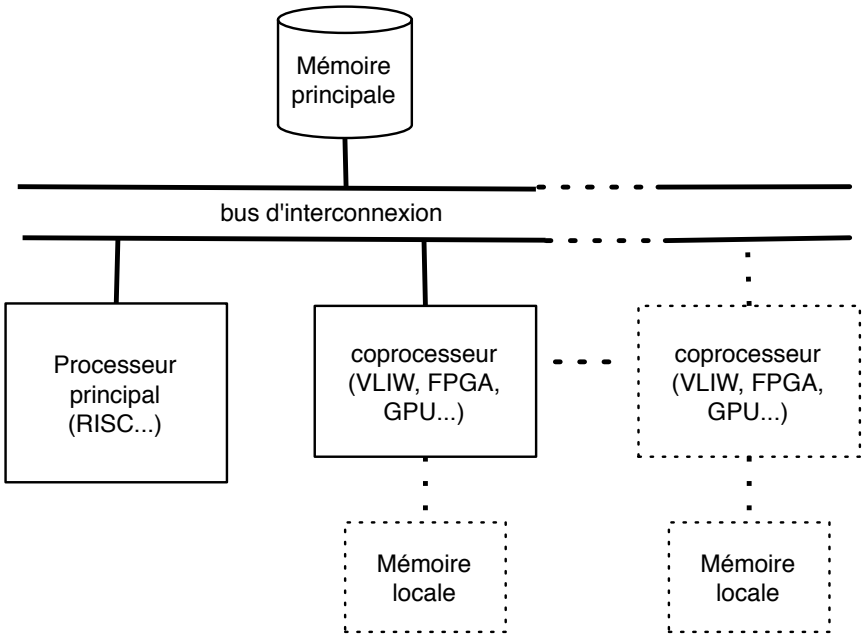

FIG. 2.1 – Modèle général d'architecture cible.

produites sur l'application et le modèle d'exécution choisi permettent la génération et l'exécution alternative des différentes versions d'un même codelet. C'est un point essentiel pour l'adaptabilité de l'application en sortie.

Le modèle général d'architecture servant de base à la conception de notre processus de partitionnement est représenté dans la figure 2.1. Un processeur principal exécute les applications. Un ou plusieurs coprocesseurs avec éventuellement leur propre mémoire locale peuvent prendre en charge l'exécution d'une sous-partie du programme.

Typiquement, le processeur principal est une architecture de type RISC alors que le/les coprocesseurs peuvent être de type VLIW ou FPGA ou toutes autres architectures spécialisées. La mémoire principale peut être partagée mais l'on ne fait aucune hypothèse sur la présence de mécanismes matériels de gestion de la cohérence mémoire entre les différentes unités. Le processeur Cell d'IBM [44, 45] est un exemple de ce type d'architecture.

Le but d'une telle architecture est d'exploiter les différentes unités sur les portions de code où elles sont le plus efficaces. L'unité principale peut alors traiter d'autres tâches pendant que le ou les coprocesseurs exécutent éventuellement en parallèle leurs codelets. L'exploitation de l'architecture est basée sur le partitionnement du programme en sous-programmes distincts et la gestion des communications et des synchronisations entre chacun de ces sous-programmes. Le partitionnement doit donc tenir compte des surcoûts de gestion de l'exécution et de la nature du code extrait.

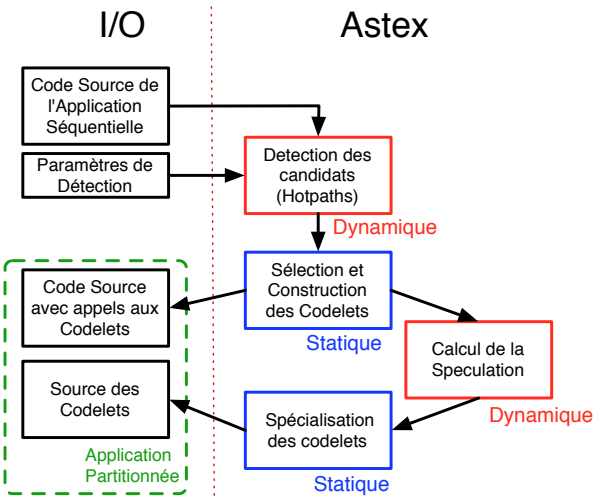

FIG. 2.2 – implantation du Pipeline d'Extraction dans ASTEX

Le processus d'extraction implanté par ASTEX suit les étapes telles que décrites dans la figure 2.2. Tout d'abord, il s'agit de détecter les candidats potentiels à l'extraction. En accord avec la loi d'Amdahl, la détection se focalise sur les parties de calcul intensif du code, c'est-à-dire avec un ratio calcul/longueur du code le plus élevé possible. Une fois la détection des candidats achevée, on sélectionne une partie des portions de code candidates pour les implanter sous forme de codelets. Par instrumentation puis exécution, on génère un profil à partir duquel il est possible de calculer le comportement spéculatif du code et de la mémoire. Ceci permet d'inférer certaines propriétés du code et des données (taille, alias, alignement...) et ainsi permettre différentes optimisations et spécialisations du code. Si les résultats sont satisfaisants, on a alors obtenu une partition du code. Afin d'optimiser le choix des codelets, il est possible, suite à l'évaluation, d'explorer des solutions alternatives afin d'optimiser la partition en sortie. Cette étape de la compilation devient alors itérative.

Dans la première partie de ce chapitre sont exposées les motivations qui ont guidé la mise au point de notre processus de partitionnement automatique d'application en codelets spéculatifs : ASTEX. Il s'agit de montrer pourquoi le choix d'une approche spéculative a été fait, ainsi que d'expliquer comment intégrer ces nouvelles données afin de rendre plus efficaces et pertinentes les partitions calculées par notre procédé.

Dans un second temps est décrit en deux parties, 2.2 et 2.3, le modèle théorique d'ASTEX. Il s'agit respectivement de la description des modèles représentant tous les éléments intervenant dans un processus de partitionnement d'application, puis la description d'ASTEX lui-même.

La section 2.4 présente l'implantation effective de ASTEX. C'est à partir de cette implantation qu'ont été effectuées les expérimentations de la partie 2.5. Ces dernières portent tant sur des critères quantitatifs tels que la performance, que sur des critères qualitatifs telle que la pertinence.

Ce chapitre conclut sur le résumé des contributions de ASTEX au domaine du partitionnement automatique d'application, ainsi que sur les perspectives et futur travaux connexes à notre approche.

2.1 Principe de fonctionnement d'une approche hybride de partitionnement

L'approche mise au point dans le cadre de ma thèse se distingue par la source des données sur lesquelles se base le processus de partitionnement pour identifier, former et spécialiser les codelets formant une partition. Elles sont de deux natures :

- Les données statiques, issues d'une analyse statique du code. Il s'agit d'algorithmes classiques tels que ceux cités dans [74]
- Les données dynamiques, issues de *profil d'exécution*. À partir de différentes *traces* d'exécution d'un programme sont inférées des propriétés du code. Étant issues du résultat d'une ou plusieurs exécutions, ces données sont spéculatives.

Notre processus d'extraction s'effectue pendant l'étape de compilation. Il ne s'agit pas d'une approche dynamique mais d'une approche statique qui se base sur des données dynamiques résumées dans un profil. Elle peut être qualifiée de processus hybride. Nous ne faisons aucune hypothèse sur l'*environnement d'exécution*.

Dans un premier temps, cette partie identifie clairement les objectifs que l'on s'est fixé pour notre approche du processus de partitionnement d'application en codelets spéculatifs.

Dans un second temps, j'établis le rôle de l'analyse statique, de l'analyse des données dynamiques, et comment ces résultats hétérogènes s'insèrent dans le processus de partitionnement.

2.1.1 Problématique visée

Outre remplir les objectifs essentiels d'un processus statique de partitionnement d'application, mon approche contribue à repousser les limites des méthodes actuellement présentes dans l'art. Dans le même temps, l'approche est étendue aux systèmes hétérogènes à mémoire distribuée, ceci afin de répondre efficacement aux nouvelles contraintes matérielles et logicielles imposées par la tendance actuelle dans l'évolution de l'architecture des systèmes.

Cette section rappelle dans un premier temps les objectifs principaux d'un processus de partitionnement d'application. Dans un second temps, j'identifie clairement les limites des approches purement statiques ou purement dynamiques auxquelles l'hybridation statique - profils dynamique peut apporter des réponses.

Enfin je fais état des contraintes liées à l'adaptation du processus de partitionnement aux systèmes hétérogènes à mémoire distribuée en particulier en terme de portabilité et adaptabilité du code produit.

2.1.1.1 Objectifs et limites d'un processus de partitionnement

Le rôle d'un processus de partitionnement d'application est de partager un programme en plusieurs tâches afin de les répartir au mieux sur les unités de calcul disponibles à l'exécution. L'objectif final est de maximiser l'intersection entre les différentes portions de code identifiées et les capacités de traitement efficace de l'architecture selon les critères utilisateurs.

Extraire sous forme de codelet une portion de code d'un programme pour l'exécuter de manière déportée, en parallèle ou non, est un problème complexe. Un grand nombre d'informations sur le code est nécessaire pour la génération de chaque codelet : sur le contrôle, sur les accès mémoire, sur les dépendances de données, sur la nature des calculs... Lorsqu'une information manque, la mise sous forme de codelet devient impossible. Il peut être alors efficace de remplacer l'information manquante par une information synthétique qui demandera à être vérifiée à l'exécution.

Bien que la spéculation permette de traiter une complexité accrue du contrôle et de la topologie mémoire (alias), certaines structures de contrôle, comme les sauts dont la destination est déterminée dynamiquement (*goto*), ou les accès mémoire à des structures récursives et au multiples indirections, restent un problème.

Dans les processus purement statiques, les incertitudes sont un problème majeur. Les indéterminations sur les *alias*, par exemple, conduisent à surestimer les contraintes sur les données et ainsi créent de fausses limitations, en particulier au niveau des dépendances. Ce type d'approche se retrouve donc très vite limité et son bon fonctionnement est tributaire de la forme originale du code fourni par l'utilisateur.
Les processus statiques de partitionnement d'application actuels implantent leurs modifications directement dans le code de l'application avant la compilation. Une fois généré, ce code est compilé dans le langage destiné aux différents processeurs visés. L'avantage certain est l'absence de compilation dynamique et donc de surcoût associé, ceci permettant d'adresser les systèmes hétérogènes. Cependant, le code de l'application n'est plus portable et son adaptabilité est nulle.

Dans un processus faisant usage de la spéculation dynamique, les contraintes sur les sections de codes que l'on veut extraire et éventuellement paralléliser sont beaucoup plus faibles. Effectivement, le nombre d'informations disponibles est plus grand et les informations manquantes sont, soit prédites d'après le code déjà exécuté, soit inférées plus ou moins au hasard. Dans les deux cas, on doit vérifier la validité de l'exécution et éventuellement corriger les erreurs afin de garantir à l'utilisateur la bonne exécution de son programme. Cependant, l'accélération possible est grandement limitée par les surcoûts de surveillance de l'exécution (profilage), la mémoire nécessaire pour garder

ces données et le code à extraire, les mécanismes d'appel et de retour des codelets avec leurs tests générés à la volée et donc peu optimisés et enfin, les mécanismes de correction d'erreur. Pour être performantes, ces approches usent de mécanismes matériels spécifiques, ce qui limite leur adaptabilité, leur portabilité et augmente leur coût de mise en oeuvre.

Qu'elles soient statiques ou dynamiques, les approches de partitionnement automatiques s'adressent aujourd'hui à un nombre de programmes très restreint et avec une efficacité en terme de performance limitée, voire très limitée.

L'ajout de données dynamiques dans un processus statique de partitionnement permet de combler les données manquantes et ainsi d'enlever les limites des approches statiques sans hériter de tous les surcoûts des approches dynamiques. L'hybridation statique-dynamique évite de faire les choses au hasard comme le font les approches spéculatives actuelles. Bien évidemment, il résulte de cette technique spéculative un surcoût qu'il conviendra d'évaluer afin de se comparer aux approches statiques et dynamiques existantes.

Dans le cadre d'une approche statique ou hybride, il est possible de s'assurer de la portabilité du code généré par l'utilisation de langage de directive désactivable comme *HMPP* [42]. Si le compilateur n'active pas l'interprétation des directives, le code reste celui d'origine, la modification faite par le processus de partitionnement est transparente et plusieurs versions du partitionnement peuvent coexister. L'adaptabilité de l'application aux variations de l'architecture cible est discutée dans la section 2.1.1.2.

2.1.1.2 Adaptation aux systèmes hétérogènes distribués

Que se soit avec les architectures de type SoC, l'utilisation de coprocesseur comme les GPUs ou l'utilisation du Cell d'IBM [44, 45], la tendance dans les architectures pour le gain de performance est l'adoption de systèmes hétérogènes à mémoire distribuée et/ou sans cohérence.

Dans ces cas là, la connaissance fine de l'utilisation de la mémoire est essentielle pour obtenir une partition performante car il faut minimiser les communications en entrée et sortie de chaque codelet, synchroniser le programme principal avec les codelets et parfois aussi les codelets entre eux, générer des tests efficaces pour la spéculation...

Par ailleurs, il est nécessaire de produire une version de codelet adaptée à chaque processeur ou coprocesseur susceptible d'exécuter le codelet. En effet, le système étant hétérogène, les *ISA* de chaque ressource le sont probablement. De plus, chacun ayant son propre espace d'adressage et sa propre hiérarchie mémoire, la répartition, la disposition et donc l'alignement des données, diffèrent d'une version à l'autre du codelet.

Une compilation de type JIT spécialisée et optimisée qui ne soit pas trop pénalisante en terme de performance n'existe pas encore à ce jour [5]. Cependant l'approche [85] présente des perspectives intéressantes, l'accélération sur les GPU étant suffisante pour masquer les surcoûts. Quoiqu'il en soit, le résultat reste plus efficace sans le surcoût du profilage logiciel à la volée. Par ailleurs, les codelets dans cette approche sont aussi

précompilés statiquement selon le procédé de RapidMind [60].

C'est dans la phase de compilation que sont formées les partitions et générées toutes les versions potentiellement exécutées des codelets. Le partitionnement, l'optimisation des codelets et leur spécialisation s'appuient alors sur des données issues de l'analyse statique et/ou de profil d'exécution.

Une fois la partition formée, il faut mettre en place une politique d'ordonnancement des codelets sur les différentes ressources. Soit on produit une affectation statique fixe où la ressource est un préalable à l'exécution ; soit on construit une fonction d'ordonnancement dont les paramètres sont les données permettant de choisir la version adaptée du codelet à l'exécution courante. Ces paramètres sont de deux natures :

- intrinsèque de l'exécution courante du programme : l'historique de branchement, l'alignement des données, les alias, la taille des données constatée, etc ;
- intrinsèque de l'environnement d'exécution : les ressources présentes/absentes lors de l'exécution, les asymétries constatées de performance entre les processeurs (*load balancing*), les affinités d'affectation permettant la réutilisation des données, congestion de la mémoire, des bus, etc ;

Cet ordonnancement est un problème extrêmement complexe dans les systèmes multiprocesseurs exécutant un ou plusieurs programmes partitionnés simultanément. La dissymétrie naturelle des systèmes hétérogènes et la non cohérence de la mémoire y apportent un grand nombre de contraintes supplémentaires. Ce problème d'un grand intérêt fait partie des perspectives d'utilisations d'Astex mais n'a pas été traité au cours de cette thèse.

2.1.2 La solution hybride

Dans cette partie je démontre pourquoi et par quel moyen l'ajout de données issues de *profils d'exécution* dans le processus de partitionnement d'application apporte une réponse efficace aux limites et contraintes établies dans la section 2.1.1.

L'hybridation apparaît dans tous les niveaux du processus de partitionnement. Les trois sections suivantes de cette partie expliquent pour chaque étape du processus pourquoi l'apport de données spéculatives permet d'obtenir une partition efficace du code. Elles portent respectivement sur la détection des codelets candidats, l'analyse des données accédées et enfin la génération et spécialisation du code.

2.1.2.1 La détection des codelets

Comme il a déjà été dit dans la section 1.4.1, la détection des codelets consiste dans un premier temps en la détection des portions de codes les plus denses en calcul dans l'application. Il s'agit principalement de boucles, implicites ou explicites, dont la fréquence des blocs est élevée et/ou les instructions exécutées sont coûteuses. On parle de points chauds ou *hotspots*.

Comparer une approche statique considérant toutes les boucles et une approche basée sur les données de *profils d'exécution* est fort utile pour deux raisons :

- Supprimer les faux *hotspots* de la liste des candidats au partitionnement. En effet certaines boucles retenues par le jeu des incertitudes des approches statiques ont des fréquences faibles ou une densité de calcul inadaptée à l'usage des accélérateurs.
- Détecter tous les *hotspots*, même ceux des boucles implicites ou des sections aux structures de contrôle complexes, difficiles à détecter pour les approches statiques.

Cependant, comme il a été fait mention dans la section 1.4.1.1, le résultat d'une approche basée sur les profils d'exécution dépend de la nature et la qualité des tests en entrée.

Se basant encore une fois sur les profils d'exécution, il est aussi possible d'affiner les *hotspots*, *i.e.* points chauds, en ne gardant en leur sein que les chemins les plus usités, on parle alors de *hotpaths*, *i.e.* chemins chauds. Certaines architectures de coprocesseurs étant très sensibles au contrôle au sein des noyaux de calcul [54], simplifier le chemin à implanter est une optimisation parfois nécessaire pour obtenir les performances attendues. Par ailleurs, réduire le nombre de chemin réduit la taille du code. Il s'agit là encore d'un facteur important de performance dans les accélérateurs.

Enfin, les fréquences relatives de chaque chemin seront une donnée fort appréciable pour l'optimisation des codelets, en particulier au niveau des communications comme il est expliqué dans le chapitre 3.

2.1.2.2 L'analyse des données

Afin de pouvoir extraire et éventuellement paralléliser les codelets, il est nécessaire de connaître un certain nombre de paramètres sur l'utilisation des données :

- les variables et zones mémoire accédées : ceci permet d'identifier les paramètres du codelet, les communications à générer en entrée et sortie, de déterminer une partie des dépendances de données ;
- La taille des données : il faut s'assurer que le processeur chargé de l'exécution sera en mesure de stocker toute les données nécessaires aux calculs ainsi que ses résultats ;
- les alias : dans le cadre d'exécution parallèle, les alias peuvent cacher des dépendances à l'analyse statique ; on est alors obligé de surestimer les alias possibles entre variables pour s'assurer de la validité du code parallèle produit.

Ces données, et d'autres encore comme l'alignement des données en mémoire ou la valeur des bornes de boucle, permettent par ailleurs d'optimiser et de spécialiser les codelets formant la partition. Cette utilisation des informations sur les données est introduite dans la section 2.1.2.3.

Une analyse statique fine du code est très complexe et limitée, donc limitante pour la détection et la génération de partition d'une application. En effet, un grand nombre d'informations sur les données comme les alias, les dépendances, les tailles, l'alignement ne peuvent pas systématiquement être inférées statiquement et peuvent dépendre de données dynamiques.

L'utilisation de profil d'exécution comme complément à l'analyse statique permet alors de remplacer les informations manquantes par une version spéculative basée sur un/des cas réels d'exécution. Encore une fois, si la qualité des profils est bonne, la qualité

des informations spéculées l'est aussi. Un fois les données estimées par spéculation, les limites de l'analyse statique tombent.

Certaines approches [77] génèrent de la spéculation au hasard et/ou systématique et utilisent l'évaluation dynamique de tests générés pour valider et/ou compléter leur génération statique. En utilisant les données issues d'un profil d'exécution, les optimisations sont mieux ciblées, la spéculation plus fine et plus sûre.

2.1.2.3 La génération et spécialisation des codelets

Une fois les codelets candidats détectés et caractérisés, on peut les générer. À ce niveau du processus de partitionnement, l'hybridation, donc l'ajout de données spéculatives, est une aide, mais aussi une contrainte.

Dans la section précédente, on a vu que l'usage de données issues de profil d'exécution permettait le calcul des sections parallèles et des communications. Cependant, étant le fruit d'un nombre fini d'exécutions, les informations introduites dans le processus sont spéculatives. Il faut générer les tests qui vont s'assurer de la validité de la spéculation et générer une version sûre du code à exécuter en cas d'erreur. Là encore, le fait de s'être basé sur un profil dans l'étape de compilation pour la spéculation est un avantage par rapport aux spéculations purement statiques ou dynamiques. En effet, les tests et leurs optimisations peuvent être calculés et générés avant l'exécution et surtout réduits au minimum.

La spéculation lors de la formation des codelets permet par ailleurs d'en générer des versions spécialisées. Ces spécialisations ont quatre sources.

- L'architecture : il est généré au moins une version par coprocesseur où il est intéressant d'exécuter la codelet. Ceci peut-être déterminé statiquement par un modèle de coût et éventuellement vérifié dynamiquement pour supprimer les versions non efficaces.
- Les *hotpaths* : se basant sur les informations issues de la détection des codelets candidats, il est possible d'utiliser les *hotpaths* plutôt que les *hotspots* pour la génération de codelet. Il faudra alors générer des tests s'assurant que le chemin suivi par l'exécution est bien celui spéculé.
- Les caractéristiques des données : il est possible de spécialiser les codelets en fonction de différents paramètres sur les données, en particulier la taille et l'alignement. Par ailleurs, les informations sur les dépendances de donnée et les alias conditionnent la parallélisation dans le codelet mais aussi des codelets entre eux.
- La valeur des données : Connaître par exemple le pas d'une itération et/ou les bornes des boucles formant le codelet permet de générer des versions optimisées du codelet en fonction des différentes valeurs de ses paramètres.

Une fois les différentes versions des codelets générées, il faut inclure à l'environnement d'exécution un mécanisme d'ordonnancement en charge de la sélection du codelet à exécuter. Ce problème complexe n'est pas abordé dans mon travail de thèse. Actuellement les codelets générés utilisent l'environnement *HMPP* afin d'être exécutés sur les unités distantes. Seule une version de codelet par architecture est générée et l'affectation des ressources se fait dans un ordre prédéterminé par l'utilisateur. L'allocation

dynamique automatisée codelets/ressources dans un environnement multiprocesseur et multitâche est une prochaine étape importante dans l'utilisation automatisée des nouvelles architectures hétérogènes.

2.2 Modélisation du partitionnement

Afin de pouvoir décrire le partitionnement d'un programme en codelets spéculatifs, il est nécessaire de se pourvoir d'un modèle de formalisation des différents éléments qui caractérisent une partition. Ce dernier doit décrire le code à extraire, son contexte ainsi que son "profil spéculatif".

Après la description du modèle pour la représentation de la mémoire et des accès de l'application dans ASTEX, cette section décrit le modèle de formalisation des codelets spéculatifs.

Enfin, je donne une définition formelle d'une partition dans le cadre de l'approche mise au point dans ma thèse.

2.2.1 Modélisation de l'utilisation de la mémoire

Cette partie présente la formalisation des éléments caractéristiques nécessaires à la construction et la représentation des codelets dans notre projet.

La figure 2.3 présente une vue synthétique du schéma d'exécution d'un codelet sur un coprocesseur avec sa propre mémoire. Dans la colonne de gauche est le CFG, dans la colonne de droite la mémoire (données) représentée sur une seule pile. L'espace d'adressage du processeur principal et du coprocesseur est différent, la valeur des adresses nous est donc indifférente dans le modèle, seules comptent les adresses relatives.

Voici la liste non exhaustive des informations utiles à modéliser pour assurer le suivi correct du schéma d'exécution de la figure 2.3.

- La taille des données créées : elle permet de déterminer la taille maximum des données du programme, la taille maximum des zones à copier et enfin vérifier qu'il n'y a pas d'accès hors zone, ce qui compromettrait le modèle.
- La taille des données accédées : si cette valeur est déterminée, elle permet de ne copier qu'un sous-ensemble des zones accédées. Dans l'exemple de la figure 2.3, A et B ne sont accédés que sur leurs 50 premiers éléments, leur copie suffit.
- La traduction des pointeurs d'un espace d'adressage à un autre : pouvoir se baser sur des formules analytiques de traduction d'adresse à partir d'une adresse définie dynamiquement est nécessaire pour créer le code du codelet. Les pointeurs que ce dernier contient doivent indiquer une valeur d'adresse dans la mémoire locale. Une valeur d'adresse se rapportant à la mémoire du processeur principal n'aurait aucun sens dans le contexte d'exécution.
- Les données accédées : plusieurs langages, comme le C, autorisent l'*alias* de variable. Si la variable est la seule donnée que l'on possède pour un accès donné, toutes les zones mémoire pouvant être pointées par cette variable seront à copier. Si l'on s'abstrait des variables pour n'associer aux expressions d'accès que des zones accédées, on élimine de fait des copies et des dépendances inutiles.

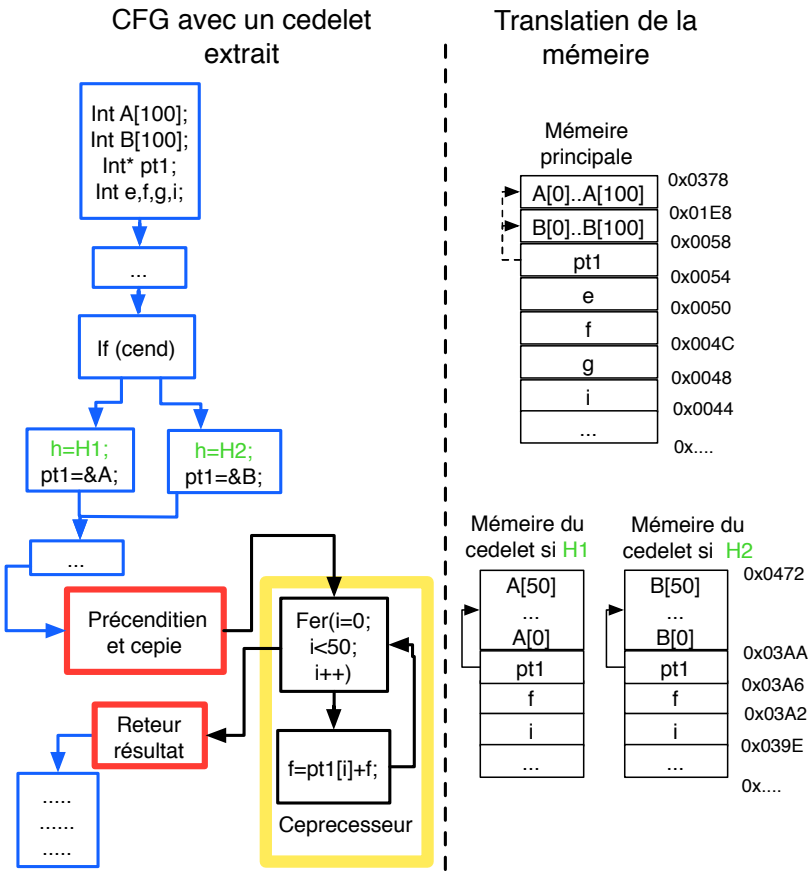

FIG. 2.3 – Dans la partie gauche du schéma figure le CFG d'une application contenant un codelet (cadre jaune). Ce dernier est exécuté sur une unité distante. La partie droite montre la ré-allocation nécessaire de la mémoire sur le coprocesseur en fonction de l'historique.

– Les conditions d'accès aux données : associer le résultat des accès mémoire au contexte permet de spécialiser leur contenu. En particulier dans l'exemple de la figure 2.3, la valeur de h au moment de l'appel au codelet permet de déterminer qui de A ou B sera à copier. Sans cette information, la copie des deux zones est nécessaire.

Dans les différentes sections de la partie 2.2.1 sont décrits les différents éléments du modèle permettant la représentation logique des éléments sus-cités. Toutes les adresses étant exprimées de manière analytique et relative, et les différents objets n'étant pas déterminés par les variables du programme (élimination des alias), il s'agit d'un modèle abstrait de l'état mémoire du programme.

2.2.1.1 Les zones mémoire

Une zone mémoire "simple", c'est-à-dire composée d'un bloc continu dans la mémoire, est constituée des éléments suivants :

– **un identifiant unique** : correspondant au point de création dans le code ;
– **une adresse de base** : notée B, elle est celle donnée dans la mémoire principale à l'allocation ; cette valeur est propre à chaque exécution ;
– **une adresse traduite** : notée B', dans la mémoire locale de chaque coprocesseur (une par coprocesseur), cette valeur est propre à chaque exécution ;
– **une taille** : qui est donnée soit lors de la déclaration (*e.g.* tableau statique) ou lors de l'initialisation pour les allocations dynamiques (*e.g.* `malloc()`).

Tous les paramètres sus-cités d'une même zone, sauf l'identifiant, peuvent prendre durant l'exécution différentes valeurs. C'est à partir des valeurs dynamiques des adresses (base et traduction) des zones que seront calculées les différentes traductions d'adresse à effectuer dans le code (m-à-j des pointeurs dans le codelet par exemple).

Voici un exemple simple en C de définitions de zone en un point de programme.

```
void * pt1;                          (1)
void * pt22;                          (2)
for(a=0;a<2;a++)
  {
    if (a)
      pt1=malloc(20);                (3)
    else
      pt1=malloc(40);                (4)
    pt2=malloc((a+1)*10);            (5)
    func(Z1,Z2);                    (6)
  }
```

Dans l'exemple précédent, au point (1) et (2), aucune zone n'est créée. Les points de création et les zones correspondantes sont respectivement les suivants, au point (3) la zone 0, au point (4) la zone 1, au point (5) la zone 2.

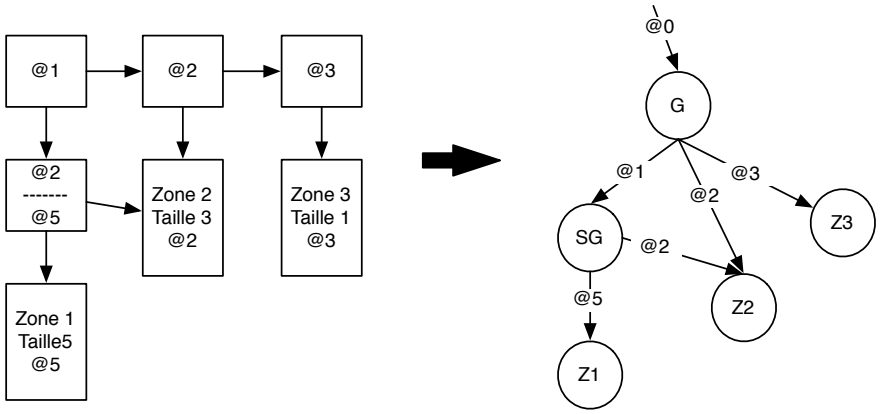

FIG. 2.4 – Schéma d'une zone mémoire composée

Lors du premier passage dans la boucle on définit la zone 0 avec l'adresse de base `pt1`, la taille 20, et comme identifiant 3. La zone 2 possède `pt2` comme adresse de base, (0+1)*10=10 comme taille et 5 comme identifiant. La fonction `func` est donc appelée avec les zones 0 et 2 en arguments.

Au second passage on définit la zone 1 d'adresse `pt1`, de taille 40 et d'identifiant 4. La zone 2 change d'adresse de base (respectivement la nouvelle valeur de `pt2`) et de taille : (1+1)*10=20. La fonction `func` est alors appelée avec les zones 1 et 2 comme arguments.

Une zone mémoire n'est pas nécessairement continue, on nomme "zone composée" une zone formée de segments non nécessairement contiguës de mémoire. La définition précédente est étendue avec les éléments suivants :

- **l'ensemble de zones simples ou composées** qui constituent la zone globale ;
- **une taille globale** qui est la somme des tailles de chaque segment.

Cette définition est récursive. Elle permet de former un graphe acyclique dont chaque nœud interne est une zone composée et chaque nœud externe une zone simple, c'est-à-dire qui ne possède pas d'arc sortant.

La figure 2.4 est un exemple de structure composée. La partie de gauche représente l'allocation physique de la mémoire, la partie de droite représente le graphe acyclique dirigé qui la formalise. Dans cet exemple, la zone 2 est accessible par deux endroits différents avec des degrés d'indirection différents. Dans ce cas, la taille physique de la mémoire allouée et le paramètre de taille globale de la zone sont différents. La taille globale de la zone suivant le graphe est :

$$taille_G = taille_{SG} + taille_{Z2} + taille_{Z3}$$

Sachant que $taille_{SG} = taille_{Z1} + taille_{Z2}$, la taille de la zone 2 est comptée deux fois dans le paramètre de taille globale alors qu'elle n'est allouée physiquement qu'une seule fois. Si SG change de forme mais pas de taille, il ne sera donc pas nécessaire de recalculer la taille globale de chaque nœud du graphe.

L'implantation actuelle ne gérant qu'un simple degré d'indirection les zones composées ne sont pas encore utilisées. L'intégration des zones composées dans le modèle actuel est là pour assurer la compatibilité du processus de partitionnement d'application avec l'éventuelle extension du support de structures de données complexes particulières (*e.g.* les listes chaînées). Dans la suite, le terme "zone" désigne des zones simples.

2.2.1.2 Les accès mémoire

Un accès mémoire, sous-entendu dans une zone mémoire, est de la forme *pointeur* + *index*. On le définit ainsi :

- **un identifiant unique** : correspondant à l'expression faisant l'accès ;
- **un numéro de zone** : correspondant à l'identifiant de la zone accédée ;
- **un numéro d'historique** : ce champs optionnel correspond à l'identifiant de l'historique courant au moment de l'accès ;
- **un offset de zone** : correspondant au déplacement du pointeur de base de l'expression dans la zone ;
- **un offset minimum** : correspondant au déplacement minimum des accès ;
- **un offset maximum** : correspondant au déplacement maximum des accès ;
- **la taille de l'élément déréférencé** : qui est la taille de l'élément accédé ;
- **un booléen R/W** : spécifiant si l'accès est en écriture ou en lecture.

L'exemple suivant illustre l'utilisation des différents champs qui composent un accès mémoire dans mon modèle.

```
int *a=z+8;
for (i=0;i<n;i=i+sizeof(int))
  {
    *(a+4+i)=0;                              (1)
  }
```

Dans le code sus-cité il y a un accès à la mémoire dans la première expression de

la ligne (1) (ie. `*(a+4+i)`). On considère que `z` est l'adresse de base de la zone. On a alors pour cet accès :

- `offset de zone` $= 8$
- `offset minimum` $= 4$
- `offset maximum` $= (4 + n) mod [sizeof(int)]$
- `taille de l'élément accédé` $= sizeof(int)$

2.2.1.3 L'historique des branchements

Bien qu'ayant été momentanément écartée de l'implantation d'ASTEX, l'utilisation d'historique dans l'optimisation de la spéculation permet en général de distinguer différentes instances d'un même événement. Une approche simple mais peu optimisée d'appréhender l'historique est la suivante : tous les branchements du code sont instrumentés et produisent une entrée unique dans une table d'historique. Cette table a une longueur finie n, paramètre utilisateur. La longueur de l'historique doit être choisie suffisamment grande pour distinguer, lorsque c'est possible, les événements. On associe à chaque historique un identifiant unique. Une fois l'exécution finie, il faut déterminer en fonction des groupes d'accès les points d'historique discriminants si il y en a, et subordonner la spéculation au test préalable de ces points discriminants du programme.

Cette optimisation est essentielle pour la spécialisation des codelets ; en effet, les expérimentations dans la partie 2.5.2.4 montrent que près de la moitié des spécialisations échoue car cette étape donne pour une même version d'un codelet des valeurs exclusives (anticorrélées) à des paramètres distincts.

Voici un exemple en C montrant l'utilisation de l'historique.

```
for(a=0;a<2;a++)
  {
    if (a)
      T1=2;              (1)
    else
      T1=4;              (2)
    func(T1);            (3)
  }
```

L'exemple précédent produit deux instances de l'événement (3) ; si l'on passe par (1) on a `func(2)`, si l'on passe par (2) on a `func(4)`. La sauvegarde d'un historique de longueur 1 au point (3) est suffisante dans ce cas pour distinguer les deux instances de `func`. En associant chaque contexte d'une instance d'un codelet à son historique en entrée, on peut donc potentiellement fournir un contexte spéculatif pertinent lors d'une nouvelle exécution.

2.2.2 Définition des partitions

Les sections précédentes ont défini les éléments nécessaires à la description du contexte mémoire de l'application à partitionner. Il reste donc à définir, pour être en accord avec la définition de la section 1.2.1, à définir un modèle pour les codelets qui puisse contenir le contexte mémoire propre au codelet et le code qu'il contient.

Une fois chaque codelet défini, la dernière étape est la construction d'un modèle contenant tout le profil spéculatif de la mémoire et tous les codelets possibles ainsi que leurs variantes. Il s'agit de l'ensemble des partitions possibles, chacune étant composée d'un sous ensemble des codelets dont la sélection est l'objet de la section 2.3.2.1.

Cette partie définit donc respectivement dans ses sous-sections les deux éléments sus-cités et conclut ainsi la définition du modèle de représentation du partitionnement.

2.2.2.1 La modélisation d'un codelet

Suite aux définitions établies dans les sections 2.2 et 2.2.1, cette partie présente la formalisation adoptée pour les codelets. Cette dernière fournit tous les éléments nécessaires à l'implantation de codelets spéculatifs. Un codelet possède l'ensemble des éléments de la liste suivantes.

- **Un identifiant unique**, fixé lors de la détection.

- **L'ensemble des points d'entrée** qui correspond aux positions possibles de l'appel du codelet dans le code principal tel que le code entre un point d'entrée et le début du code extrait n'ait aucune dépendance de données ou de contrôles avec le codelet. La version actuelle n'en choisit qu'un qui est le point situé avant la première instruction du codelet.

- **L'ensemble des points de sortie** qui correspond aux positions possibles pour le retour du codelet dans le code principal tel que le code entre le point d'entrée du codelet et le point de sortie soit indépendant sur les données et le contrôle avec le codelet. La version actuelle n'en choisit qu'un qui est le point situé après la dernière instruction du codelet.

- **L'ensemble des identifiants des instructions formant le codelet,** ces identifiants définissent le code à extraire. La spéculation portant aussi sur le contrôle, le code extrait du code principal est connexe mais non nécessairement continu.

- **L'ensemble des communications en entrée de type scalaire**, c'est-à-dire l'ensemble des variables accédées en lecture sans indirection dans le code du codelet. Elles sont classées par ensemble corrélé, si l'historique est activé.

- **L'ensemble des communications en sortie de type scalaire**, c'est-à-dire l'ensemble des variables accédées en écriture sans indirection dans le code du codelet. Elles sont classées par ensemble corrélé, si l'historique est activé.

– **L'ensemble des accès mémoires** du codelet représentés par leur identifiant unique et éventuellement dupliqués et classés suivant les valeurs corrélées.

– **L'ensemble des valeurs spéculatives** liées à chaque accès, en particulier la valeur des scalaires et l'alignement des données.

Dans l'exemple suivant en C, le codelet à extraire est formé des lignes (7) et (8).

```
int u,m,i=0;
int *pt1;
int *z1=(int *) malloc(40)
int *z2=(int *) malloc(60)
unsigned char z3[100];
init(z2);

if (sscanf("%d",z2))
   {
      history(1);                          (1)
      pt1=z1;                              (2)
      m=10;                                (3)
   }
else
   {
      history(2)                           (4)
      pt1=z2;                              (5)
      m=15;                                (6)
   };


for (;i<m;i++)                             (7)
  {
    *(pt1+4*i)=sscanf("%d",z3);            (8)
  }

  u=0;                                     (9)
  free(z3);                                (10)
```

L'ensemble des points d'entrée est réduit à (7) car les possibles instructions qui la précèdent, c'est-à-dire (6) ou (3), définissent la variable `m` qui est utilisée dans le codelet.

L'ensemble des points de sortie est constitué des instructions (8) et (9), c'est-à-dire que le retour peut s'effectuer après (8) ou (9) car `u` n'est pas une variable du codelet.

L'ensemble des communications en entrée de type scalaire est formé des variables `i` et `m`, celles-ci étant initialisées avant le codelet.

L'ensemble des accès pour l'historique {1} est composé de :

- celui effectué en lecture sur `z3` cumulant l'information d'offset des `m`= 10 tours de boucle,
- l'accès en écriture sur z1 par l'intermédiaire de `pt1` encore une fois sur les 10 tours de boucle.

Avec l'historique {2} on n'accède plus `z1` par l'intermédiaire de `pt1` mais `z2`, et l'on effectue `m`= 15 tours de boucle.

A partir de la formalisation du codelet, il est possible de calculer les portions de zones accédées lors de l'exécution déportée du codelet. Connaissant les accès mémoire dans le codelet, l'ensemble des zones en entrée avec l'historique {1} ou {2} est composé uniquement de `z3` car `z1` ou `z2` ne sont accédées qu'en lecture par l'intermédiaire de `pt1`. Une fois sortit du codelet, `z3` est invalidée avant tout accès, l'ensemble des zones en sortie est composé de `z1` pour l'historique {1} et `z2` pour l'historique {2}. La portion de la zone accédée lors de l'exécution du codelet se calcule à partir des offsets de zone, offset minimum et offset maximum qui la concernent.

2.2.2.2 Modélisation des partitions

Tous les éléments constitutifs d'une partition étant définis, il est maintenant possible de définir des partitions de l'application. Chaque partition est la donnée d'un ensemble d'identifiants de codelets tels que définis dans la section 2.2.2.1. Un codelet peut donc apparaître dans plusieurs partitions différentes.

L'ensemble des éléments définis nous donne pour le modèle de l'ensemble des partitions en sortie la liste des éléments nécessaires ou optionnels suivants :

1. **ensemble des zones mémoire** : (nécessaire) tel que défini dans la partie 2.2.1.1 ;
2. **ensemble des données d'historique** : (optionnelle) tel que défini dans la partie 2.2.1.2 ;
3. **ensemble des codelets** :(nécessaire) tel que defini dans la partie 2.2.2.1 ;
4. **ensemble des partitions** :(nécessaire) il s'agit d'une liste d'ensemble d'Id de codelet, chaque ensemble représentant une partition.

Lors de l'implantation effective du processus de partitionnement, le modèle défini dans cette section devra être intégré dans un format de fichier qui servira d'entrée avec le code C original au(x) générateur(s) de code. Des langages de modélisation hiérarchique tel que le XML sont parfaitement adaptés.

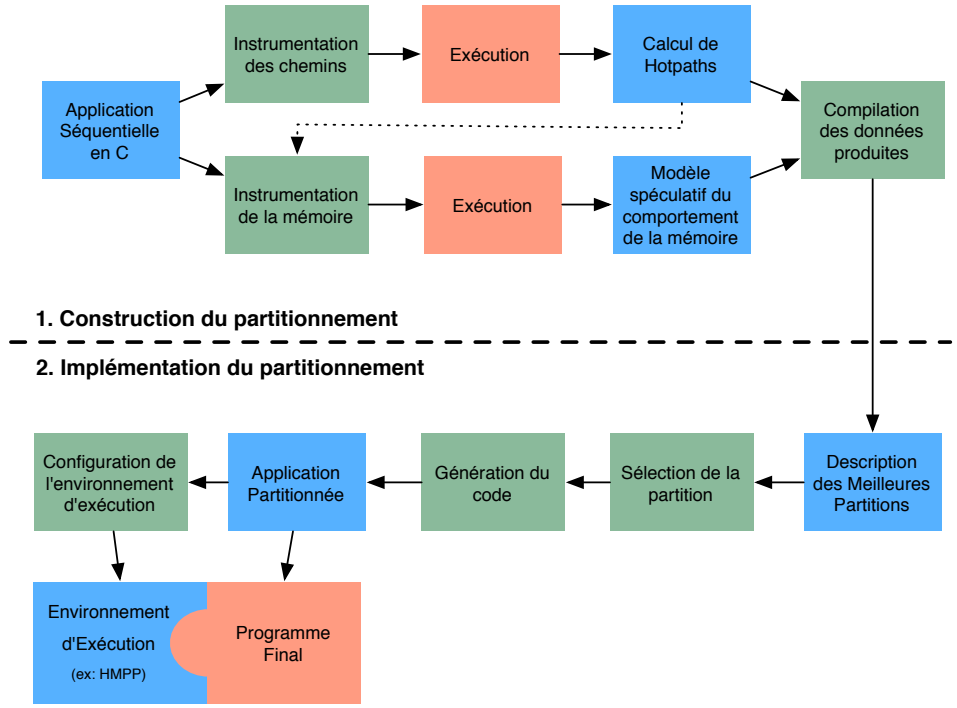

FIG. 2.5 – Schéma du processus d'extraction d'ASTEX. Il se divise en deux étapes majeures : la construction du partitionnement et son implantation.

2.3 Le processus de partitionnement d'application

Dans le processus de partitionnement mis au point pour ASTEX, il y a de nombreuses étapes et résultats intermédiaires. Comme le montre la figure 2.5 le processus peut se diviser en deux étapes majeures :

1. **la construction du partitionnement** : cette étape vise à construire et compléter le modèle de description des partitions tel que défini dans la section 2.2 ;

2. **l'implantation du partitionnement** : à partir du code original et du modèle de partitionnement, cette étape implante l'application partitionnée et fournit tous les éléments nécessaires à l'intégration de l'application finale dans son environnement d'exécution.

Cette partie du document est constituée de deux sections reprenant respectivement les deux étapes principales du partitionnement telles que sus-citées.

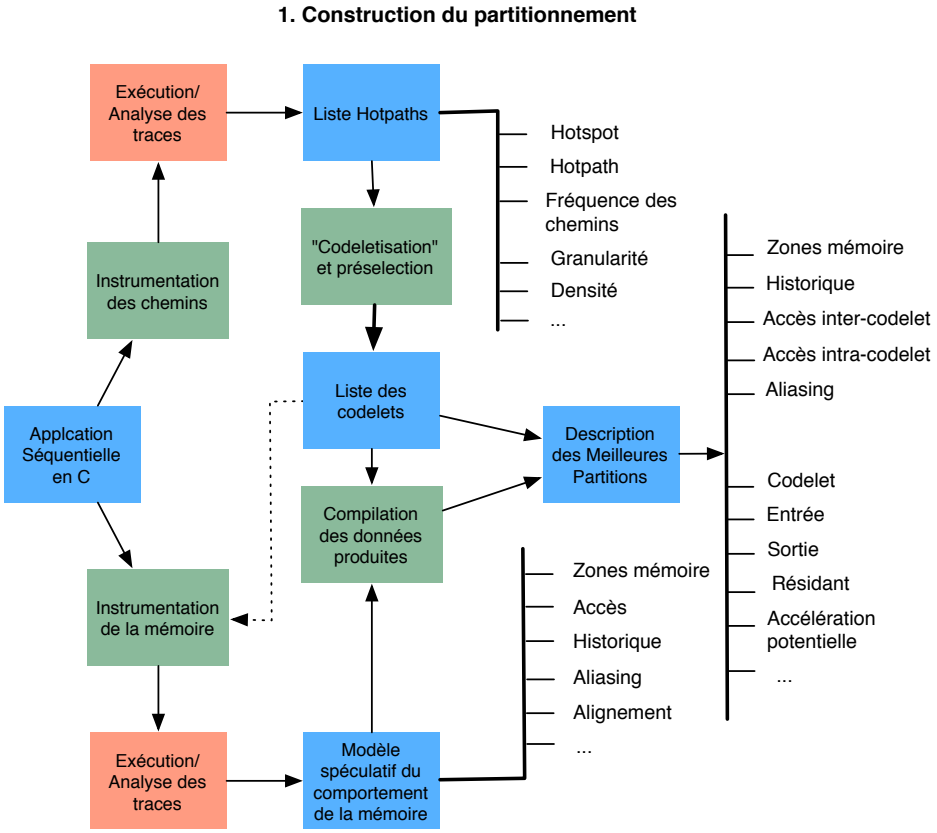

FIG. 2.6 – Schéma du processus de construction du partitionnement. Il s'agit de la partie 1 du schéma 2.5 représentant le pipeline de partitionnement dans son ensemble.

2.3.1 Construction d'une partition

La première étape du processus de partitionnement a pour mission de calculer la modélisation des partitions possibles afin de permettre leur sélection et implantation.

Le processus de construction des partitions est lui-même composé de deux étapes principales.

- **Le calcul des portions de code candidates** : Il s'agit de la partie haute de la figure 2.6. Le code est instrumenté puis exécuté afin de calculer les *hotpaths* et leurs caractéristiques. Une fois ce calcul effectué, les *hotpaths* sont implantés sous forme de codelet candidat, si ces derniers passent l'étage de présélection.

- **Le calcul de l'utilisation des données** : Il s'agit de la partie basse de la figure 2.6. Par instrumentation et exécution, on extrait des profils d'exécution les données nécessaires à la modélisation des partitions.

La flèche en pointillé reliant le calcul des *hotpaths* à l'instrumentation mémoire montre la possibilité d'exécuter en séquence les étapes de détection des *hotpaths* et l'étape d'analyse de l'utilisation de la mémoire. Ceci permet, en effet, la spécialisation de l'instrumentation et de l'analyse du comportement de la mémoire en fonction des lieux et contexte d'observation dans le code de l'application.

Une fois les *hotpaths* calculés et la mémoire caractérisée, on compile les données produites en une description des partitions contenant les codelets tels que définis dans la section 2.2.2.1.

La première section de cette partie présente l'étape de la construction du partitionnement, à savoir la détection des *hotpaths* selon des paramètres définis par l'utilisateur.

En plus du caractère filtrant des paramètres de détection, une présélection s'effectue lors de la mise en codelet sur des critères syntaxiques, sémantiques et de performances réelles tels que détaillés dans la section 2.3.1.2.

La dernière étape de construction des codelets est le calcul du modèle spéculatif du comportement mémoire.

Enfin, je décris un modèle d'évaluation des coûts permettant de guider la sélection des codelets pour l'élaboration des partitions. Il s'agit de calculer l'accélération, dite *speed-up*, minimale à partir de laquelle l'extraction du codelet est rentable en terme de performance.

2.3.1.1 La détection des chemins chauds

La détection des codelets candidats constitue la première étape du processus d'extraction de codelets.

Notre approche de détection des codelets est basée sur l'analyse des *hotspots* et *hotpaths* telle que définie dans la section 1.4.1. L'exécution du code instrumenté produit un ensemble de séquences de code (chemins du CFG) les plus usitées. La recherche des séquences dans la trace est coûteuse et exige un algorithme optimisé pour obtenir une complexité acceptable. Cet algorithme est décrit par G. Pokam dans l'article [11].

L'algorithme de G. Pokam basé sur des tableaux de suffixe, *suffix arrays*, est extrapolé à partir de l'algorithme dit KMR de Karl, Miller et Rosenberg [56] pour la recherche de séquences dans une liste d'éléments. En entrée, l'algorithme prend la trace d'une ou plusieurs exécutions du programme. Dans l'algorithme de G. Pakam, celle-ci contient la séquence des blocs de base du programme tels qu'il sont apparus dans l'exécution avec toutes leurs occurrences. Cette trace est très grande pour les applications de grande tailles ; sans optimisation la place mémoire nécessaire et le temps de traitement très long limite l'utilisation de l'algorithme. Afin de permettre son utilisation dans le cadre d'ASTEX, l'algorithme a été optimisé.

La complexité de l'algorithme est $C(T) = \mathcal{O}(T + t\log(T))$, t étant la longueur maximum des séquences détectées et T la longueur de la trace. En effet, l'initialisation du tableau de suffixes est linéaire avec la taille de la trace soit $\mathcal{O}(T)$et les étapes de réduction à effectuer pour obtenir tous les suffixes de toutes les tailles possibles de 1 à t en

$\mathcal{O}(t \log(T))$ réductions . Une fois les séquences trouvées, le calcul des différents paramètres tels que la couverture et la complexité est linéaire dans le nombre de séquences trouvées, ce qui dans la pratique est négligeable devant la taille de la trace.

Bien qu'ayant fait l'objet d'expérimentation effective et concluante de la part de G. Pokam, certaines optimisations n'ont pas été poussées jusqu'au bout. D'autres, en particulier sur la compression, sont nouvelles. L'optimisation a été faite selon quatre axes.

- Passer de tous les BBs du programme à ceux qui sont des points d'intérêt plus pertinent. Une analyse statique simple permet d'éliminer l'observation d'un grand nombre de BB sans perdre d'informations. (Un seul point au cœur des nids de boucle, pas de point en entrée des fonctions...). La longueur de la trace diminue (T diminué) et la longueur des séquences à détecter est plus faible (t diminué). Une description des optimisations efficaces pour le profilage d'application est donné dans [14].
- Compresser la trace à la volée. Dans notre version de l'algorithme, la trace est compressée dynamiquement en détectant et regroupant les séquences courtes d'une longueur maximale paramétrable. La compression de la trace à plusieurs effet : diminuer le volume de la trace (T diminué), diminuer les écritures dans le fichier de trace, précalculer les petits suffixes (t diminué) qui imposent dans l'algorithme originale un grand nombre de parcours coûteux d'une trace très longue.
- Retirer des fonctionnalités de l'algorithme original : ce dernier a une vue large du problème d'identification de séquences dans une trace. Certaines statistiques extraites sont inutiles dans notre cas spécifique (répartition de *hotpaths*, distance moyenne de réutilisation...) D'autres sont redondantes avec celles issues de l'étude des codelets.
- Réécrire une version optimisée du programme en utilisant des artefacts logiciel pour améliorer les performances (structure mémoire optimisée, localité améliorée, entrées sorties mises en tampon...).

Par ailleurs, afin de correspondre à notre objectif de partitionnement, les paramètres de détection ont été redéfinis. Ils sont les suivants :

- Complexité : on compte le nombre d'opérations dans la séquence courante (en les pondérant éventuellement selon leur nature). Cette complexité sert à quantifier la quantité de calcul contenu dans la séquence.
- Couverture : il s'agit de la proportion de la trace couverte par la séquence courante et toutes ses occurrences. Elle est donnée par la formule suivante :

$$Cov = \frac{Nb_{Bloc \in Sequence} * Nb_{Occurrence}}{Nb_{Bloc \in Trace}} \tag{2.1}$$

- Taille minimum et maximum : Il s'agit de la granularité en nombre de blocs dans la séquence

Lors de la construction des *hotpaths*, les séquences sont donc filtrées selon une complexité minimum, une couverture minimum, ainsi qu'un intervalle de granularité. Afin

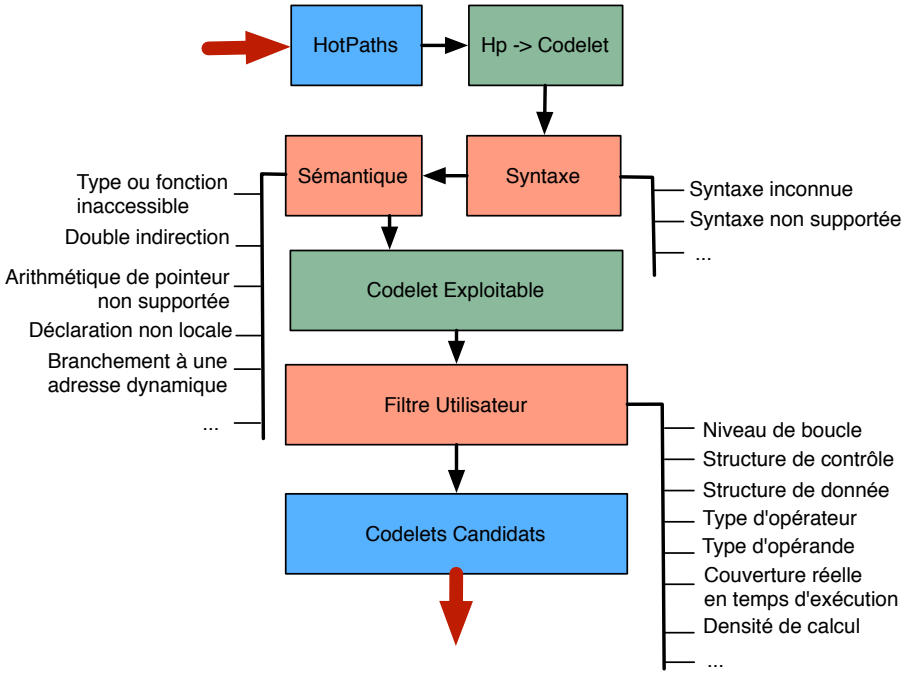

FIG. 2.7 – Processus de présélection de l'ensemble des codelets candidats au partitionnement parmi les *hotpaths* détectés.

de compléter ce premier filtre, un paramètre synthétique, qui est la combinaison des trois autres est construit. Il s'agit de la densité relative d'un *hotpath*. Elle est donnée par la formule suivante :

$$D_R = \frac{Complexite * Cov}{Nb_{Bloc \in Sequence}} = \frac{Complexite * Nb_{Occurrence}}{Nb_{Bloc \in Trace}} \tag{2.2}$$

Le critère D_R minimal a pour but d'éliminer les candidats présentant une complexité proche du minimal admis, une couverture proche du minimal et une longueur en blocs proche du maximal. En effet, peu de calcul, peu de temps d'exécution, et un code relativement long sont la définition inverse d'un *hotpath*.

La couverture étant estimée en fonction de nombre de blocs et pas de temps d'exécution, sa valeur n'est pas réelle et la couverture en temps d'exécution sera à confirmer dans l'étape de sélection des codelets candidats.

2.3.1.2 La présélection et l'implantation en codelet

En plus du caractère filtrant des paramètres de détection, une présélection s'effectue sur des critères sémantiques et fonctionnels détaillés dans la présente section.

La figure 2.7 présente une vue graphique du processus de présélection des codelets

servant à la formation des candidats pour les partitions. Une fois l'ensemble des *hotpaths* formé, ils sont mis sous forme de codelet dans le code. Durant cette phase de «codeletisation», ces derniers sont filtrés sur des critères syntaxiques et sémantiques. Enfin, sont éliminés les codelets qui n'ont pas une couverture suffisante. Ces derniers ont pu échapper aux filtres précédents eu égard à l'imprécision de la mesure basée sur la couverture en élément de la trace. D'autres filtres paramétrables par l'utilisateur sont possibles à ce niveau du processus.

L'influence de ces différents critères de sélection apparaît dans les expérimentations au niveau de l'étude des causes de rejet d'un codelet dans la section 2.5.2.3

On ne sélectionne, parmi les *hotpaths* restant dont on tente la transformation en codelet, que ceux qui ne se chevauchent pas. Ils ne peuvent donc être que inclus strictement ou disjoints. Dans le cas contraire, le premier codelet appelle le second et il y a ainsi deux points de retour pour le codelet, ce qui n'est pas compatible avec la définition donnée en 2.2.2.1. On ne peut avoir qu'un unique point de sortie qui a plusieurs positions possibles dans le code. Si deux codelets se chevauchent, on élimine celui dont la granularité est la plus faible (d'autres critères pourraient être choisis, comme par exemple la densité, la couverture, ou bien encore choisir la concaténation des deux).

Une fois les *hotpaths* mis en codelet, la première sélection se fait sur des critères syntaxiques. Il s'agit d'éliminer les codelets dont l'analyse sera impossible et que l'on ne pourra pas emmener plus loin sans amélioration du logiciel. Ces éliminations sont le fait des syntaxes inconnues du logiciel ou déclarées non-traitées.

La seconde sélection se fait sur les critères sémantiques. Les différents critères sont les suivants :

- **type ou fonction inaccessible** : un appel à des fonctions ou types situés dans des bibliothèques inaccessibles au logiciel de partitionnement rend impossible toutes analyses ;
- **double indirection** : actuellement le processus ne traite qu'une seule indirection ; il n'existe pas de forme générale pour traiter les doubles indirections ; certains cas particuliers répandus comme les listes chaînées sont une extension envisageable ;
- **arithmétique de pointeur non supportée** : aucune hypothèse n'est faite sur les différentes natures des espaces d'adressage, certaines écritures d'arithmétique de pointeur sont donc proscrites ;
- **déclaration de zones non locales** : si l'on autorise la création dans le codelet de zones de mémoire, il faut à la sortie la recréer dans la mémoire principale, traduire toutes les variables qui lui font référence dans le codelet, mais aussi toutes les variables dans le code futur de l'application séquentielle originale, ce qui peut être impossible statiquement du fait des alias ;
- **branchement à une adresse dynamique** : si la destination d'un branchement est calculée dynamiquement, il ne peut être extrait car on ne peut pas garantir que sa cible reste dans le codelet, ou bien simplement que son calcul ait encore un sens une fois le code modifié pour l'extraction.

Une fois passés les filtres statiques et sémantiques, tous les codelets restants sont implantables par notre processus de partitionnement. Toutes les autres sélections suivantes sont optionnelles et le fait de l'utilisateur ou de l'architecture cible. Sur la figure

2.7 apparaîssent plusieurs exemples de filtres utilisateurs possibles.

2.3.1.3 L'analyse de la mémoire

L'analyse de la mémoire a pour but de compléter la formalisation des codelets comme nous l'avons vu dans la section 2.2.2.1 en y ajoutant les informations concernant les accès, zones, alignements et valeurs des paramètres définis au 2.2.1.

Pour les zones, on instrumente chaque point d'allocation de mémoire dans le code. Il s'agit en C soit d'appels à la fonction `malloc` soit de déclarations de tableau. Lors d'un appel à `malloc`, l'argument passé est la taille de la zone, et la valeur retournée est son adresse. Concernant la déclaration d'un tableau, son adresse est donnée par l'opérateur `&` appliqué au symbole du tableau, et la taille est le produit des dimensions par la taille du type des éléments du tableau.

Lors d'un accès, on relève l'adresse accédée et l'adresse de base du pointeur accédé. À partir de la zone accédée on calcule l'offset de zone. À partir de l'adresse de base, on calcule l'offset minimum, l'offset maximum des accès (*i.e.* les déplacements extrêmes par rapport à l'adresse de base dans l'expression de déréférencement). Si l'historique (optionnel) ou la zone accédée change, on change d'accès. Ainsi seul le triplet {historique ; zone ; identifiant de l'expression} détermine un accès unique. Afin de maîtriser la taille de la trace et la complexité, l'information recueillie est projetée sur le modèle abstrait condensé que nous décrivons partie 2.2.

Le code à l'intérieur des codelets ne contient pas de définition ou redéfinition de zone. Le chemin étant spéculé, il n'y a plus de branchement conditionnel et donc plus de points d'historique. Ainsi tous les accès d'une même instance d'un même codelet sont considérés comme corrélés.

A l'exécution, on sait quelle portion de zone recopier grâce aux données sur les accès. Les accès dans le codelet définissent donc les communications de zones en entrée. Les accès après le codelet définissent les données en sortie.

Les accès étant calculés à partir d'un nombre fini d'exécutions, l'information sur la mémoire est spéculative. Les paramètres copiés en entrée et en sortie le sont donc aussi.

Une version beaucoup moins coûteuse et néanmoins suffisante dans de nombreux cas consiste en l'implantation sous forme de fonction pure des codelets. Le profil mémoire est alors donné par l'instrumentation des paramètres (variables) d'entrée et sortie de cette fonction. Cette version ne permet cependant pas, entre autre, la spéculation sur les accès entre deux appels codelets, on ne connait pas précisément les segments mémoire accédés au sein d'une zone. Dans le cadre de l'optimisation des communications, elle sera insuffisante.

A ce stade, la formalisation des codelets est complète. L'analyse des *hotpaths* a donné les limites du codelet et le/les éventuels chemins spéculatifs. L'analyse de la mémoire a fourni les communications spéculatives des zones et variables scalaires en entrée et en sortie.

L'usage de la spéculation des accès intra-codelet est abordé dans les sections 1.2.2.2 et 1.2.2.3, et son usage dans le cadre particulier de l'optimisation des communications est détaillé dans le chapitre 3.

2.3.1.4 Analyse de coût

La collection de données statistiques obtenues par instrumentation et exécution du code permet d'obtenir une estimation des surcoûts et des taux de bonnes et mauvaises spéculations. On ne sélectionne alors que les codelets dont les caractéristiques satisfont aux critères de rentabilité fixés par l'utilisateur.

Les surcoûts présentés dans la partie 1.3.4 ne sont pas tous pris en compte dans l'analyse du coût d'extraction d'un codelet. Les surcoûts liés à l'exploitation du parallélisme entre les codelets entre eux ou avec le programme principal sont liés à la mise au point de l'environnement d'exécution.

La granularité, notée *granularite*, est la longueur en nombre d'instructions du codelet ; on peut noter que plus le codelet est long plus les surcoûts sont absorbés ; cependant, la pénalité est potentiellement plus forte en cas de mauvaises spéculations. Si le codelet est court, les surcoûts prennent une part importante du temps total d'exécution et donc l'accélération potentielle est réduite. Le produit entre la granularité et le taux de mauvaises prédictions, noté $miss\_rate$ est donc un facteur tangible de discrimination des codelets, car il est proportionnel à la pénalité.

Comme le mettent en évidence les expérimentations de la section 2.5.2.4 et 3.5, les communications jouent un rôle prépondérant dans les exécutions sur un système à mémoire distribuée et/ou non cohérente. Le rapport entre l'estimation temporelle des communications et le temps d'exécution du codelet est un deuxième critère discriminant dans l'analyse des surcoûts.

L'objectif est de déterminer quels sont les codelets qui avec une accélération de la portion de code extraite, notée *speedUp* réalisable sur le coprocesseur, présente un gain par rapport au temps d'exécution séquentielle de cette même portion de code sur le processeur principal, noté tps_{exec}, ce qui donne l'équation (2.3) :

$$\frac{surcout + \frac{tps_{exec}}{speedUp}}{tps_{exec}} < 1 \tag{2.3}$$

Il existe deux processus de discrimination des codelets permettant leur sélection dans la partition finale : le premier par comparaison des codelets entre eux, le second par l'estimation de l'accélération nécessaire à la rentabilité de l'extraction.

Pour un codelet donné : Soit $false\_exec\_ratio$, la proportion moyenne d'instructions exécutées avant une mauvaise spéculation. La pénalité en nombre d'instructions, notée P_{inst}, est définie par l'équation (2.4) :

$$Pinst = granularite * false\_exec\_ratio * miss\_rate \tag{2.4}$$

Les instructions dans le codelet étant par nature essentiellement constituées de la répétition périodique d'une séquence (boucle), il est raisonnable de faire l'hypothèse que le temps d'exécution est uniformément reparti dans le codelet. La pénalité en temps, notée P_{tps}, sachant (2.4), correspond à l'équation (2.5) :

$$Ptps = \frac{tps_{exec} * P_{inst}}{granularite} \tag{2.5}$$

Les coûts fixes (cf. 1.3.4) étant les mêmes pour tous les codelets, ils ne sont pas significatifs pour un critère comparatif. Connaissant le coût variable de communication d'une unité de mémoire, noté tps_{1_com}, la quantité de mémoire à copier, notée $data\_size$, on peut calculer une estimation de la portion variable du coût des communications, notée TV_{com}, intervenant dans le critère de comparaison. Il est donné par l'équation (2.6) :

$$TVcom = data\_size * tps_{1\_com} \tag{2.6}$$

A des fins de simplification, on fait l'hypothèse que le coût du contrôle d'erreur est uniformément proportionnel à la longueur, c'est-à-dire que le pouvoir discriminant du surcoût dû au contrôle d'erreur se confond avec le paramètre de granularité. Le coût du contrôle d'erreur n'apparaît donc pas dans le critère comparatif. En réalité, ce coût augmente moins vite que la longueur du code car les tests peuvent être factorisés. Il s'agit donc d'une surestimation.

Avec (2.5) et (2.6) on obtient le critère de comparaison, noté CC, donné par l'équation (2.7) :

$$CC = miss\_rate * false\_exec\_ratio + \frac{data\_size * tps_{1\_com}}{tps_{exec}} \tag{2.7}$$

Suivant la nature du code, il peut être préférable de rendre la sélection plus sensible à l'un des deux paramètres et donc de pondérer l'équation (2.7) selon la méthode des barycentres avec un paramètre u libre, choisi par l'utilisateur. Par exemple, si la stabilité du chemin suivi dans le code à l'exécution est un critère déterminant sur une architecture où le retour d'erreur est difficile, on rendra plus sensible le critère au calcul du coût variable de la pénalité donné par l'équation (2.5). On obtient l'équation finale du critère de comparaison pondérée, noté PCC (2.8) :

$$PCC = u * miss\_rate * false\_exec\_ratio + (1-u)\frac{data\_size * tps_{1\_com}}{tps_{exec}} \tag{2.8}$$

Le second critère, dit de discrimination absolue, estime l'accélération sur le code extrait, notée *speedUp* nécessaire à la rentabilité de l'extraction ; ce facteur tenant compte du temps d'exécution du codelet, les coûts fixes entrent dans sa composition. Comme il a été fait référence dans la partie 1.3.4, ils sont les suivants :

– Le coût fixe d'appel du codelet comprenant la/les instructions d'appel du codelet et le coût fixe de la recopie de la mémoire en entrée.

– Le coût fixe du retour qui comprend la/les instructions de retour du codelet et le coût fixe de la recopie de la mémoire en sortie.

Avec les coûts fixes, notés CF, et les coûts variables, notés CV, donnés par (2.5), l'équation (2.6) additionnée du surcoût dû à l'instrumentation pour la détection des erreurs, la condition de rentabilité donnée par (2.11) devient :

$$\frac{CV + CF + \frac{tps_{exec}}{speedUp}}{tps_{exec}} < 1 \tag{2.9}$$

Ce qui donne dans la forme classique d'expression d'une équation de gain (2.10) :

$$speedUp > \frac{1}{1 - \frac{CV+CF}{tps_{exec}}} \tag{2.10}$$

Notons que $\frac{CV}{tps_{exec}}$ est égale à CC .

Il est possible d'ajouter d'autres critères de discrimination que celui de la performance en temps d'exécution : répartir les calculs pour diminuer l'échauffement thermique, améliorer le coût énergétique des calculs, diminuer le nombre de ressources utilisées... L'utilisation croisée de ces paramètres d'optimisation peut se faire en plaçant les différents critères dans un vecteur dont la norme est l'évaluation finale du potentiel du codelet. Il est possible encore une fois de pondérer les différents éléments de ce vecteur en fonction de l'importance que l'utilisateur donne à chacun des critères. On parle d'optimisations multicritères.

2.3.2 Implantation d'une partition

Une fois la liste des codelets bien définie par la première partie du processus de partitionnement, on arrive à l'étape de l'implantation dans l'application du partitionnement.

La figure 2.8 représente cette étape d'implantation. Elle commence par la sélection de la partition à implanter, c'est-à-dire la liste des codelets à extraire du code.

Une fois les codelets à extraire connus, on passe à la génération du code. À chaque codelet correspond une ou plusieurs fonctions indépendantes. Le choix de la version du codelet à exécuter est délégué à l'environnement d'exécution.

C'est ce dernier qui, correctement paramétré se charge de la bonne exécution du code en terme de spéculation, d'allocation de ressources, de communications, de choix de version, ... Astex doit fournir en sortie tous les éléments utiles à la configuration

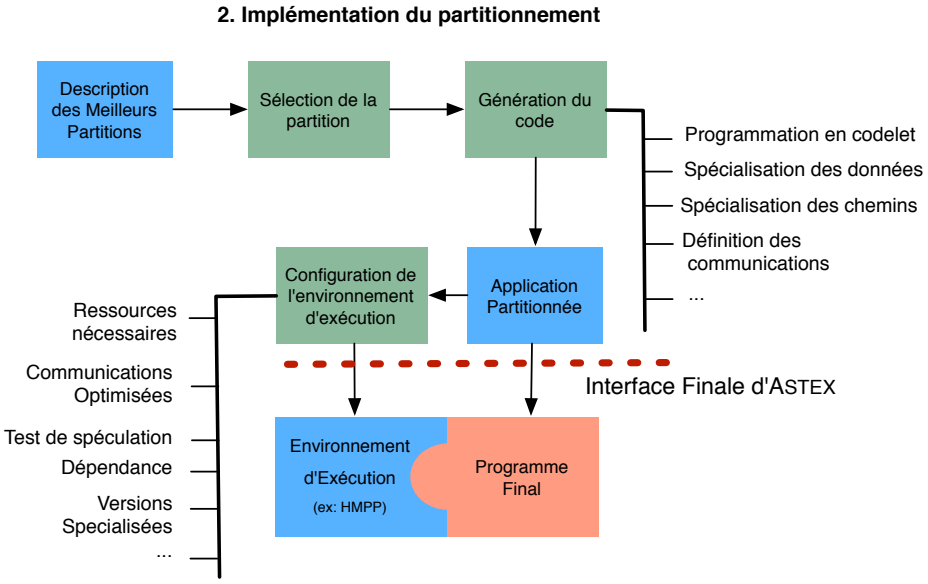

FIG. 2.8 – Schéma du processus d'implantation du partitionnement. Il s'agit de la partie 2 du schéma 2.5 représentant le processus de partitionnement dans son ensemble.

de l'environnement d'exécution en sus de l'application partitionnée dans son langage de haut niveau.

Dans cette partie est présentée, dans un premier temps, l'étape de sélection des codelets basée sur les critères définis dans la partie 2.3.1.4 mais aussi sur des mesures de l'adéquation entre les codelets et les ressources à disposition.

Dans un second temps, je présente l'interface fourni par ASTEX au générateur de code et au contrôle de l'environnement d'exécution.

2.3.2.1 Sélection de la partition

La sélection repose essentiellement sur la minimisation du rapport entre le temps d'exécution du codelet avec ses surcoûts et le temps d'exécution séquentiel avant l'extraction. Il s'agit de l'équation 2.4. On cherche à déterminer à l'aide du modèle de coût les versions des codelets qui ont le *SpeedUp* minimal le plus en rapport avec le coprocesseur visé. Lorsque ces chiffres de *SpeedUp* minimal sont trop élevés ou que les différences de performance en temps sont relativement homogènes entre les processeurs présents dans le système, seul le parallélisme peut apporter une accélération.

Le second critère exploité pour la sélection des codelets est la densité absolue (à opposer à l'équation 2.2 de la densité relative). On la définit par l'équation 2.11 :

$$D = \frac{Complexite}{nb\_inst} \tag{2.11}$$

La complexité est, comme pour les *hotpaths*, un décompte pondéré des opérations dans le codelet. Sur une exécution de taille raisonnable du codelet, il est possible de remplacer cette valeur de complexité par le temps d'exécution, ces deux valeurs étant proportionnelles. Le nombre d'instructions peut prendre plusieurs valeurs ; celles-ci devant rester en rapport avec la longueur réelle du code en instruction. En effet, choisir le nombre d'instructions dans un langage de haut niveau, ou la longueur en BB n'est pas nécessairement une bonne mesure, choisir un assembleur plutôt qu'un autre, est aussi un compromis. Dans les expérimentations de la section 2.5.2.2, pour le calcul de densité, a été choisie la longueur en assembleur x86 obtenue après compilation avec `gcc` en $-O2$. Une valeur seuil est fixée de manière à obtenir des codelets potentiellement performants sur l'architecture cible. Ce seuil est calculé, soit par expérimentation soit sur la base de modèle empirique du comportement de l'architecture cible, puis affiné.

La conjugaison de ces deux critères permet donc la sélection de la partition du programme final en fonction de/des architectures cibles. C'est à ce moment là qu'Astex spécialise sa sortie, tous les résultats précédents sont portables.

2.3.2.2 Génération du code et configuration d'exécution

La dernière étape du processus de partitionnement d'application est l'implantation de la partition choisie. Cette dernière est constitué par la génération du code partitionné du programme, et la configuration de l'environnement d'exécution. Afin de permettre de tirer le meilleur parti de tous les éléments calculés par Astex, la dernière étape du processus de partitionnement est donc celle de la présentation des résultats. Cette présentation est la donnée de deux éléments distincts tels que représentés sur la figure 2.8 :

- L'application partitionnée dans son langage original.
- La caractérisation des codelets ainsi que les différentes versions spécialisées et leurs gardes associées.

L'application partitionnée est une application dont les sources seules permettent son exécution distribuée. L'implantation des codelets a déjà été faite par l'implantation de fonction autonome et leur appel placé dans le code source original. Il s'agit de l'étape finale de la transformation des *hotpaths* en codelets, décrite dans la section 2.3.1.2. Après l'étape de sélection décrite dans la section 2.3.2.1, on désactive les codelets non sélectionnés dans la partition finale.

Avant de confier l'application au générateur de code, toutes les optimisations sur le contrôle des codelets doivent être insérées. Il s'agit en particulier de l'optimisation des communications.

Lorsque l'environnement matériel et logiciel le permet il est possible de mettre en place un préchargement des données en entrée et sortie des codelets. Communiquer

de manière asynchrone permet de cacher les latences en exécutant, en parallèle de ces transmissions, une partie de l'application principale. Placer les communications au plus tôt pour utiliser le préchargement en parallèle de l'application principale, éviter les redondances, supprimer les mises à jour inutiles de la mémoire distante, sont autant d'optimisation dont l'expression doit être confiée au générateur de code ; leur élaboration est l'objet du chapitre 3.

Cette étape est aussi celle de la génération de toutes les variantes de chaque codelet et du code par défaut, c'est-à-dire le code qui est exécuté en cas d'erreur dans l'exécution. Les variantes de codelets générés correspondent aux différentes versions possibles d'un codelet en fonction :

- Du statut spéculatif du chemin ou des données. L'objectif est de simplifier le contrôle, supprimer les problèmes d'alias et réduire les communications. Ceci facilite la phase de compilation et d'optimisation en limitant les contraintes inhérentes à la nature du code. L'intersection entre le code des codelets et le code exécutable sur les architectures hôtes s'en trouve élargie.

- De la spécialisation. Sont récoltées par Astex les différentes valeurs spéculatives des paramètres scalaires des codelets. C'est de ces derniers dont dépendent le plus souvent les bornes de boucles, les tailles de vecteurs, les chemins suivis. Certaines valeurs permettent aussi l'optimisation de calcul. C'est le cas, par exemple, sur certaines architectures du remplacement de la multiplication ou division par des puissances de 2 par un décalage. Enfin, la spécialisation peut être fonction de l'alignement des données, valeur capturée par Astex.

- De l'architecture cible, c'est-à-dire des ressources disponibles à l'exécution. Les règles d'optimisations comme le choix de la forme du code sont fonction des capacités de l'hôte. Taille des mémoires, nombre de registres, taille et précision des opérateurs flottants, présence d'unité vectorielle, gestion des structures de contrôles, sont des exemples de facteurs significatifs dans le choix d'une implantation d'un codelet.

À l'exécution, en fonction des ressources disponibles, l'environnement d'exécution se charge du choix de la version du codelet à exécuter en fonction du statut spéculatif courant du code et/ou des ressources disponibles.

Une fois l'application générée, il faut paramétrer l'environnement d'exécution pour l'implantation courante des codelets. Cette étape est la partie dépendante de l'architecture du processus de partitionnement. La configuration de l'environnement d'exécution comprend : l'allocation des tâches aux différentes ressources, le choix des versions spécialisées que l'on veut exécuter, l'implantation et optimisation des tests de spéculation

et de spécialisation, l'activation ou non d'optimisation comme celle des communications en fonction des capacités du système.

L'allocation des tâches aux différentes ressources peut-être effectuée de manière statique ou dynamique. Il s'agit d'un problème d'ordonnancement. Il faut optimiser la répartition des tâches en fonction des versions de codelet possibles et des ressources disponibles, mais aussi favoriser d'éventuelles localités ou affinités de données de manière à optimiser l'utilisation de l'architecture et éviter les congestions sur les unités de calcul, les mémoires et les médiums de communication. Fournir un nombre raisonnable de variantes de codelet avec des caractéristiques différentes, augmente l'adaptabilité de l'application ainsi que sa souplesse d'utilisation par le système.

2.4 Implantation du processus de partitionnement

Alors que les sections 2.3 et 2.2 décrivent le modèle théorique qu'Astex tend à implanter, cette partie décrit les aspects pratiques de l'implantation et de l'utilisation d'Astex.

La figure 2.9 représente l'utilisation des briques logicielles dans l'implantation du processus de partitionnement. En plus d'Astex, on y retrouve une technologie support : *HMPP*.

Les différentes étapes d'Astex sont décrites dans la section 2.3. Il utilise un analyseur/générateur de langage C pour produire ses résultats et *HMPP* pour les exprimer.

Le code généré par Astex utilise les directives du langage *HMPP* pour exprimer le partitionnement dans l'application originale. Une fois préprocessé, et avec les générateurs fournis par *HMPP*, le code de l'application finale partitionnée et les éléments d'environnement d'exécution sont générés. Chaque codelet est compilé selon le/les processeurs auxquels il est destiné. On obtient alors l'application finale.

Cette partie décrit la mise en oeuvre du logicielle ainsi que le format des résultats obtenus pour les deux grandes étapes du partitionnement d'application : la construction des partitions puis leur implantation effective dans l'application finale.

Les sections sont organisées dans l'ordre de construction d'un codelet. Un seul et même exemple sera déroulé de bout en bout pour illustrer l'action des différentes étapes de construction. Il s'agit d'un codelet issu de l'application h264 de compression/décompression vidéo.

2.4.1 La construction du code partitionné

Dans cette partie est présentée l'implantation effective des codelets dans le code de l'application originale.

Cette implantation commence par le passage du chemin chaud au codelet et l'insertion de son appel dans le code.

La seconde étape est celle de la spécialisation. Grâce aux valeurs spéculatives collectées par Astex, le codelet est spécialisé et sa garde en appel modifiée pour vérifier à l'exécution la validité de cette transformation.

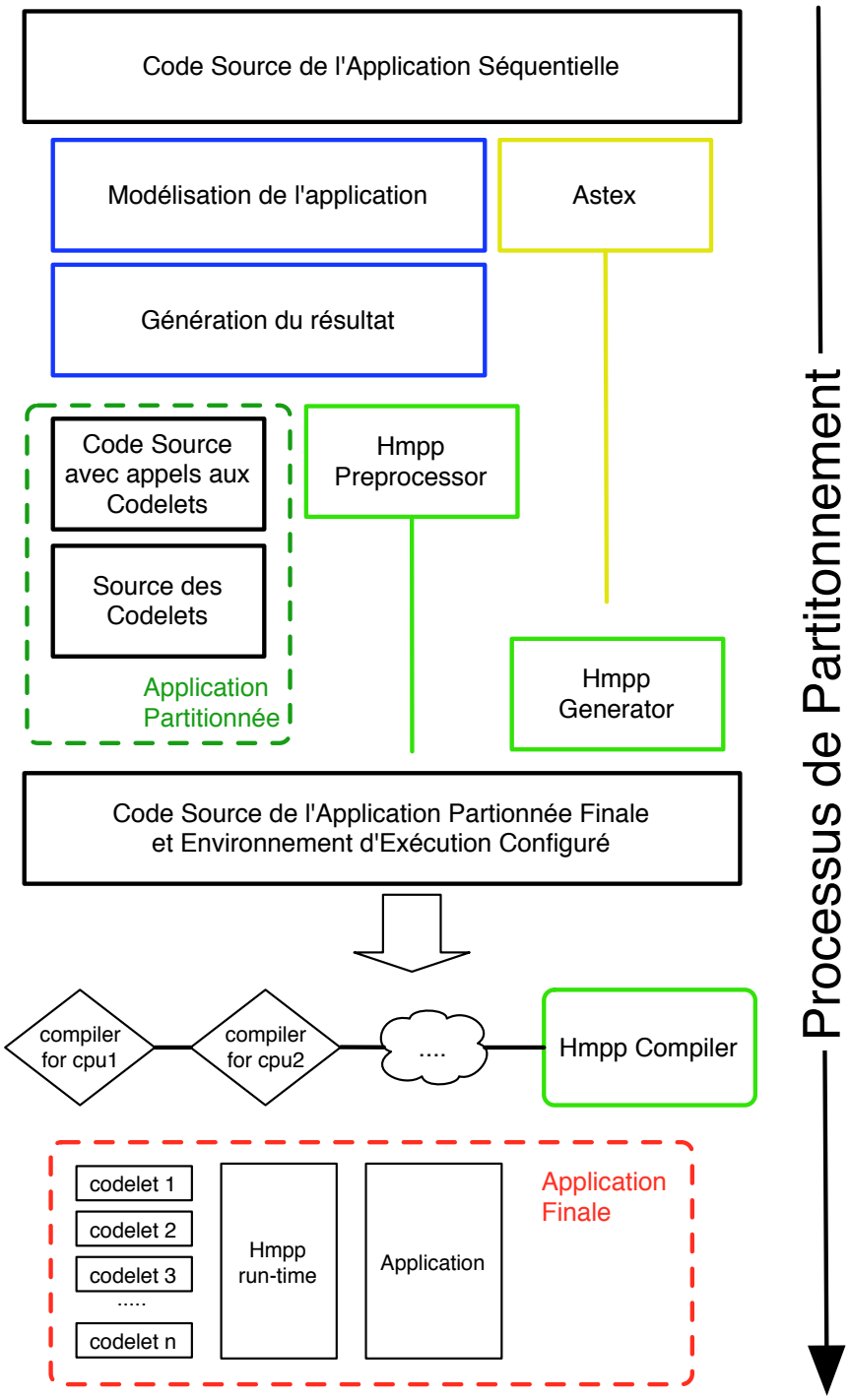

FIG. 2.9 – Schéma de la répartition des briques logicielles dans l'implantation du pipeline de partitionnement.

Enfin, la dernière étape de l'implantation est celle des optimisations. Dans l'état actuel du développement d'ASTEX, il s'agit en particulier du préchargement de données pour l'optimisation des communications.

Cette partie présente, dans l'ordre, les trois étapes de l'implantation telles que suscitées.

2.4.1.1 Du chemin chaud au codelet

La portion de code suivante est un *hotpath*, HP, de l'application H264 dans sa forme brute (langage C) :

```
{
   i_sum += abs(pix1[x] - pix2[x]) ;
}
```

Un HP, conformément à sa définition pré-citée, est l'ensemble d'un ou plusieurs *BB* connexes dont les occurrences représentent une partie significative de l'exécution. Dans le cas présent, il s'agit d'un bloc contenant l'incrément d'une variable `i_sim` par la valeur absolue de la différence entre deux valeurs de tableau indexées par la variable `x` : `pix1[x]` et `pix2[x]`.

Afin de donner au codelet une granularité suffisante, le *hotpath* est étendu par les structures de contrôle englobantes qui sont dans un voisinage immédiat, c'est-à-dire les structures qui n'englobent que des structures elles-mêmes englobantes du HP. La notion de voisinage est par défaut la fonction englobante, mais elle peut aussi être une donnée de l'utilisateur sous la forme d'un niveau d'imbrication. Le code suivant est la version étendue de notre HP exemple :

```
for (y = 0 ; y < 8 ; y++)
{
   for (x = 0 ; x < 8 ; x++)
   {
      i_sum += abs(pix1[x] - pix2[x]) ;
   }
   pix1 += i_stride_pix1 ;
   pix2 += i_stride_pix2 ;
}
```

Il s'agit d'un nid de boucle, *i.e.* plusieurs boucles strictement imbriquées. La première boucle fait varier de 0 à 7 la valeur de l'index `x`. La seconde fait aussi huit tours et met à

jour les pointeurs `pix1` et `pix2` qui désignent des sous tableaux d'une zone mère espacée de pas prédéfinis `i_stride_pix1` et `i_stride_pix2`. À des fins de spécialisation, il serait intéressant de connaître ces valeurs ; on pourrait alors, par exemple, calculer en parallèle toutes les valeurs de `i_sum`.

Une fois étendu, le HP est mis sous forme de codelet compatible avec le format de*HMPP*. Chaque codelet est précédé d'une garde. Elle est ici toujours vraie car aucune spéculation à ce stade n'a été implantée. Ce ne sera plus le cas après la spécialisation. Le code du codelet et sa garde sont les suivants :

```
inline int Astex_guard__1(uint8_t *pix1, int i_stride_pix1,
uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
{
   return 1 ;
}


void Astex_codelet__1(uint8_t *pix1, int i_stride_pix1,
uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
{
   int y ;
   int x ;
   int i_sum = __Astex_addr__i_sum[0] ;
   Astex_thread_begin : {
      for (y = 0 ; y < 8 ; y++)
      {
         for (x = 0 ; x < 8 ; x++)
         {
            i_sum += abs(pix1[x] - pix2[x]) ;
         }
      pix1 += i_stride_pix1 ;
      pix2 += i_stride_pix2 ;
      }
   }
   Astex_thread_end : ;
   __Astex_addr__i_sum[0] = i_sum ;
}
```

Le codelet est une fonction pure englobant le HP étendu. Les paramètres locaux sont définis comme des variables locales. Les paramètres réentrants comme `i_sum`, c'est-à-dire dont la valeur de sortie de l'appel k au codelet est la valeur d'entrée à l'appel k+1, sont définis de manière locale. Une variable sert à initialiser la valeur en entrée et récupérer le résultat. L'utilisation et l'allocation de cette variable sont gérées par l'environnement d'exécution*HMPP* qui se charge de transférer la valeur d'un appel à l'autre tout en gérant son partage éventuel. C'est ainsi qu'est traité le cas des données

dites résidentes, c'est-à-dire pouvant rester allouées d'un appel à l'autre dans la mémoire locale du coprocesseur. Toutes les autres valeurs sont des paramètres de la fonction. Les variables `pix1` et `pix2` étant des pointeurs, ils ne sont pas considérés comme réentrant. En effet, leur valeur est fonction du contexte de l'exécution. Plusieurs appels au codelet sur des coprocesseurs différents donnent à `pix1` et `pix2` des valeurs différentes.

2.4.1.2 Implantation de la spécialisation

Se basant sur la spéculation des valeurs scalaires des paramètres en entrée du codelet, il est possible de spécialiser ce dernier. Par ailleurs, l'ensemble des paramètres est classé en trois catégories : entrée, entrée-sortie, sortie. Ceci afin de générer les communications correspondantes. Voici le code spécialisé de notre codelet exemple :

```
#pragma hmpp Astex_codelet__1 codelet &
#pragma hmpp Astex_codelet__1 , args[__Astex_addr__i_sum].io=inout &
#pragma hmpp Astex_codelet__1 , args[pix2].io=in &
#pragma hmpp Astex_codelet__1 , args[pix1].io=in &
#pragma hmpp Astex_codelet__1 , target=C &
#pragma hmpp Astex_codelet__1 , version=1.4.0

void Astex_codelet__1(uint8_t *pix1, int i_stride_pix1,
uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
{
   int y;
   int x;
   int i_sum = __Astex_addr__i_sum[0];
   Astex_thread_begin : {
      for (y = 0; y < 8; y++)
      {
         for (x = 0; x < 8; x++)
         {
            i_sum += abs(pix1[x] - pix2[x]);
         }
         pix1 += 16;
         pix2 += i_stride_pix2;
      }
   }
   Astex_thread_end :;
   __Astex_addr__i_sum[0] = i_sum;
}
```

Avant le début de la fonction sont exprimés sous forme de *pragmaHMPP* le numéro de version du codelet et les propriétés des entrées/ sorties du codelet.

Dans le code du codelet spécialisé, la valeur de `i_stride_pix1` a été remplacée par la valeur scalaire 16 ; en effet, se référant au tableau d'histogramme 2.16, on remarque que 85% des appels environ utilisent la valeur 16. On pourrait à juste titre se demander pourquoi la variable `i_stride_pix2` n'est-elle pas spécialisée. Cependant, encore une fois d'après l'histogramme 2.16, la valeur de `i_stride_pix2` est jugée trop volatile pour être spéculée.

Une fois le code spécialisé, il est nécessaire de s'assurer à l'exécution que les valeurs spéculées pour la génération du codelet sont bien les valeurs utilisées lors de l'exécution. Il faut donc insérer des tests de spéculation avant l'appel du codelet : il s'agit de la garde du codelet. En voici le code :

```
inline int Astex_guard__1(uint8_t *pix1, int i_stride_pix1,
uint8_t *pix2, int i_stride_pix2, int __Astex_addr__i_sum[1])
{
   const void *__Astex_region_addr__pix1 ;
   unsigned long int __Astex_region_size__pix1 ;
   __Astex_get_region(
      &__Astex_region_addr__pix1,
      &__Astex_region_size__pix1, pix1) ;
   return i_stride_pix1 == 16 &
      (__Astex_region_addr__pix1 != (const void *) (0) &
      (pix2 < __Astex_region_addr__pix1 |
      pix2 >= __Astex_region_addr__pix1 + __Astex_region_size__pix1)) ;
   }
```

Cette garde reprend les mêmes paramètres que le codelet et effectue les tests correspondants aux spéculations employées. Dans l'exemple précédent, deux tests sont effectués. Le premier vérifie que `i_stride_pix1` est bien égale à 16, la valeur choisie pour la spécialisation. Le deuxième test vérifie qu'il n'existe pas d'alias entre la zone désignée par l'intervalle `[pix1 ; pix1+Astex_region_size__pix1]` et `pix2`.

Une fois le code spécialisé, le codelet, sa garde, et un code par défaut sont inséré dans l'application. Si la garde est vraie on exécute le codelet spécialisé, sinon, on exécute le cas par défaut. Le code suivant est celui faisant appel au codelet exemple tiré de H264. Le code par défaut est le code original de l'application choisie pour être extrait en codelet. La section de code générée prend place dans l'application originale à l'endroit du code extrait.

```
if Astex_guard__1(pix1, i_stride_pix1, pix2,
i_stride_pix2, __Astex_addr__i_sum[1])
{
```

```
   #pragma hmpp astex_codelet__1 callsite
   void Astex_codelet__1(pix1, i_stride_pix1, pix2,
   i_stride_pix2, __Astex_addr__i_sum[1])
}
else
{
      for (y = 0 ; y < 8 ; y++)
      {
         for (x = 0 ; x < 8 ; x++)
         {
            i_sum += abs(pix1[x] - pix2[x]) ;
         }
         pix1 += i_stride_pix1 ;
         pix2 += i_stride_pix2 ;
      }
   }
}
```

2.4.1.3 Le préchargement des données

Dans l'utilisation de coprocesseur matériel, le principal surcoût identifié vient des communications. Pour contourner ce goulet d'étranglement, quand les architectures le permettent, il est possible de recouvrir les latences de communications en usant de préchargements asynchrones (*advanced load*) et d'enregistrements différés des résultats (*delegated store*). L'exemple de code choisi, pour illustrer l'usage des directives de préchargements et d'enregistrements différés, est extrait de la documentation de *HMPP* [42] produite par Caps-entreprise. Il s'agit du code suivant :

```
#pragma hmpp matvec advancedload, args[inm], args[inm].size={n,m},
....
while (...){
   #pragma hmpp matvec callsite, args[inm].advancedload=true, args[inm].size={n,m},asynch
   matvec(n, m, (inc+(k*n)), inm, (outv+(k*m))) ;
   #pragma hmpp matvec synchronize
   #pragma hmpp matvec delegatedstore, args[outv]
   if (...) {
   for (i=0 ; i<m ; i++) {
      inm[...] = 0.1 ;
   } /* endfor */
   #pragma hmpp matvec advancedload, args[inm]
}/* endif */
}/* endwhile */
```

La première directive `advancedload` précharge la matrice inm de dimension $n * m$ avant 'appel au codelet dans la directive `callsite`. La directive `synchronize` spécifie que l'application doit attendre la fin de l'exécution du codelet avant de continuer. Le programme principal continue son exécution jusqu'à cette directive. La directive `delegatedstore` est la barrière de synchronisation pour la variable *outv*. Le rapatriement du résultat du calcul dans *outv* doit être fini pour que l'exécution continue.

L'insertion automatique et optimisée des directives de préchargement est l'objet du chapitre 3.

2.4.2 Format des statistiques en sortie d'ASTEX

La finalité d'ASTEX est de fournir un outil paramétrable de partitionnement automatique d'application. Ce dernier guide l'utilisateur dans le partitionnement d'une application et son implantation. Le programmeur peut modifier les paramètres d'entrée, le code en sortie, les codelets sélectionnés... Il doit donc être en mesure d'évaluer la qualité de la partition et des codelets. Il est, par conséquent, indispensable d'avoir une sortie lisible et parfaitement exploitable par un utilisateur humain.

Cette partie présente le format de sortie des statistiques pour l'utilisateur et des pistes pour son interprétation à travers un exemple, H264, d'où sont repris tous les graphiques. Cette application est présentée plus en détail dans la section 2.5. La structure de cette page HTML est donnée dans la figure 2.10. Le document se décompose en deux parties principales. Les données globales, au niveau application, et les données locales, au niveau codelet. L'étude de ces deux ensembles est respectivement l'objet des sections 2.4.2.1 et 2.4.2.2. Il y a un seul ensemble de statistiques globales et plusieurs ensembles de statistiques locales, un par codelet (de 1 à N dans la figure).

2.4.2.1 Les statistiques au niveau application

La figure 2.11 présente l'histogramme des couvertures en bloc des *hotpaths* trouvés. Chaque barre représente la couverture d'un *hotpath* dans la trace d'exécution du programme instrumenté. Il s'agit d'une couverture en bloc et non en temps d'exécution. Dans l'exemple (H264), on a trouvé 13*hotspots* couvrant environ 75% de la trace.

Le temps total initial est celui du programme original sans instrumentation. Il est calculé avec l'outil de mesure de temps Gprof sous Unix.

La figure 2.12 est l'histogramme de la couverture réelle en temps d'exécution des codelets. Les codelets 4 et 6 ont été supprimés car ils se chevauchaient avec d'autres, ils portent la mention *Duplicated*. Si le rejet avait été d'une autre nature ils auraient porté la mention *Rejected*.

Après la transformation en codelet, ces derniers couvrent environ 52% de l'application en 10 codelets ; le codelet 13 étant en mesure réelle inférieur au seuil des 1% de

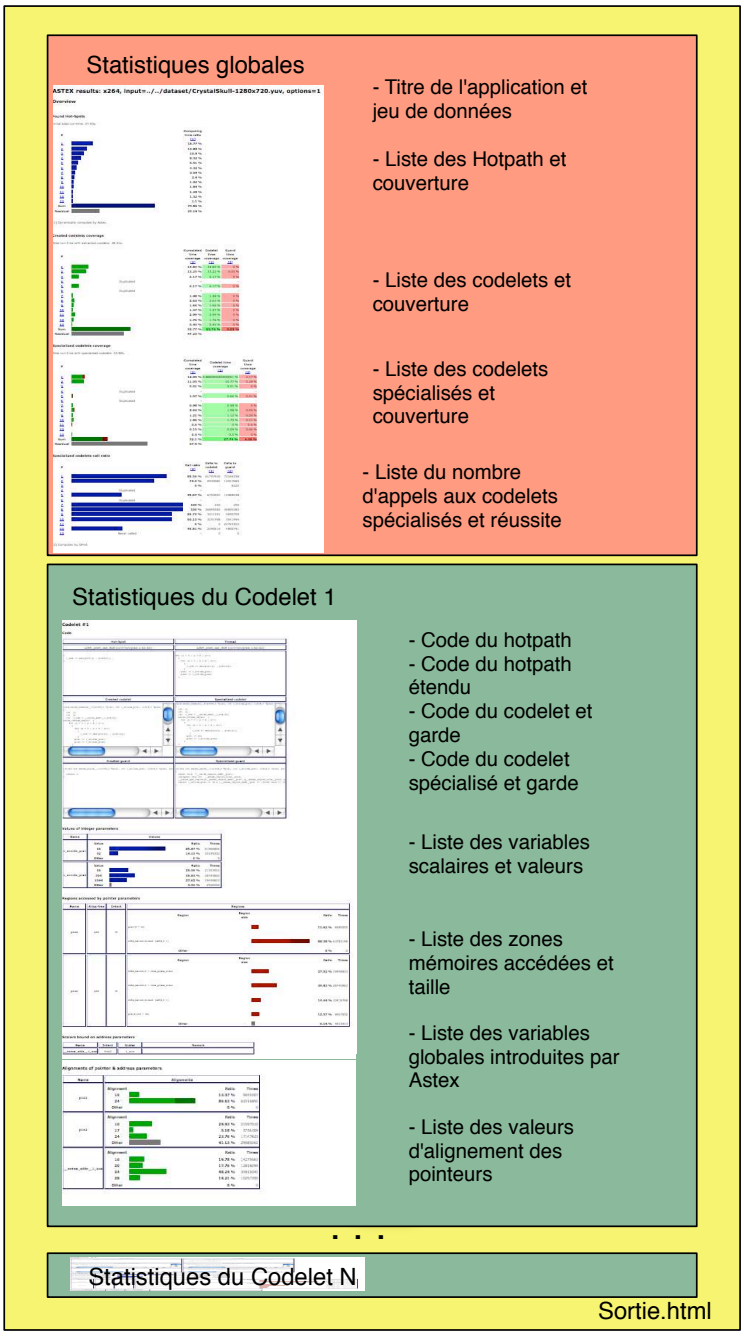

FIG. 2.10 – Vue annotée du fichier HTML en sortie d'ASTEX présentant les différentes statistiques à l'utilisateur.

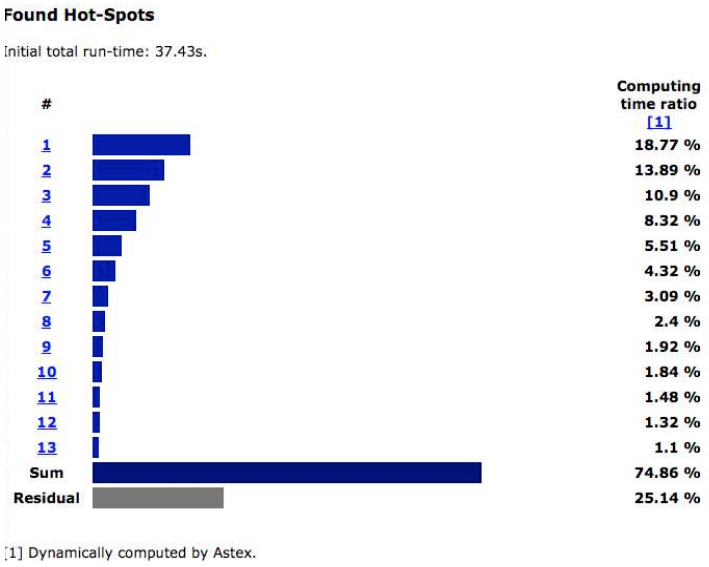

FIG. 2.11 – Histogramme des *hotpaths* trouvés par Astex. Chaque barre représente la couverture en bloc d'un *hotpath* dans la trace.

couverture fixé pour la détection. Le temps d'exécution de l'application avec codelet est de 38,31s, ce qui, mis en rapport avec les 37,43 s de l'application originale, nous donne un surcoût en temps d'exécution pour l'appel des codelets depuis le code principal de 3,6%.

Une fois les codelets créés et les valeurs spéculatives récoltées par ASTEX, on tente pour chaque codelet une spécialisation. À terme, ce sont plusieurs versions spécialisées par codelet qu'il faut générer. La figure 2.13 présente l'histogramme de la couverture en temps d'exécution des versions spécialisées des codelets, la couverture cumulée en temps, la couverture en temps du codelet seul et enfin de sa garde seule.

Le graphique de la figure 2.13 se rapporte toujours à notre même exemple, H264. Cependant, à part pour les codelets 1, 2, 7, 8, 9 et 10, le code spécialisé ne semble pas fonctionner, la couverture étant nulle ou quasi nulle.

L'explication partielle est donnée dans la figure 2.14. Cette dernière figure présente les proportions d'appel au codelet spécialisé par rapport à la garde, c'est-à-dire la réussite de la spécialisation. Le nombre d'appels à la garde est logiquement équivalent au nombre d'appels au codelet original. Seule la spécialisation des codelets 1, 2, 7, 8, 9 et 10 peut être considérée comme réussie. Ce qui confirme l'observation faite sur la figure 2.13. Les causes d'échec seront abordées dans la partie 2.5.

Dans l'exemple présent, d'après la figure 2.13, le temps passé dans la garde oscille entre 2 et 30% environ du temps passé dans le codelet. Ce temps est fonction de la

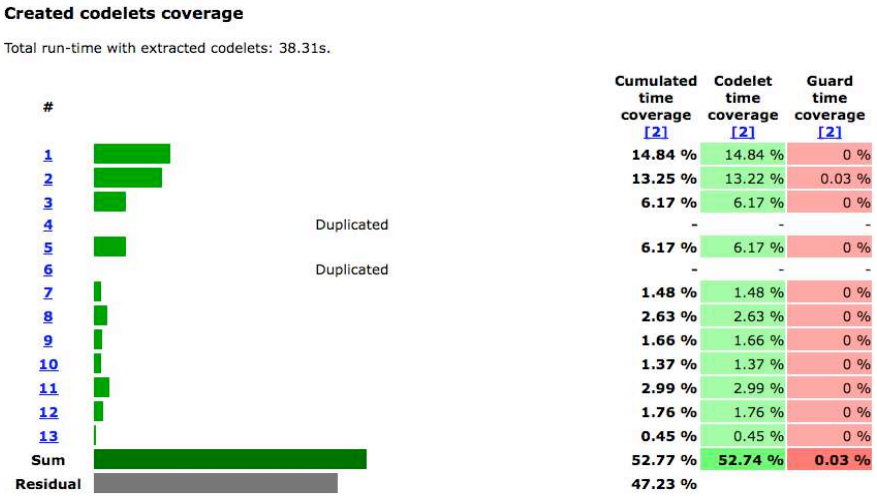

FIG. 2.12 – Histogramme présentant la couverture en temps d'exécution de chaque codelet candidat.

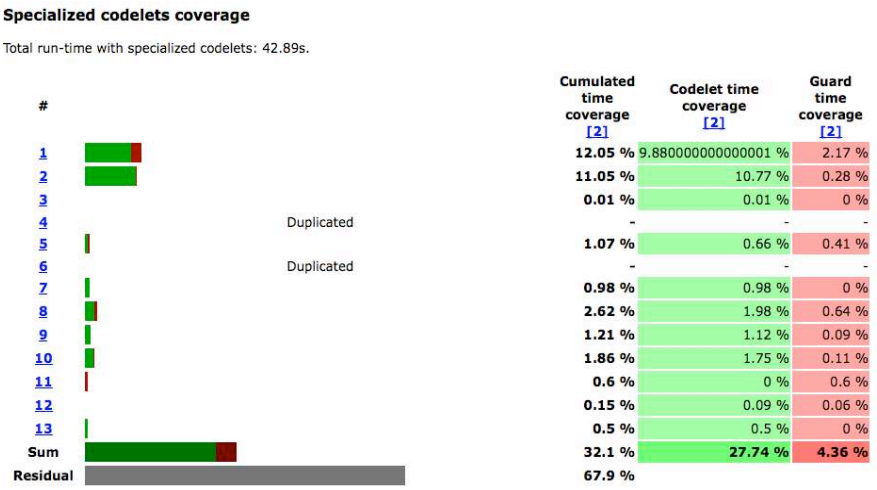

FIG. 2.13 – Histogramme présentant la couverture en temps d'exécution des versions spécialisées des codelets. Il se décompose en la couverture du codelet lui-même et celle de sa garde.

granularité du codelet. Les codelets 1 et 2, par exemple, ont une couverture équivalente en temps d'exécution, mais comme le montre la figure 2.14, le nombre d'appels pour une même couverture en temps du codelet 1 est six fois plus élevé. C'est donc que la granularité du codelet 1 est environ six fois plus faible. Il est donc logique que les gardes du codelet 1 et 2 de complexité équivalente représentent respectivement 20 et 2% de la

couverture en temps d'exécution des codelets seuls.

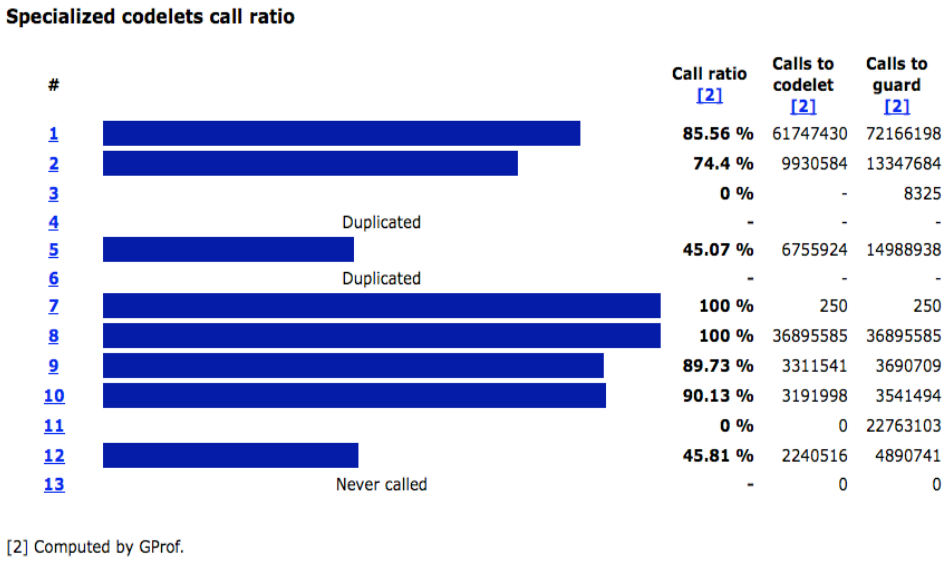

FIG. 2.14 – Graphique présentant pour chaque codelet spécialisé le ratio entre les appels à la garde et les appels du codelet, ce qui implicitement implique que la garde est vraie.

2.4.2.2 Les statistiques au niveau codelet

La figure 2.15 présente pour chaque paramètre du codelet les zones qui lui ont été allouées au cours de l'exécution, le nombre de fois qu'elle l'ont été, la taille de ces zones et si ces paramètres ont des alias. Dans la figure, la valeur de la taille est absente. Ce chiffre est devenu momentanément indisponible suite à des changements dans l'implantation de son calcul. L'histogramme des tailles est cependant bien une sortie possible d'ASTEX.

Le tableau de la figure 2.15 concerne toujours l'exemple issu de H264. On y retrouve donc les variables `pix1` et `pix2`. La première colonne nous dit qu'il n'y a pas d'alias entre les paramètres, la seconde qu'il sont tous les deux des entrées. La troisième colonne donne les possibles expressions de création des zones mémoire allouées à la variable, la quatrième leur taille, et enfin la dernière, la proportion des appels en fonction de chaque valeur de zone mémoire. On y remarque en particulier, que `pix1` est appelé plus de 88% des fois avec la zone correspondant à l'expression : `x264_malloc(sizeof(x264_t))`. Cette information est intéressante, par exemple, pour la génération des communications ; en effet, en suivant dans l'application l'utilisation de la zone de mémoire désignée, on peut placer le préchargement au plus tôt. La variable `pix2` est beaucoup moins stable ; on retrouve cette instabilité dans les alignements ; la spéculation dans ce cas devient impossible et `pix2` est exclu de toute spécialisation.

La figure 2.16 présente les valeurs observées pour les paramètres scalaires et leur

Regions accessed by pointer parameters

Name	Alias-free	Intent	Regions			
			Region	Region size	Ratio	Times
pix1	yes	in	pix1[9 * 32]		11.62 %	8385000
			x264_malloc(sizeof (x264_t))		88.38 %	63781198
			Other	-	0 %	0
			Region	Region size	Ratio	Times
pix2	yes	in	x264_malloc(4 * luma_plane_size)		27.02 %	19499823
			x264_malloc(4 * luma_plane_size)		39.83 %	28745862
			x264_malloc(sizeof (x264_t))		14.44 %	10418768
			pix[2][32 * 18]		12.57 %	9067832
			Other	-	6.14 %	4433913

FIG. 2.15 – Histogramme de l'allocation des zones mémoire aux paramètres de type pointeur du codelet.

proportion d'apparition dans les appels au codelet. On peut constater que la variable `i_stride_pix1` présente le même profil que le paramètre `pix1`, et la variable `i_stride_pix2` la même instabilité que le paramètre `pix2`. Pour la spécialisation, on associera donc la valeur 16 de `i_stride_pix1` à l'utilisation la plus courante de `pix1`. C'est ce qui a été fait, comme présenté dans le code de la section 2.4.1.2.

Values of integer parameters

Name	Values		
	Value	Ratio	Times
i_stride_pix1	16	85.87 %	61966866
	32	14.13 %	10199332
	Other	0 %	0
	Value	Ratio	Times
i_stride_pix2	16	29.59 %	21353953
	704	39.83 %	28745862
	1344	27.02 %	19499823
	Other	3.56 %	2566560

FIG. 2.16 – Ce tableau d'histogramme représente les valeurs prises par les variables scalaires et leur proportion dans les appels ; par exemple, `i_stride_pix1` est appelé 85,87% des fois avec la valeur 16.

La figure 2.17 présente la valeur d'alignement pour chaque paramètre de type poin-

teur. Ceci permet de déterminer si les valeurs sont compatibles avec le fonctionnement de l'architecture cible et éventuellement de corriger l'alignement en changeant le *padding*.

Alignments of pointer & address parameters

Name	Alignments		
pix1	**Alignment**	**Ratio**	**Times**
	16	**13.37 %**	9649307
	24	**86.63 %**	62516891
	Other	**0 %**	0
pix2	**Alignment**	**Ratio**	**Times**
	16	**29.93 %**	21597010
	17	**5.18 %**	3736304
	24	**23.76 %**	17147823
	Other	**41.13 %**	29685061
__astex_addr__i_sum	**Alignment**	**Ratio**	**Times**
	16	**19.78 %**	14275663
	20	**17.76 %**	12818099
	24	**48.24 %**	34815041
	28	**14.21 %**	10257395
	Other	**0 %**	0

FIG. 2.17 – Tableau présentant pour chaque valeur scalaire les valeurs possibles d'alignement et leur proportion dans les appels au codelet.

2.4.3 La génération de codelet en programme indépendant

A des fin d'expérimentations, pour pouvoir commodément exercer un noyau de calcul sur une architecture donnée, il est utile de fournir à l'utilisateur le moyen d'exécuter le noyau seul. Cela implique non seulement d'extraire le code et de lui générer un code écrin pour l'exécuter, mais aussi de construire à partir du jeu de tests d'entrée les jeux de données spécifiques pour les paramètres du codelet.

La première section de cette partie présente la capture et la construction de tels jeux de données.

La seconde section explique brièvement les éléments permettant l'implantation autonome du codelet.

2.4.3.1 La génération automatique des données de test

On capture pour une execution donnée la valeur et la forme des parametres du codelet. On les encode alors dans un fichier binaire.

Pour chaque codelet est créé un ou plusieurs jeux de tests permettant d'exercer le codelet seul.

On capture aussi les données produites par le codelet. Ceci permet de comparer les résultats originaux à ceux des futures versions spécialisées, optimisées ou exécutées

sur des architectures différentes. Ce point de repère est critique pour la validation de l'implantation déportée des codelets.

À terme, seront générés, à partir des histogrammes de spécialisation des variables scalaires et de la tailles des données, des familles de jeux de test pour chaque spécialisation possible.

2.4.3.2 La génération du programme appelant

Ce point particulier ne présente pas de grande difficulté technique à partir du moment où les paramètres du codelet sont clairement identifiés et leur valeur capturée.

Le programme généré se compose d'une fonction `main` appelant le codelet qui commence par ouvrir et transcrire le fichier binaire en données réelles du programme. Chaque donnée est associée à un ou plusieurs (alias) paramètres du codelet.

L'appel au codelet peut alors avoir lieu. Les valeurs en sortie sont elles aussi capturées et peuvent être comparées aux valeurs originales connues (pour l'évaluation d'erreur de calcul par exemple).

2.5 Expérimentation

Le développement d'Astex a pris quatre années. Depuis le développement du prototype de recherche, de mi-2005 à fin 2006, à la dernière version industrialisée de mars 2009, les fonctionnalités implantées ou non et les méthodes de mises en oeuvre ont beaucoup varié.

Pour cette partie sur les expérimentations, les résultats présentés sont issus de la dernière version d'Astex. Pour certaines fonctionnalités non ré-implantées, les anciens résultats sont rappelés.

L'objectif premier est de démontrer la pertinence et l'efficacité de l'approche spéculative choisie pour Astex. On cherche à apporter des réponses aux questions suivantes :

- Quel type de codelet trouve-t-on ?
- Quelle proportion des codelets est implantable ? Pourquoi des rejets ?
- Couvrent-ils un temps significatif de l'application ?
- Les statistiques extraites des profils sont-elles pertinentes ?
- Où et comment intervient la spéculation ?

Dans un second temps est esquissée une méthode d'appariement codelet/architecture. Il s'agit de déterminer l'intersection entre les besoins en ressources des codelets trouvés et les capacités du système hôte, mais aussi comment maximiser cette intersection.

Cette partie présente dans ses premières sections les critères d'évaluation choisis ainsi que les programmes de tests utilisés pour l'élaboration des résultats.

Une fois les critères déterminés, sont présentés et analysés les résultats afin d'apporter des réponses aussi argumentées que possible aux questions pré-citées dans cette introduction. Cette section finit sur l'utilisation des résultats d'expérimentation dans l'étude

encore très prospective de la mise au point d'un outil d'appariement codelet/architecture basé sur ASTEX.

2.5.1 Protocole expérimental

La validation de l'approche spéculative utilisée dans ASTEX se déroule en deux phases :

– La validation des différentes étapes du processus d'extraction
– La validation de la pertinence des résultats en sortie du processus

La première de ces phases s'effectue tout naturellement par l'observation du déroulement du processus de partitionnement. Il s'agit de trouver les critères de mesure et d'analyse permettant de qualifier chaque étape par rapport au modèle initial. On veut s'assurer de la conformité du processus au modèle.

Une fois la qualité de la mise en oeuvre éprouvée, l'expérimentation doit montrer la pertinence du résultat en sortie d'ASTEX. Ces résultats doivent permettre, outre le partitionnement, un calcul efficace de la spéculation et des surcoûts liés à l'implantation sur un système hétérogène distribué.

Cette première partie des expérimentations présente donc, dans un premier temps, les critères choisis pour les expérimentations et leur motivation.

Dans un second temps est développé le choix argumenté des applications tests (*Benchmarks*) utilisées.

2.5.1.1 Les critères d'évaluation

Les critères d'évaluation à mettre en oeuvre dépendent de ce que l'on veut mesurer. Les mesures devront répondre aux trois questions suivantes :

– Quels codelets candidats détecte-t-on ?
– Quelles caractérisations des codelets obtient-on ?
– Le partitionnement est-il applicable et dans quelle condition ?

Cette section développe en trois paragraphes les mesures à effectuer pour appuyer la réponse aux trois questions pré-citées.

Quels codelets candidats détecte-t-on ?

La première étape du processus de partitionnement est la détection des codelets candidats ; son fonctionnement est critique. Il s'agit de détecter les «bons» codelets.

Pour chaque application, il s'agit de déterminer combien de codelets ont été détectés. Ce résultat est le fruit d'un simple décompte des codelets créés.

Quelle est leur couverture en temps d'exécution ? Pour cette mesure, les programmes de tests utilisés étant d'une taille suffisante, on utilisera l'outil `gprof` sous Unix.

On s'intéresse aussi à leur granularité. Pour mesurer cette dernière, on se basera sur la longueur du code assembleur obtenu pour la compilation de la fonction codelet seule avec la commande : `gcc -O2 -c codelet_n.o`, puis la commande `objdump -d |wc -l` qui désassemble le code et compte le nombre d'instructions.

Enfin, il faut éprouver la robustesse de la détection à la variation des jeux de tests en entrée et aux paramètres du programme. On fait pour cela sur différentes applications varier les données en entrée, en taille et en nature, puis indépendamment, varier les paramètres.

Quelle caractérisation des codelets obtient-on ?

Afin de choisir et construire la partition, on extrait des profils d'exécution des applications les informations spéculatives pouvant guider le processus. Certains codelets s'avèrent impossibles à caractériser.

Lorsqu'il n'est pas possible de traiter les codelets, il est intéressant de savoir quelle proportion a été rejetée, mais aussi de savoir pourquoi. Cela permettra de savoir qui du modèle théorique ou du logiciel est limité et dans quelle mesure.

Quand on peut extraire les informations, on peut alors créer un certain nombre de données statistiques nous donnant le profil des codelets retenus. Il s'agit du volume de données en entrée, du volume de données en sortie, de la stabilité des valeurs des paramètres d'entrée. Ces données sont essentielles pour la spéculation et donc la spécialisation. Elles sont par ailleurs utiles à l'évaluation du modèle de coût utilisé par la suite.

Pour la spécialisation et la génération des tests de spéculation, on analyse combien de codelets sont spécialisés, avec quelles informations spéculatives, quelle est la complexité des tests induits. Enfin, une fois spécialisés, l'exécution permet de déterminer combien de codelets ont réussi. Il s'agit alors de déterminer les causes d'échec pour les autres.

Le partitionnement est-il applicable et dans quelles conditions ?

Après toutes les étapes du processus, les codelets restants sont en nombre restreint. On s'attache donc dans l'expérimentation à savoir combien et quelle proportion peuvent être implantés à la fin du processus. S'ils sont implantés, l'accélération minimale à atteindre pour le coprocesseur qui les exécute est-il réaliste ? Cette évaluation se fera à partir du modèle de coût établi dans la section 2.3.1.4.

2.5.1.2 Le choix des programmes de tests

Le logiciel Astex ayant atteint aujourd'hui une certaine maturité, il n'existe pas de contre-indication raisonnable dans le choix des applications à lui soumettre. Le choix

des applications tests se fait donc librement et dans l'intérêt des expérimentations. Les trois principales familles d'applications forment la liste suivante.

- Les `NAS parallel benchmarks` : il s'agit d'applications de calcul scientifique et/ou multimedia sous forme d'algorithmes parallélisables en C et openMP. Leur forme où le noyau de calcul parallèle est connu et bien isolé dans l'application permet de vérifier la conformité des résultats produits par Astex. Retrouver automatiquement ces sections et les caractériser est un critère minimal de fonctionnement du logiciel.
- `H264` : il s'agit d'une application d'encodage/décodage de video. Ce code multimédia, au contrôle complexe et fortement paramétrable, est connu pour sa difficulté de traitement. Obtenir de bons résultats sur un code réputé difficile pour les programmeurs est l'un des résultats attendus pour Astex.
- Les `SPEC2006` : complexes et variés ; l'étude des SPEC permet d'avoir un panorama statistiquement réaliste du fonctionnement d'Astex. Cette famille d'applications permet d'obtenir des codelets de natures très différentes. Il s'agit d'un test en largeur de la robustesse du logiciel mais aussi d'obtenir un panel de codelets de natures différentes afin d'étudier leurs applications.

Enfin le traitement de ces trois familles d'applications est complété par l'utilisation d'application choisie de manière unitaire. Il s'agit de retrouver les résultats de fonctionnalités non ré-implantées dans la dernière version d'Astex ; certaines applications ne passant pas encore dans les versions précédentes du logiciel.

2.5.2 Qualité des solutions, efficacité et robustesse

Dans cette partie sont présentés les résultats issus de l'exécution d'Astex sur les différentes applications tests.

Tout d'abord, afin d'être pratiquable par l'utilisateur final avec un coût raisonnable, Astex doit être fiable et d'une complexité maîtrisée en temps d'exécution et de taille de mémoire utilisée. Ce point est traité dans la première section de cette partie.

Dans la seconde section, je présente le profil des codelets candidats détectés. Ils sont l'unité de base du partitionnement. Afin de valider l'approche par *hotpath* choisi pour la détection des codelets, il faut comprendre par l'expérimentation ce qui est détecté, pourquoi, et s'assurer de la pertinence des résultats.

Une fois détectés, un certain nombre de candidats sont rejetés. Obtenir le nombre de rejets et identifier leurs causes permet de mesurer l'efficacité et la qualité des méthodes d'implantation choisies pour Astex, mais aussi de mesurer les limites du modèle mis en œuvre. Ces résultats d'expérimentation sont présentés dans la troisième section.

2.5.2.1 Performance de l'implantation

Dans les trois paragraphes suivants sont traitées les performances d'Astex en terme de temps de calcul, de mémoire nécessaire, et enfin de la nature des applications traitées avec succès.

Temps de calcul d'une partition

La figure 2.18 présente la correspondance entre le temps initial d'exécution d'une application et le temps de calcul de sa partition par ASTEX. Chaque point correspond à une application des trois familles d'applications tests choisies et pour différentes tailles de problème d'entrée. Tous les points se placent sur sur une droite de coefficient directeur 7,33. Ce qui signifie que le temps de partitionnement théorique est environ de 7,33 fois le temps d'exécution initial de l'application.

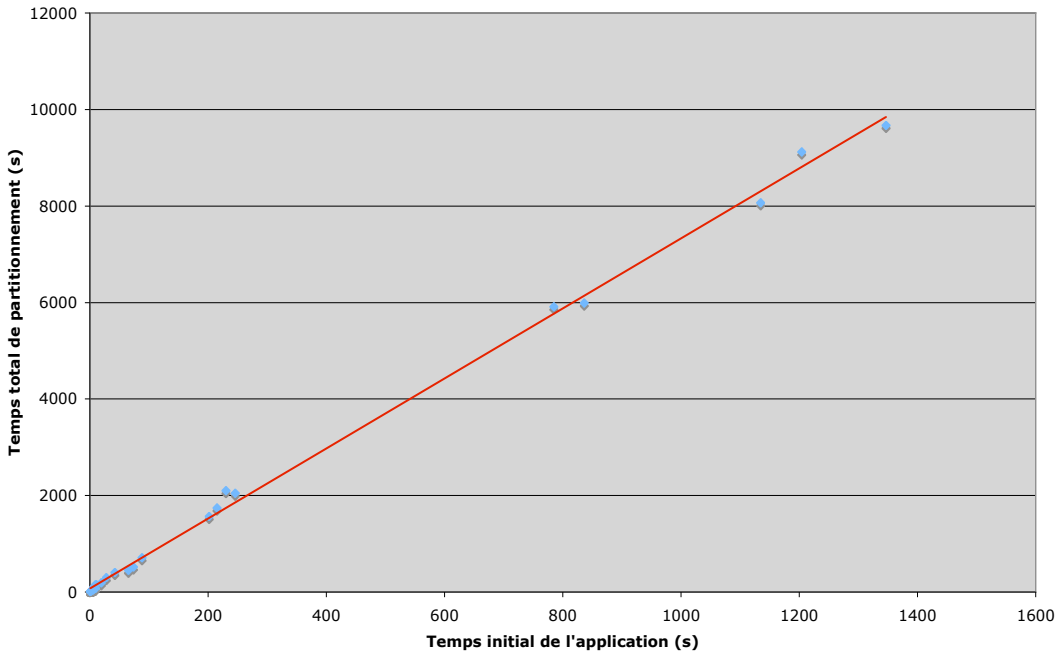

FIG. 2.18 – Sur ce graphique est représenté le temps d'exécution du processus d'ASTEX en fonction du temps initial de l'application.

Le surcoût du calcul d'une partition avec ASTEX semble donc être proportionnel au temps initial d'exécution de l'application originale.

Dans sa version actuelle, ASTEX effectue 7 exécutions de l'application modifiée. Ceci explique un facteur 7 minimal. Seules les versions instrumentées pour le calcul des *hotpaths* et pour le calcul de la spéculation mémoire ont un surcoût significatif en temps d'exécution. Pour le calcul des *hotpaths*, ce surcoût est relativement constant et de l'ordre de 5%. Pour le calcul de la spéculation, le surcoût oscille entre 10% et

40%, 10% étant la valeur vers laquelle converge les applications avec un grand temps d'exécution.

Le reste des surcoûts vient du traitement par ASTEX des informations récoltées et du temps de compilation des 7 versions exécutées de l'application. Bien que certaines étapes aient des complexités exponentielles, elle n'est pas fonction de la taille de l'application, mais de celle des objets traités. Il s'agit en particulier de la taille des codelets. Cette dernière, comme le montre la section 2.5.2.2, n'est pas corrélée avec la taille de l'application et reste dans un même ordre de grandeur. Dans une application plus longue, on aura donc proportionnellement plus de codelets mais d'une taille identique. Le temps de traitement global varie donc de manière proportionnelle avec la taille de l'application.

Toutes les exécutions effectuées par ASTEX ne sont pas nécessaires. L'acquisition de certaines statistiques et la validation du code généré peuvent être factorisés ou exécutés en parallèle sur une machine multicœur. Diminuer le nombre d'exécution de l'application dans le processus de partitionnement d'ASTEX est le facteur principal d'optimisation du temps de traitement. Le surcoût peut cependant être considéré comme raisonnable et ne demande pas de puissance de calcul supérieure à celle nécessaire pour l'exécution du code original.

Mémoire nécessaire au calcul

Le tableau 2.1 présente la taille de mémoire dynamique maximum utilisée par ASTEX et la taille des applications traitées. Cette taille de mémoire dynamique est acquise en utilisant le logiciel `valgrind` sur les différents étages du processus ASTEX en excluant les différentes exécutions de l'application instrumentée.

Famille	Application	Taille de l'application	Max mémoire dyn. (Mo)
NAS	MG	3483	10,61
NAS	LU	11416	11
NAS	FT	1858	9,12
NAS	EP	458	9,33
NAS	CG	1095	9,32
NAS	BT	11550	23.09
NAS	SP	11416	25,04
SPEC2006	401.bzip2	12208	11,30
H264	opt7	146990	113,30

TAB. 2.1 – Volume maximum de mémoire dynamique utilisé par ASTEX dans les applications NAS.

La taille de la mémoire dynamique utilisée dans le calcul du partionnement par

ASTEX est très modérée. En effet, la taille maximum mesurée sur les applications tests expérimentées est de 113 Mo, ce qui comparé aux 4Go de mémoire RAM de l'hôte est raisonnable. Ce point particulier n'est pas critique dans le passage à l'échelle du logiciel ASTEX.

Applications traitées

La taille des programmes en instruction est calculée par la méthode donnée dans la section 2.5.1.1. En appliquant cette technique aux trois familles d'applications, on obtient le tableau 2.5.

Famille	Application	Granularité en instructions de l'application
SPEC2006	429.mcf	2468
SPEC2006	433.milk	30682
SPEC2006	456.hmmer	63942
SPEC2006	458.sjeng	24042
SPEC2006	462.libquantum	8774
SPEC2006	470.lbm	2925
SPEC2006	401.bzip2	12208
NAS	MG	3483
NAS	LU	11416
NAS	FT	1858
NAS	EP	458
NAS	CG	1095
NAS	BT	11550
NAS	SP	10790
H264	optX	146990

TAB. 2.2 – Taille des applications tests choisies pour les expérimentations.

ASTEX traite sans difficulté des applications de taille raisonnable, *e.g.* 150Kinst. La taille des applications n'est pas limitative pour l'utilisation d'ASTEX.

Le nombre d'applications non traitées suite à des bogues logiciels est d'environ 10% sur un ensemble de plus de 150 applications tests, incluant les trois familles d'applications choisies pour cette section d'expérimentation. Ce chiffre, quoique raisonnable pour un logiciel en cours d'intégration, peut encore être facilement réduit. Les bogues sont quasiment tous identifiés et peuvent facilement être corrigés. Il s'agit de cas non considérés et d'erreurs de programmation.

Conclusion

Au vu des résultats d'expérimentation, sans prendre en compte la pertinence de la sortie d'ASTEX, le bilan sur l'état de fonctionnement du logiciel est positif. Il a les propriétés suivantes :

- il présente en terme de ressource utilisée et temps de calcul un profil favorable au passage à l'échelle, *i.e.* l'utilisation sur des applications plus conséquentes ;
- il est fiable, *i.e.* il s'applique sans difficulté à toutes les applications qui lui sont présentées ;
- son coût en ressource raisonnable le rend utilisable sur une machine de bureau classique et ne nécessite pas de matériel spécifique pour son exécution.

2.5.2.2 Évaluation de la détection des candidats

Cette section présente l'évaluation de l'étape de détection des codelets candidats. Il s'agit en particulier de garantir leur validité, leur stabilité et de caractériser leur nature.

Validité des résultats

Il s'agit de s'assurer que les codelets qui doivent être détectés le sont. Se basant sur des applications où ces noyaux de calculs sont connus, on s'assure de leur détection.

Les « NAS parallel benchmarks » ont cette propriété intéressante d'avoir leur noyaux de calculs clairement identifié. L'utilisation d'ASTEX sur ces applications écrites pour être parallélisées ne pose aucun problème. 100% des applications sont traitées et tous les noyaux présentant un temps significatif du temps d'exécution global mesuré avec Gprof ont été détectés. Trois codelets sur soixante-huit sont rejetés pour de problème de syntaxe non supportée. Le fonctionnement minimal d'ASTEX est validé.

Stabilité des résultats

La figure 2.19 présente les résultats obtenus sur un panel d'applications de NAS en couverture du temps d'exécution par les codelets en fonction de la taille du problème.

En faisant augmenter la taille des problèmes traités, possibilité offerte par les applications de NAS, quasiment aucun nouveau codelet n'est détecté, seuls certains non liés à la taille du problème deviennent insignifiants et donc finissent par être filtrés. Il s'agit, dans NAS, principalement des prologues et épilogues des applications. L'implantation effective ou non d'un codelet peut dépendre de la taille du problème en entrée.

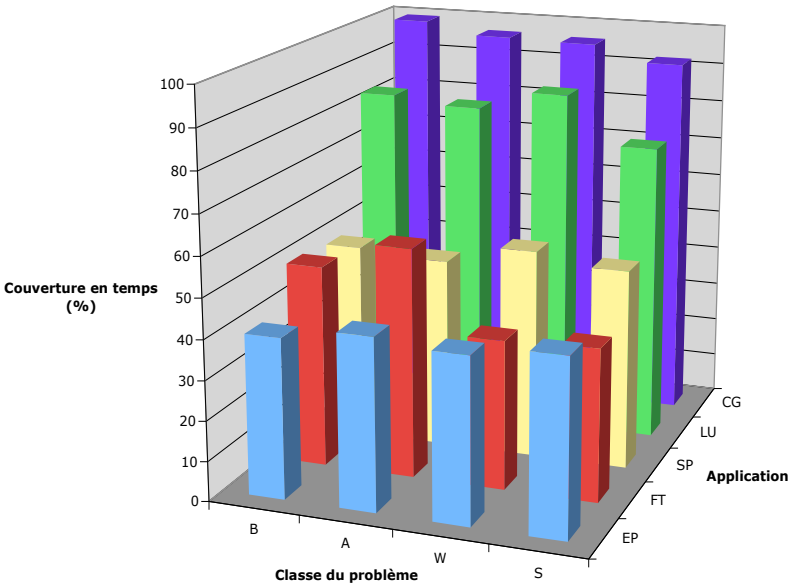

FIG. 2.19 – Variation de la couverture globale de la partition calculée par ASTEX en fonction de la taille du problème pour les applications de NAS.

De manière générale, la couverture en temps de l'ensemble des codelets varie peu avec la taille. Les résultats sont stables en apparence. Dans le détail, les codelets principaux ont leur couverture qui augmente quand dans le même temps les codelets secondaires, plus nombreux, stagnent ou diminuent en couverture. Le bilan est globalement nul. En conclusion, une fois implanté, l'importance d'un codelet est liée à la taille des données d'entrée.

La même expérimentation, *i.e.* faire varier la taille des entrées, sur h264 rend un résultat équivalent. Les données d'entrée étant découpées en éléments de tailles fixes et la complexité étant également répartie dans l'algorithme, la taille du problème en entrée (ici la taille de la vidéo à décoder/encoder) n'a pas d'influence.

Cependant, l'algorithme h264 possède de nombreux paramètres pour la taille de compression-décompression, le mode d'encodage, la résolution, le débit ... Ces derniers influent sur le contrôle de l'algorithme.

En faisant varier ces paramètres, les codelets détectés ne sont pas tous les mêmes, ou les bornes de boucles changeant ou bien leur couverture varie. Le tableau 2.3 présente la répartition des codelets détectés suivant les différentes options.

L'usage de certaines sections de code étant exclusif entre les options (codelet 4,6,7,8,-9,13,22,23,24,25), la mise en codelet, bien que potentiellement inutile, n'est pas proscrite dans l'élaboration d'une version paramétrable du programme partitionné. Enfin, dans de nombreux cas, les codelets restent les mêmes entre plusieurs options. S'il est fait usage de la spécialisation, afin de faire cohabiter les différentes options, les tests de spéculation

H264 option	0	1	2	3	4	5	6	7
codelet 1	X	X	X	X			X	X
codelet 2	X	X	X	X				
codelet 3	X	X						X
codelet 4	X							
codelet 5	X	X			X			
codelet 6	X							
codelet 7	X							
codelet 8	X							
codelet 9	X							
codelet 10	X		X	X				
codelet 11	X		X	X	X	X		
codelet 12		X	X	X	X	X	X	X
codelet 13		X						
codelet 14		X	X	X	X	X	X	X
codelet 15		X	X	X	X	X	X	X
codelet 16		X					X	X
codelet 17		X	X	X	X	X	X	X
codelet 18		X	X	X	X	X	X	X
codelet 19			X	X				
codelet 20			X	X	X			
codelet 21				X	X	X	X	
codelet 22						X		
codelet 23						X		
codelet 24							X	
codelet 25							X	

TAB. 2.3 – Codelet détecté suivant les options pour l'exécution de h264. Les codelets 13 et 21 ont une intersection non nulle.

devront intégrer les choix d'options pour utiliser la bonne version spécialisée. Quand deux codelets de deux options différentes se chevauchent sans être strictement égaux (13 et 21), leur existence simultanée est un cas non traité.

Nature des résultats

Famille	Appli.	Nb Codelet cov >10%	Nb Codelet cov <10%	Couverture (temps)
SPEC2006	429.mcf	0	3	14%
SPEC2006	433.milk	4	10	92%
SPEC2006	456.hmmer	reject	2	5%
SPEC2006	458.sjeng	0	20	49%
SPEC2006	462.libquantum	1	2	90%
SPEC2006	470.lbm	2	1	99%
SPEC2006	401.bzip2	1	18	52%
NAS	MG	2	2	87%
NAS	LU	2	11	86%
NAS	IS	reject	2	1%
NAS	FT	1	0	72%
NAS	EP	1	1	34%
NAS	CG	1	1	98%
NAS	BT	2	11	73%
NAS	SP	1	11	56%
H264	opt0	0	11	48%
H264	opt1	0	11	52%
H264	opt2	0	11	60%
H264	opt3	0	12	64%
H264	opt4	0	9	61%
H264	opt5	0	9	66%
H264	opt6	0	10	44%
H264	opt7	0	8	46,5%

TAB. 2.4 – Tableau récapitulatif des codelets candidats pour chaque application de test. Sont donnés le nombre de codelets qui ont une couverture d'exécution supérieure à 25%, inférieure à 25%, la granularité en nombre d'instructions assembleur x86 et enfin la couverture globale des codelets détectés.

Le tableau 2.4 présente les statistiques de détection des codelets pour les différentes applications tests. Pour toutes les options de h264, on obtient, en moyenne, environ 60% de couverture, répartie de manière assez uniforme sur une douzaine de codelets dont aucun ne dépasse les 10% de couverture. Pour les SPEC2006, le résultat est très

variable, de 90% de couverture en une quinzaine de codelets à 5% en deux codelets. Cette instabilité est liée à la nature très variée des applications qui forment les SPEC2006. Pour les NAS, on obtient environ 85% de couverture en un ou deux codelets importants et deux ou trois de plus faible couverture mais homogènes.

Famille	Application	Granularité de l'appli.	Granularité moyenne des codelets	Granularité totale des codelets	Couverture (instructions x86) (%)
SPEC2006	429.mcf	2468	32	96	3,89
SPEC2006	433.milk	30682	143	2002	6,52
SPEC2006	456.hmmer	63942	53	159	0,25
SPEC2006	458.sjeng	24042	73	1460	6,07
SPEC2006	462.libquantum	8774	62	186	2,12
SPEC2006	470.lbm	2925	293	879	30,051
SPEC2006	401.bzip2	12208	92	1840	15,07
NAS	MG	3483	135	540	15,50
NAS	LU	11416	397	5161	45,21
NAS	FT	1858	100	100	5,38
NAS	EP	458	62	124	27,07
NAS	CG	1095	97	194	17,72
NAS	BT	11550	345	4485	38,83
NAS	SP	10790	254	3048	28,25
H264	opt0	146990	170	1870	1,27
H264	opt1	146990	73	803	0,55
H264	opt2	146990	266	2926	1,99
H264	opt3	146990	249	2988	2,03
H264	opt4	146990	111	999	0,68
H264	opt5	146990	100	900	0,61
H264	opt6	146990	72	720	0,49
H264	opt7	146990	51	408	0,28

TAB. 2.5 – Tableau présentant les statistiques sur la granularité des codelets et des applications.

Le tableau 2.5 présente les résultats de granularité en nombre d'instructions assembleur des codelets détectés. Le chiffre de la première colonne est le nombre d'instructions total de l'application. Le chiffre de la seconde colonne est le nombre d'instructions moyen par codelet. La quatrième est la couverture totale des codelets de l'application en instructions et enfin la dernière colonne est la traduction en pourcentage de l'application globale de cette dernière donnée.

La proportion de l'application recouverte par les codelets en terme d'instruction est le plus souvent limitée, ce qui correspond bien à la définition de *hotpath*. Cependant, certaines applications présentent des taux élevés de recouvrement. Ils s'expliquent par

le grand nombre de codelets à la couverture en temps limité que l'on a détecté. Cette dernière statistique est donnée dans le tableau 2.4. Cet ensemble de « petits » codelets n'a de sens que si le système hôte a un surcoût raisonnable par codelet. Il appartient à l'utilisateur de les filtrer grâce aux paramètres d'entrée d'Astex en fonction de sa cible.

Globalement, tous les codelets détectés sont du même ordre de grandeur en nombre d'instructions. La longueur des codelets détectés, relativement homogène, montre la précision relative d'Astex et présente un résultat intéressant quant à la forme des noyaux de calcul dans les applications. Focaliser le développement d'un outil automatique de génération de code pour coprocesseur, ou le coprocesseur lui-même, sur un ordre de grandeur d'une centaine d'instructions assembleur semble un choix cohérent. Ce résultat demande a être confirmé par de plus larges expérimentations.

Conclusion

Parmi les résultats obtenus, tous sont cohérents avec la nature de l'application concernée. Ces résultats sont très encourageants pour la suite du partitionnement puisque la majorité des applications présentent des codelets avec une couverture en temps significative pour une longueur constante de l'ordre de la centaine d'instructions.

Une fois les codelets détectés, il faut les extraire, Ceci dans le cadre des systèmes hétérogènes à mémoire distribuée. Cette étape donne lieu au rejet d'un certain nombre de codelets. Il s'agit de l'objet de la partie 2.5.2.3.

2.5.2.3 Validation des candidats

Une fois les codelets candidats détectés, il faut leur faire franchir les étapes successives du processus de partitionnement. Il existe diverses raisons pour qu'un codelet candidat ne puisse aller jusqu'à son implantation dans une partition. Sur la base des expérimentations réalisées, j'ai classé les erreurs en trois catégories distinctes :

1. Syntaxe non supportée : expressions rejetées, structures de contrôle non traitées (goto)... Ces problèmes peuvent théoriquement être résolus, il s'agit d'une marge de progression pour les fonctionnalités du logiciel Astex.

2. Structure mémoire non supportée : double déréférence, arithmétique de pointeur... Ce type d'erreur est la concrétisation des limites de notre modèle.

3. Couverture insuffisante : cette erreur vient principalement de l'imprécision de l'évaluation de la couverture dans la détection des *hotpaths*. Cette erreur permet de filtrer les fausses détections.

La figure 2.21 présente la répartition des causes de rejet pour les familles d'applications NAS et SPEC2006.

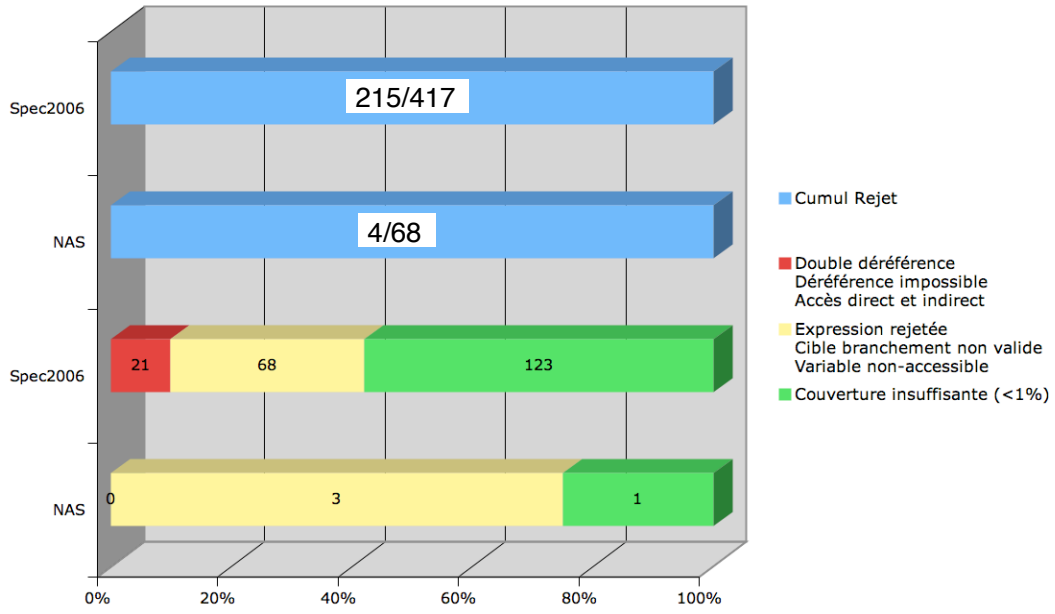

FIG. 2.20 – Schéma de répartition des causes de rejet des codelets, h264 n'apparaît pas car il n'y a aucun rejet.

Il y a peu de rejets dans les NAS. En effet, le code des applications ayant été écrit sous forme parallèle, les structures de contrôle et de données sont adaptées au partitionnement. Ces applications sont donc parfaitement adaptées au modèle d'ASTEX.

Exceptée une fausse détection, les codelets correspondent tous aux noyaux identifiés dans les NAS. Par conséquent, leur couverture en temps significative dans l'application globale n'est pas surprenante.

Les codelets rejetés dans NAS le sont du fait du logiciel ASTEX, il s'agit d'expression rejetées. Une amélioration du logiciel supprimera donc les problèmes restants liés au NAS.

Les applications dans SPEC2006 sont de nature plus variée et leur provenance diverse. Elles présentent un panel intéressant de méthodes de programmation de la part de leurs auteurs.

10% des rejets s'établissent sur la base d'incompatibilité entre le modèle d'ASTEX et la sémantique du programme. Il s'agit en effet de problèmes liés à la structure des accès à

la mémoire. Les applications connaissant le plus grand nombre de rejets de cette nature nécessitent une réécriture pour être compatibles avec notre modèle. En identifiant les problèmes d'écriture sur les chemins chauds, ASTEX fournit une information essentielle pour la réécriture des programmes pour le partitionnement, faisant gagner un temps certain aux programmeurs.

30% environ des rejets le sont pour les cas non traités par le logiciel implantant ASTEX. La diversité des écritures dans SPEC2006 explique l'importance de ces résultats. Pour réduire cette statistique, un soin particulier doit être apporté dans l'évolution du logiciel afin de traiter les cas momentanément écartés dans le développement.

Enfin, la moitié des rejets intervient sur des cas de fausses détections. Il ne s'agit pas là d'un véritable problème puisque une fois détectés, ces candidats sont aisément filtrés. La seule perte qui puisse être imputée à cette fausse détection est celle des calculs inutiles dans les premières étapes du processus de partitionnement d'ASTEX. Ces calculs intervenant avant la partie d'analyse complexe et coûteuse en temps, le surcoût n'est pas significatif. Cependant, si de fausses détections interviennent, le contraire, c'est-à-dire l'oubli de vrai codelet, est-il possible ? La section 2.5.2.2 répond en partie à ce problème. Cependant, il n'existe aucune démonstration formelle du bon fonctionnement de la détection des points chauds dans le cadre d'ASTEX.

2.5.2.4 La spécialisation

Dans le modèle mis au point pour son développement, ASTEX produit une version spécialisée des codelets extraits selon trois types de spéculation :

- spéculation sur les valeurs des paramètres scalaires, en particulier les bornes de boucles et les tailles des structures,
- spéculation sur l'alignement, afin de permettre un alignement plus adapté sur le système hôte ou de spécialiser le code qui contient des accès à la structure,
- spéculation sur les zones accédées afin de résoudre les alias et ne considérer que les zones réellement accédées.

Dans la version actuelle d'ASTEX, cette spéculation est calculée après extraction sur les paramètres des arguments passés à la fonction du codelet. À partir de ce profil spéculatif, sont générés une version spécialisée du codelet et le test de spéculation associé. Les trois paragraphes suivants de cette section présentent les résultats obtenus pour chaque type de spéculation. Il s'agit de vérifier que le choix des éléments spéculés est pertinent et que la mise en œuvre de cette spéculation est efficace.

Afin d'obtenir suffisamment de résultats, le seuil de filtration sur la couverture des *hotpaths* détectés a été abaissé à 0. Les statistiques portent donc sur les données de 321 codelets.

Spéculation sur les valeurs scalaires

	NAS	H264	SPEC2006	Total
Local	14	3	83	100
100%	32	99	38	169
85%	1	37	8	46
66%	1	30	7	38
2 valeurs	0	13	24	37
3 valeurs	0	18	6	24
4 valeurs	0	2	4	6
Bcl. ext.	9	3	20	32
Instable	0	33	68	101
Total	70	241	278	589

TAB. 2.6 – Tableau présentant le résultat du calcul du profil spéculatif pour toutes les variables scalaires. La ligne *local* correspond à l'usage dans un codelet de variables locales exclusivement. L'absence de variable scalaire vivante en entrée et sortie est un résultat notable.

Le tableau 2.6 présente les résultats du calcul du profil spéculatif pour les variables scalaires. Les statistiques présentées sont classées dans des catégories dont les définitions sont les suivantes :

- local : ce sont les cas où l'on a observé aucune entrée scalaire, toutes les variables peuvent être recréées localement et ne nécessitent ni copie ni initialisation ;
- 100% : ce sont les variables pour lesquelles une seule valeur a été observée ;
- 85% : ce sont les variables qui n'appartiennent pas aux catégories précédentes et pour lesquelles une valeur unique a été observée dans plus de 85% des cas ;
- 66% : ce sont les variables qui n'appartiennent pas aux catégories précédentes et pour lesquelles une valeur unique a été observée dans plus de 66% des cas ;
- 2 valeurs : ce sont les variables qui n'appartiennent pas aux catégories précédentes et pour lesquelles deux et seulement deux valeurs ont été observées avec une couverture significative ;
- 3 valeurs : ce sont les variables qui n'appartiennent pas aux catégories précédentes et pour lesquelles trois et seulement trois valeurs ont été observées avec une couverture significative ;
- 4 valeurs : ce sont les variables qui n'appartiennent pas aux catégories précédentes et pour lesquelles quatre et seulement quatre valeurs ont été observées avec une couverture significative ;
- boucle externe : ce sont les variables qui correspondent à des boucles externes ;
- instable : ce sont des variables qui ne présentent ni valeur dominante ni régularité.

De manière générale, le nombre de paramètres scalaires en entrée est assez réduit. Dans 31% des cas, les variables sont exclusivement locales. Les variables scalaires sont souvent locales. En répartissant le reste des variables sur le reste des codelets, on obtient

une moyenne de 2,21 variables par codelet.

Dans 89% des cas, la spéculation donne des résultats exploitables, *i.e.* les résultats sont ni instables ni non traités (boucle externe). Il est très souvent possible de spécialiser les codelets en fonction des paramètres scalaires, ce qui inclut les bornes boucles.

Parmi les 11% d'échecs pour le calcul de la spécialisation, 33% sont des problèmes de détection des variables d'induction externe. Si le codelet est dans une boucle externe, même si le paramètre constitué par la variable d'induction de la boucle externe n'est pas un paramètre stable du point de vue du codelet, sa variation régulière pourrait être exploitée pour la spécialisation.

Les 7,3% d'échecs restants sont les variables instables pour lesquelles il n'est pas possible de spéculer et donc de spécialiser. Elles doivent être passées en paramètre du codelets.

Par ailleurs, on observe dans le tableau que les trois familles d'applications tests ont des signatures différentes. L'utilisation des scalaires varie d'un programmeur à l'autre. Il faut adapter la politique de spécialisation au profil des applications et des codelets.

Après l'étape du calcul de la spéculation, sont générées les versions spécialisées des codelets. Près de la moitié des versions spécialisées est un échec. Quand la spécialisation des scalaires contribue à cet échec, sauf exception, cela vient de problème de compatibilité entre les valeurs spéculées pour les différentes variables d'un même codelet. L'exclusivité éventuelle entre les variables demande pour être traitée le calcul d'un coefficient de corrélation entre les valeurs des différentes variables.

Actuellement, une seule valeur par variable, la plus fréquente, est choisie comme valeur de spécialisation. Le cas des variables ayant une répartition quasi équivalente entre un nombre restreint de valeure n'est pas traité. Pour un nombre de valeure compris entre 2 et 4, leur proportion est de 15% des variables qui peuvent être spéculées. Cette proportion étant non négligeable, il est intéressant de mettre au point les mécanismes nécessaires à l'exploitation effective dans la spécialisation de cette spéculation.

Spéculation sur l'alignement

Le tableau 2.7 présente le résultat du calcul de la spéculation sur l'alignement des données. Les catégories utilisées pour classer les statistiques sont un sous-ensemble de celles utilisées dans le paragraphe précédent.

Dans 71% des cas, il n'y a qu'une seule valeur d'alignement qui est parfaitement capturée par Astex. C'est un très bon résultat qui augure un taux de réussite conséquent dans la spécialisation sur les alignements.

Il y a seulement 7,2% de rejets. Il s'agit de cas où aucune valeur d'alignement stable n'a été trouvée.

H264 et les SPEC2006 présentent un profil de statistique similaire qui *a priori* est la norme. Le profil de NAS n'est pas incompatible, au contraire, tout est très bien aligné ; ceci est dû au soin apporté dans le choix des structures dans la programmation. Contrairement au cas de la spécialisation sur les variables scalaires, la spécialisation

	NAS	H264	SPEC2006	Total
100%	204	95	346	645
85%	0	31	11	42
66%	0	8	5	13
2 valeurs	15	32	31	78
3 valeurs	0	0	2	2
4 valeurs	0	57	2	59
Instable	0	64	2	66
Total	219	287	399	905

TAB. 2.7 – Tableau présentant le résultat du calcul du profil spéculatif pour l'alignement des différentes structures (variables non scalaires) passées en paramètre du codelet.

sur les alignements de données ne semble pas nécessiter de politique différente d'une application ou d'un codelet à l'autre.

Par ailleurs, les tests de spécialisation sont générés et fonctionnent. Le bilan général sur la spécialisation en fonction des alignements est donc positif. Les expérimentations valident l'approche implantée dans ASTEX.

Spéculation sur les zones accédées

L'expérimentation de la spéculation sur les zones de mémoire présente actuellement un bilan mitigé. Dans l'écrasante majorité des cas, chaque pointeur est utilisé sur une seule zone mémoire, cette zone mémoire étant éventuellement très grande. Sans le découpage en portions de zones mémoire et sans histogramme des valeurs, peu de cas intéressants pour une spécialisation se présentent. De plus, rien n'est observé en dehors des codelets, ce qui supprime la possibilité de détecter les zones pouvant résider en mémoire entre deux appels de codelet. Bien que ces fonctionnalités est été montrées pertinentes dans le prototype d'ASTEX [40], elles nécessitent leur intégration dans l'outil actuel pour être testées sur un nombre suffisant d'applications.

Enfin, dans le logiciel actuel, le test d'alias entre les paramètres de type pointeur sur zone mémoire est faux et non optimisé. Il est actuellement la source de la plupart des rejets de spécialisation qui figurent sur l'histogramme, de la figure 2.21, dans la section 2.5.2.5.

Conclusion

Au vu des résultats des expérimentations, l'approche pour la spécialisation n'est pas encore complètement au point. Cependant les chiffres obtenus sont très encourageant ; dans les cas où la spécialisation fonctionne, elle fonctionne parfaitement.

Les points critiques à améliorer sont :

– la correction des tests de spéculation faux,

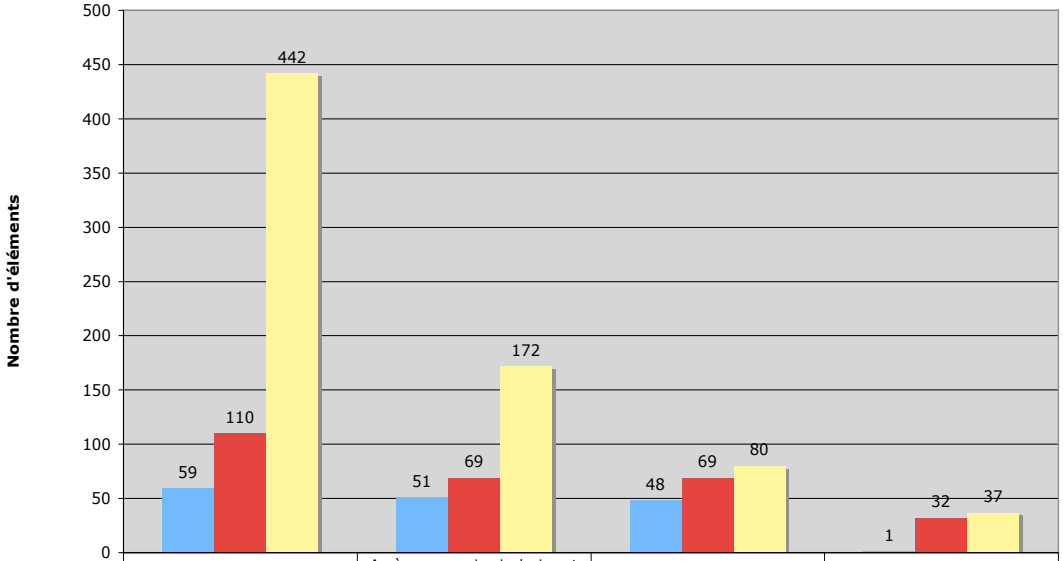

FIG. 2.21 – Schéma du nombre de codelets restants à l'issue de chaque étape du partitionnement.

– la subdivision des zones observées,
– la détermination des zones résidentes dans la mémoire hôte.

Tous ces points concernent la spéculation sur les zones mémoire.

À des fins d'optimisation dans tous les cas, il serait intéressant d'ajouter :

– le calcul d'un coefficient de corrélation entre les différentes valeurs spéculées,
– la gestion des cas où un nombre fini et restreint de valeur est observé avec une fréquence significative, *e.g.* deux variables couvrant chacune 50% des cas.

Enfin, la spéculation sur les chemins, c'est-à-dire l'utilisation du surplus d'information des *hotpaths* par rapport aux *hotspot*, a dans l'implantation actuelle été écarté. Cette perspective présente un intérêt particulier pour certaines architectures comme les GPU dans lesquelles les branchements ont un impact fort sur les performances.

2.5.2.5 Bilan sur le fonctionnement d'ASTEX

À l'issue des différentes étapes du processus du calcul du codelet, un certain nombre de rejet intervient. La figure 2.21 présente sous la forme d'un histogramme le nombre de codelets restants après chaque étape pour chaque famille d'application traitée : NAS, SPEC2006 et H264.

En bleu, figure le nombre de chemins chauds détectés, en violet le nombre restant de ces chemins chauds après suppression des éléments à l'intersection non vide. Le deuxième chiffre est donc celui des codelets candidats, cette statistique est traitée en détail dans la section 2.5.2.2.

L'inclusion des chemins détectés apparaît naturellement dans les structures en nid de boucle. Le chevauchement simple est, lui, le fait de la nature des chemins chauds. Sans être égaux, ils peuvent partager certaines portions du code.

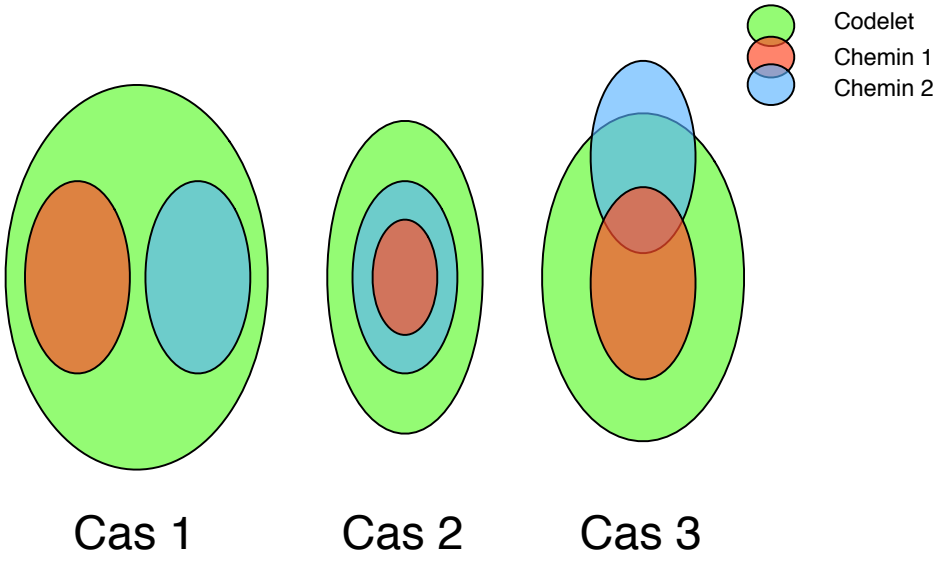

FIG. 2.22 – Cas de figure pouvant se présenter lors du passage des chemins chauds aux codelets.

Le phénomène d'inclusion et de chevauchement est amplifié par l'extension des chemins chauds en codelets. La figure 2.22 présente une vue graphique du passage chemin/codelet dans les trois cas présentant un chevauchement ou une inclusion dans le résultat. Bien qu'exploitable, le cas le plus à droite n'est pas traité. Le cas central est celui de l'inclusion stricte, il est un cas particulier de celui de droite. Il n'oblige cependant pas à exclure des codelets candidats une portion de code détectée comme point chaud. Il n'y a donc pas de perte de couverture des chemins détectés. Enfin, le cas de gauche est celui d'une structure englobante recouvrant deux chemins distincts. Dans ce cas, le codelet formé englobe les deux chemins comme dans le cas de l'inclusion simple. On ne conserve donc qu'un seul codelet pour deux chemins. Afin de traiter ces trois cas et de raffiner les résultats produits par ASTEX, il est possible d'apporter les deux améliorations suivantes :

- réintroduire la spéculation sur le contrôle possible avec le modèle ;
- introduire une hiérarchie de codelet laissant la possibilité d'avoir des codelets dans les codelets.

Après le passage en codelet intervient les rejets tels que ceux traités dans la section 2.5.2.3. La proportion de rejet est très variable d'une famille d'application à l'autre. Dans H264 et NAS interviennent très peu de rejets, en effet ces applications ont reçu un soin particulier lors de leur implantation pour permettre l'analyse automatique de programme. Le fonctionnement d'Astex est donc facilité. Dans les SPEC2006 prés de la moitié des codelets est rejetée. Les causes sont détaillées dans la partie 2.5.2.3. Afin de pallier ces rejets constatés par expérimentation, les améliorations proposées sont les suivantes :

- étendre les capacités du logiciel aux cas non traités pour réduire la distance entre l'implantation et le modèle théorique ;
- développer des outils pour la réécriture des codes d'applications incompatibles avec le modèle ; il s'agit en particulier d'extraire les informations permettant d'automatiser la réécriture ou à défaut de guider le programmeur ;
- étendre le modèle à la gestion de cas courants en programmation et exclus du cadre actuel ; il s'agit, en particulier, de l'utilisation de structures à plusieurs indirections mais régulières, *e.g.* les listes chaînées.

Dans la version courrante d'Astex, sans la totalité des fonctionnalitées du prototype, il est difficile d'évaluer la qualité de la spécialisation des codelets. Cependant, certaines lacunes, en particulier sur les zones mémoire, apparaissent critiques pour l'exploitation efficace des capacités de spécialisation des codelets. Pour aller plus loin dans le développement, sont à mettre ou remettre en place les fonctionnalités suivantes :

- corriger et optimiser des tests de spéculation,
- subdiviser les zones mémoire en découpant selon les éléments réellement accédés,
- ajouter un calcul de corrélation entre les différentes valeurs spéculées,
- déterminer les zones résidentes et permettre l'optimisation spéculative des communications en réintroduisant l'instrumentation du code inter-codelet,
- créer plusieurs versions spécialisées d'un même codelet et mettre en place un choix a l'execution (arbre de decision).

2.5.3 Étude prospective de l'applicabilité des codelets

Une fois les codelets détectés, les profils spéculatifs étant extraits, ils deviennent implantables. La question est de savoir dans quel contexte les résultats d'Astex sont applicables. Il s'agit de la partie la plus complexe de l'analyse d'Astex. Le logiciel arrivant à peine à maturité, cette étude est encore très préliminaire et prospective.

Il existe beaucoup de possibilités pour les paramètres d'implantation et les architectures cibles. Une partition bonne pour une série de paramètres d'exécution donnés sur une machine donnée n'est pas forcément valide et/ou efficace sur une autre. C'est cette interférence entre système et implantation qu'il faut savoir exprimer et mesurer avant de l'optimiser.

La méthode choisie est l'étude du calcul de paramètres discriminatoires des codelets portant sur la mesure d'affinité d'un codelet avec une architecture. En utilisant le

modèle de coût théorique et en se basant sur des expérimentations ou des données de la littérature, on construit le modèle théorique de plusieurs exemples d'architectures parmi les plus usitées du moment : le GPU [54, 60, 61, 64, 67], le CELL [44, 45, 76] et le multicœur *general purpose* [66, 69].

À l'issue de cette étude sur les modèles théoriques des architectures pour le choix des partitions en codelet, est proposée une représentation de l'affinité entre architecture et codelet et ses possibles interprétations et utilisations.

2.5.3.1 Les coûts d'implantation

Actuellement les partitions calculées sont exprimées dans le code à l'aide du langage support de *HMPP*. Le code compilé s'appuie sur un *run-time* simple créé pour les expérimentations. L'utilisation automatique des codelets sur des architectures distribuées et spécifiques, en particulier les GPU, est en cours d'expérimentation. La donnée sur le surcoût réel de l'implantation d'un codelet dans un système réel est donc absente.

Dans le cas particulier de l'exécution sur un multicœur, support des expérimentations de ce chapitre, les résultats en temps d'exécution, quelles que soient les applications tests, montrent une différence inférieur à 10% entre la version avec codelet et sans, le surcoût minimal est donc faible. Cette conclusion est à confirmer sur d'autres systèmes.

Dans environ un tiers des cas, la version utilisant le découpage en codelets est plus rapide à s'exécuter sur les applications de grande taille (avec succès des spécialisations et gcc4.4 -O3 ou -O2). Cette accélération se chiffre à environ 4 à 6%. La différence majeure entre les applications qui sont accélérées et celles qui sont ralenties par l'utilisation de codelets semble être le taux de succès des spécialisations. Une spécialisation ratée à donc un impact fort sur les performances. Cette étape du processus de partitionnement d'Astex ne fonctionne pas encore bien, c.f. 2.5.2.4. Dans le cas général, les résultats présentent donc un fort potentiel d'amélioration, le pire des cas semblant devoir converger autour d'une différence de performance non significative.

L'accélération des applications avec codelets est un résultat inattendu et demande à être confirmé et analysé pour être validé et éventuellement exploité. Il confirme cependant l'utilité d'étudier l'éventuel impact que pourrait avoir l'utilisation du partitionnement pour une compilation différenciée, c'est-à-dire la compilation indépendante avec des options spécifiques et optimisées pour chaque codelet.

En conclusion, les expérimentations et observations actuelles n'ont pas montré d'influence critique en terme de performance du surcoût d'implantation des codelets.

2.5.3.2 Les coûts de communication

Dans l'état actuel de l'implantation du logiciel, comme détaillé dans la section 2.5.2.4, l'impact réel des communications est difficile à évaluer.

Les expérimentations passées [40, 41] et d'autres études dans la littérature évoquent l'impact critique des communications et leurs nécessaires optimisations [9, 20, 21, 22, 26, 54, 59].

Le chapitre 3 traite en profondeur le problème de l'optimisation des communications et propose des pistes possibles pour leurs optimisations.

2.5.3.3 Utilisation du modèle de coût

Lors de l'implantation des codelets sur un système donné, le choix de la partition à implanter doit être cohérent avec les capacités de l'architecture cible. Il s'agit en particulier d'évaluer l'accélération minimum nécessaire à obtenir sur un codelet, pour que son implantation soit rentable. Si ce chiffre de *speed-up* est réaliste comparé aux autres résultats observés sur ce type d'architecture, alors on peut tenter l'implantation effective.

Les deux paragraphes suivants de cette section sont l'application du modèle de coût respectivement au GPU et au Cell. Le volume de données à communiquer est considéré comme étant le total des entrées et sorties pour l'ensemble des appels au codelet. Par exemple, un codelet appelé 1000 fois avec 1Mo en entrée et 1 Mo en sortie représente un volume de communication de 2 Go. Pour chaque architecture, les communications sont considérées effectuées depuis la mémoire centrale du système, *e.g.* la mémoire RAM pour le multicœur ou le GPU. Autant que possible, les différents chiffres liés aux technologies utilisées sont à jour, mais leur imprécision éventuelle n'enlève rien au raisonnement.

L'utilisation d'accélérateur de type GPU

Les courbes du graphique 2.23 sont celles obtenues pour l'utilisation d'un GPU comme coprocesseur. La bande passante considérée est de 8 Go/s, ce qui correspond à la norme PCIe16x 2.0. On considère le surcoût d'implantation des codelets comme 10% du temps de l'application globale, ce qui comprend les appels aux fonctions codelets, l'initialisation du matériel, les tests de spéculation...

Dans le cas du GPU, quelque soit la granularité des codelets, on arrive très vite dans un état saturé. Le volume de données communiquées est donc le point critique dans l'utilisation des GPU.

Bien que la copie initiale soit coûteuse, une fois les données chargées en mémoire dans le GPU, la RAM GPU est beaucoup plus rapide que celle du CPU. Quand le problème ne tient plus dans le cache L2 du CPU, on peut obtenir un gain substantiel même si l'algorithme initial n'avait pas de *speed-up* significatif sur le GPU pour les tailles de problème plus petites []. Il faut pour cela que les principales entrées sorties ne puissent être chargées qu'une seule fois et rester résidentes dans la mémoire du GPU.

Utilisation d'un coprocesseur vectoriel : Cell

Le graphique 2.24 montre l'évaluation du modèle de coût dans le cadre du Cell. La valeur choisie pour la bande passante correspond à un maximum difficilement atteignable de 32 Go/s [76]. Le surcoût considéré pour l'exécution en parallèle est de 4% du temps de l'application globale [76].

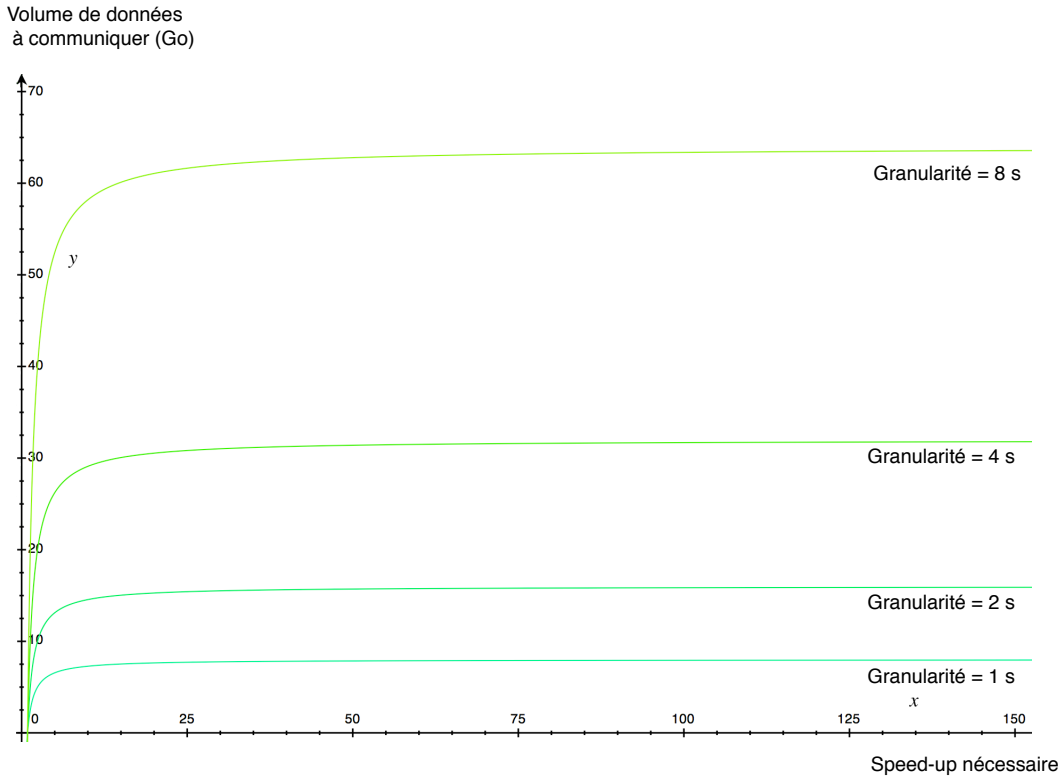

FIG. 2.23 – Courbes résultant de l'application du modèle de coût avec les données théoriques du GPU et de la connexion PCIe 16x 2.0 en fonction de la granularité en temps des codelets détectés.

Dans le cas du Cell, les seuils de saturation arrivent aussi très rapidement. La bande passante, très grande, autorise des transferts de données beaucoup plus conséquent qu'avec le GPU. La plage d'utilisation du Cell est donc plus large que celle du CPU.

Dans le cas du Cell comme dans celui du GPU, volume de communication et granularité sont les deux paramètres essentiels du surcoût de mise en œuvre des codelets dans un système à mémoire distribuée.

2.5.3.4 Définition et utilisation de l'affinité codelet/architecture

D'autres approches récentes abordent le problème du choix d'exécuter ou non un noyau de calcul sur le coprocesseur. C'est le cas par exemple de Qilin [87]. Le modèle empirique choisi dans cette approche comme entrée de l'algorithme de choix d'éxécution des codelets sur les coprocesseurs peut être adapté aux données spéculatives d'ASTEX ; en particulier, au niveau des tailles des zones de données et donc de l'estimation des communications. Qilin est une approche intéressante d'implantation du choix dynamique de

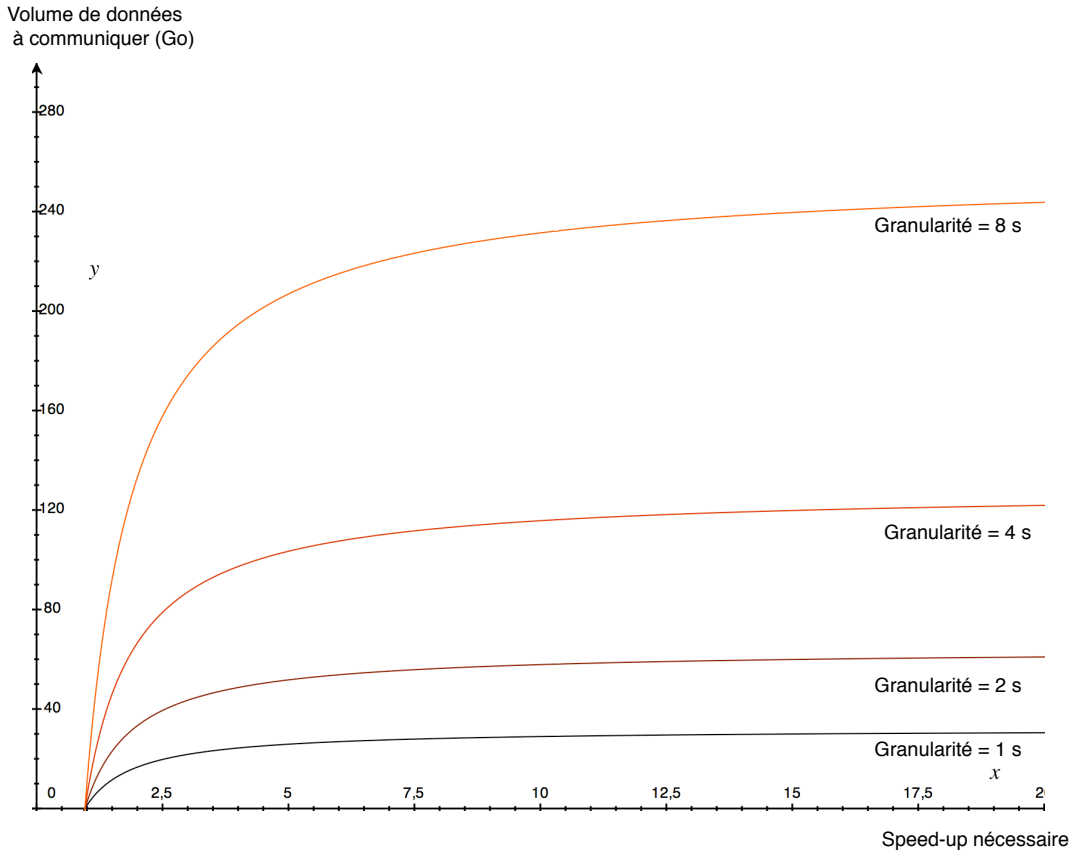

FIG. 2.24 – Courbes résultant de l'application du modèle de coût avec les données théoriques du Cell en fonction de la granularité en temps des codelets détectés.

l'exécution ou non d'un codelet.

L'approche prospective envisagée dans un premier temps pour la mesure de l'applicabilité d'un codelet dans ASTEX est statique. Elle intervient en fin de processus pour la spécialisation et la sélection finale de la partition. L'objectif visé est triple :

- mesurer l'affinité des codelets avec les architectures disponibles dans le système ;
- orienter la modification des paramètres de détection et de construction des codelets dans le cadre d'une utilisation itérative d'ASTEX ;
- guider les optimisations pour faire converger le profil d'un codelet vers celui d'affinité de l'architecture.

Le schéma 2.25 est une "carte d'applicabilité" des codelets. Il s'agit d'une vue synthétique non basée sur des observations réelles. Chaque motif géométrique correspond à une version d'un codelet d'une application. Les régions triangulaires représentent la zone d'affinité pour une architecture donnée.

Une fois construit, ce schéma aide à déterminer quelles versions et quels codelets

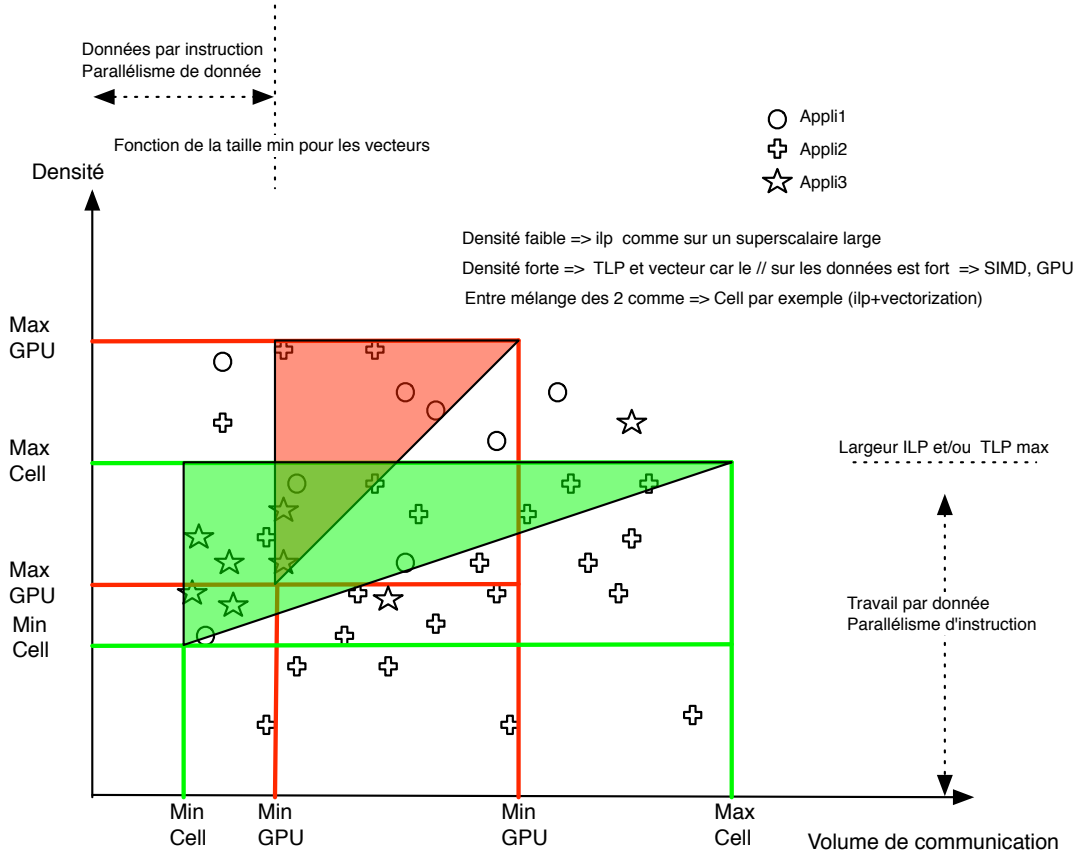

FIG. 2.25 – Schéma synthétique représentant l'applicabilité des codelets sur chaque ressource en fonction des caractéristiques de granularité et de volume de communication.

implanter. L'automatisation du choix des versions à implanter pour les ressources du système se résume à l'évaluation des coordonnées du codelet dans un système d'inéquation décrivant les limites de la zone d'affinité.

Afin de maximiser les chances d'obtenir une accélération pour l'application finale, il faut maximiser l'intersection entre les zones d'affinités et les codelets. Cela peut se faire par l'optimisation du système ou bien encore en « déplaçant » les codelets.

Pour optimiser le système, deux solutions sont possibles. Soit, si le système le permet, on modifie les caractéristiques ou le choix du coprocesseur (co-design), soit on optimise le *run-time*.

Afin de déplacer un codelet dans la zone d'affinité d'un coprocesseur donné, il faut réécrire son code. Il peut s'agir d'un changement complet d'algorithme, d'un changement de granularité par l'inclusion d'une boucle externe ou l'exclusion d'une boucle interne, de l'optimisation des communications, de parallélisation...

L'acquisition de ce schéma permet donc de guider le programmeur dans son choix

d'architecture, d'algorithme et d'optimisation pour son application.

2.6 Conclusions et perspectives

L'objectif principal de mon travail de thèse était la mise au point d'un processus automatique de partitionnement d'application. L'approche choisie se base sur la spéculation et possède les caractéristiques suivantes :

- elle supporte l'exécution spéculative,
- elle supporte les systèmes à mémoire distribuée,
- elle supporte les systèmes hétérogènes,
- elle est indépendante de l'architecture.

La complexité acceptable de la détection et la taille maîtrisée des informations de profilage rendent le processus de calcul des codelets compatible avec les applications de relativement grande taille. Les résultats obtenus par expérimentation valident l'approche hybride (basée sur les données statiques et dynamiques) pour le partitionnement automatique d'application.

Dans sa version actuelle, le logiciel Astex ne reprend pas toutes les possibilités du modèle exploré et validé dans le prototype de recherche. Il est cependant parfaitement fonctionnel et entame depuis 2009 son exploitation commerciale. Du point de vue du logiciel, de nombreux développements restent à faire.

Les expériences menées jusqu'à aujourd'hui ont permis de mettre en lumière un certain nombre de perspectives pour l'évolution d'Astex :

- l'optimisation des tests de spéculations,
- l'utilisation de l'outil pour déterminer l'affinité entre architecture et application,
- l'ordonnancement des tâches,
- l'allocation des ressources (*affinity scheduling*),
- l'utilisation de l'outil de partitionnement pour une compilation « différenciée »,
- la gestion du parallélisme -il s'agit notamment de gérer les problèmes de synchronisation entre les codelets-,
- la gestion de structures de données plus complexes par la mise en place de mécanismes de spéculation et de translations d'adresse nécessaires à la gestion d'accès comportant de multiples indirections.

L'étape actuelle dans la prospection sur les applications du profil spéculatif généré par Astex est de tester l'utilisation sur le code partitionné et spécialisé d'un compilateur Auto-adaptatif : gcc milepost [71].

Lors de discussions scientifique avec des architectes, ces derniers ont montré un intêret certain pour l'outil et ces résultats dans le cadre du *co-design*, cette voie est à explorer.

Dans le développement de nouvelles optimisations avec Astex, la prochaine étape est l'optimisation des communications. Elle bloque actuellement sur la simplification excessive de la spéculation sur les zones mémoires dans la version actuelle du logiciel, en particulier l'absence de profilage sur les accès en dehors des codelets. Cette situation est appelée à évoluer rapidement.

Chapitre 3

Optimisation des communications

Introduction

Étant donné le ratio coût performance des GPUs actuels, l'utilisation des processeurs graphiques comme accélérateurs matériels tend à se généraliser. Plusieurs environnements de programmation comme CUDA [61], RapidMind [60], PeakStream [64], HMPP [42] ou CTM [70] ont rendu plus accessible et efficace l'utilisation des GPU. Le travail demandé aux programmeurs afin d'obtenir une accélaration significative de son application grâce au GPU reste très lourd, en temps comme en compétence.

Actuellement, l'usage des GPU comme coprocesseurs matériels pour accélérer les applications est limité par les communications induites entre la mémoire principale et celle du GPU. Le surcoût dû aux communications est tellement élevé qu'il n'est même pas toujours rentable de déporter l'exécution d'un noyau de calcul intensif sur l'unité distante. Le problème d'optimisation des communications est habituellement laissé aux programmeurs qui affineront leurs solutions par essais successifs. Le programmeur doit décider quand précharger, quand mettre à jour et quand rapatrier les données contenues dans la mémoire distante. Tout ceci en s'assurant que les données dans la mémoire du GPU sont à jour lors de l'exécution déportée du noyau de calcul.

Les méthodes d'optimisation de communication portent sur trois points principaux dont l'influence est variable suivant les architectures et la granularité des codelets. Ils sont les suivants :

- charger les données au plus tôt afin de masquer une partie de la latence par le calcul du processeur principal ;
- éviter les copies redondantes entre deux appels du codelet afin de laisser libre la bande passante pour les communications utiles et éviter les surcoûts éventuels d'annulation de copie en cours ;
- recopier uniquement les zones ou parties de zones modifiées entre deux appels du codelet, en particulier détecter les zones résidentes, *i.e.* celles qui peuvent rester dans la mémoire du coprocesseur entre deux appels du codelet.

Dans l'article [41] nous présentons une solution simple, entièrement logicielle, basée sur les données spéculatives. Dans cette approche préliminaire, l'objectif est de montrer

la nécessité d'une optimisation et l'intérêt des données spéculatives pour l'élaboration de cette dernière. Cependant, l'évaluation de la qualité de cette solution passe par l'élaboration de solutions plus complexes basées sur la définition formelle du problème. Il s'agit de déterminer si une solution optimale existe et de savoir mesurer la distance entre deux solutions. De cette étude, encore incomplète, présentée dans ce chapitre semble se dégager l'absence de solution optimale et la suffisance d'une solution simple comme celle de [41] avec quelques améliorations pour traiter les cas les plus fréquents d'exception. La preuve de l'indécidabilité du problème reste ouverte.

Ce chapitre présente l'étude préalable à la mise en place de techniques automatiques d'optimisation du préchargement logiciel des données pour l'exécution déportée sur des unités distantes des noyaux de calcul. L'application se concentre sur les GPU. Les techniques proposées reposent sur une approche hybride mêlant les données spéculatives issues de l'utilisation d'Astex [40, 41] et les données issues d'une analyse statique du code. Trois algorithmes d'optimisation sont alors proposés.

La technique abordée porte sur le langage C et s'appuie sur l'environnement HMPP [42]. Elle gère les problèmes d'aliasing. Toutefois, les principes de base de l'algorithme d'optimisation restent transposables à d'autres langages et architectures cibles.

Dans ce chapitre, suite à la formalisation du problème, je donne la définition d'une solution, ses conditions de validité, et discute la mesure de sa qualité.

Afin de pouvoir mettre en œuvre les algorithmes d'optimisation des communications, je présente tout d'abord la construction d'une fonction de coût adaptée au cas des systèmes hétérogènes distribués. J'introduis ensuite les données dynamiques dans le calcul de l'optimisation à partir des données spéculatives issues d' Astex, en particulier afin d'introduire une classification des accès.

Une fois les différents éléments construits et initialisés, je présente trois algorithmes possibles pour la phase d'optimisation du placement des communications et discute leurs avantages et inconvénients.

Après les expérimentations préliminaires, je conclus sur les travaux à venir et les perspectives ouvertes pour l'utilisation de la spéculation dans l'optimisation des programmes partitionnés. En particulier, je propose des éléments d'une solution dérivée de [41] permettant l'implantation simple et efficace pour optimiser le placement des communications.

3.1 Formalisation du problème

La formalisation du problème et de ses solutions est un préalable nécessaire à la construction et l'évaluation par un algorithme de l'application optimisée.

Afin d'énoncer le problème d'optimisation du placement des communications dans une application partitionnée, il faut se doter d'un modèle formel pour les différents éléments du programme : le graphe de contrôle de flot, les codelets, les accès en écriture, les accès en lecture et enfin les communications. Ces définitions font l'objet de la section

3.1.1.

Ces éléments définis, il est possible de définir notre problème d'un point de vue formel dans la section 3.1.3.

La mesure à optimiser est le surcoût global des communications dans l'application finale. Un soin particulier doit être apporté à la définition de la fonction de coût des communications intervenant dans le problème. Cette fonction est définie dans la section 3.1.2

Enfin, la définition et la mesure de la performance d'une solution étant clairement établie, la dernière section de cette partie définit les conditions de validité d'une solution et l'évaluation de sa qualité.

3.1.1 La représentation des éléments du programme

La problématique étant le placement optimal dans le flot de contrôle des communications, les algorithmes mis au point s'appuient sur la représentation du *CFG* du programme donné par la définition 3.1.

Définition 3.1 *Soit $G(V, E)$ un graphe de contrôle de flot G formé des éléments suivants :*

- *V l'ensemble des nœuds représentant les blocs de base, BB, du programme ; un bloc de base est un bloc d'instructions consécutives sans instruction de saut sauf éventuellement en dernière position.*
- *E l'ensemble des arcs $e(v1, v2), \{v1, v2\} \in V^2$, i.e. $v2$ est une cible possible du branchement à la fin de $v1$.*

Par définition et construction (cf. 2.2.2.1), le codelet est une fonction ayant un unique point d'entrée et un unique point de sortie dans le flot de contrôle. Sa représentation peut donc se résumer dans le GFG en un unique bloc de base k (et ceci en conservant la propriété éventuelle de réductibilité dont la définition et l'utilité sont définies dans la section 3.1.2.1). On obtient alors la définition 3.2 de la réduction du CFG pour un codelet donné.

Définition 3.2 *Soit K le sous-ensemble de V contenant les nœuds du codelet. On définit la restriction pour le codelet k du CFG $G(V, E)$ par un graphe $Gk(Vk, Ek)$ tel que :*

- *k un nœud représentant la réduction de l'ensemble K en un bloc unique*
- *Vk est l'ensemble des sommets de Gk tel que :*

$$Vk = (V \setminus K) \cup \{k\}$$

- *Ek est l'ensemble des arcs de Gk tel que :*
  - $\forall e(v1, v2) \in E,\ \{v1, v2\} \in V^2$, *si* $v1 \notin K,\ v2 \notin K$ *alors* $e(v1, v2) \in Ek$
  - $\exists! v2 \in K,\ \forall v1 \in \{v'_i \in (V \setminus K), e(v'_i, v2) \in E\} \Rightarrow (v1, k) \in Ek$
  - $\forall e(v1, v2) \in E,\ \{v1, v2\} \in V^2$, *si* $v1 \in K,\ v2 \in K$ *alors* $e(v1, v2) \notin Ek$
  - $\exists! v1 \in K,\ \forall v2 \in \{v'_i \in (V \setminus K), e(v1, v'_i) \in E\} \Rightarrow (k, v2) \in Ek$

Les communications entre le programme principal et le codelet doivent casser toutes les dépendances entre les accès en écriture à une zone dans le programme et les accès en lecture à cette même zone dans le codelet. Les cinq définitions 3.5, 3.6, 3.7 et 3.8, forment respectivement l'objet de la communication, les sources, la restriction des sources, les destinations et la restriction des destinations.

Définition 3.3 *Soit MEM l'ensemble des zones mémoire utilisées par le codelet. Chaque zone de l'ensemble correspondant au modèle défini dans* ASTEX*, elle peut être désignée par son identifiant unique.*

Définition 3.4 $\forall x \in MEM,\ EnsSRC(x) \subseteq V$ *est l'ensemble des nœuds de V représentant les blocs de base contenant un accès en écriture à la zone mémoire* x*.*

Définition 3.5 $\forall x \in MEM,\ EnsSRC(x,k) \subseteq EnsSRC(x)$ *défini par*

$$EnsSRC(x,k) = EnsSRC(x) \setminus K$$

Définition 3.6 $\forall x \in MEM,\ EnsDEST(x) \subseteq V$ *est l'ensemble des nœuds de V représentant les blocs de base successeurs de EnsSRC(x) contenant un accès en lecture à la zone mémoire* x*.*

Définition 3.7 $\forall x \in MEM,\ EnsDEST(x,k) \subseteq EnsDEST(x)$ *défini par*

$$EnsDEST(x,k) = EnsDEST(x) \cap K$$

Tous les éléments nécessaires à la modélisation des communications dans la restriction Gk du CFG ayant été établis, ces dernières peuvent être construites selon la définition 3.8.

Définition 3.8 *Une communication c est définie par les éléments suivants :*
- *id_zone : L'identifiant* ASTEX *unique représentant la zone mémoire à communiquer x ;*
- *origine : l'ensemble EnsSRC(x,k) ; la communication intervient nécessairement après le dernier élément exécuté ;*
- *cible : l'ensemble EnsDEST(x,k) ; la communication intervient nécessairement avant le premier élément exécuté ; dans la restriction Gk il s'agit de l'unique bloc k ;*
- *position : il s'agit de l'arc* $e \in Ek$ *qui porte la communication.*

L'ensemble des définitions énoncées dans cette partie est le point de départ de la définition du problème établi dans la section 3.1.3.

3.1.2 Définition de la fonction de coût d'une communication

Afin de calculer les surcoûts dus aux communications, il faut calculer la contribution de chacune d'entre elles au coût global. Dans cette section est construit l'équation donnant, dans le cadre de l'approche traitée dans ma thèse, la mesure du surcoût de communication d'une solution de placement. Cette mesure est discriminante, *i.e.* elle permet un classement relatif des solutions.

La mise en œuvre d'algorithmes basés sur cette mesure requiert plusieurs informations, dont certaines sont impossibles à déterminer statiquement. On suppose donc connues par profilage et analyse statique dans ASTEX les données suivantes :

- la probabilité $pb_{branch}(e)$: la probabilité d'exécution de l'arc $e(BB_o, BB_d)$ après BB_o,
- la probabilité $pb_{access}(z, BB)$: la probabilité qu'un accès en écriture à z intervienne dans BB,
- la longueur $lg(BB)$: la longueur en nombre d'instructions du bloc de base BB,
- la fréquence f : la fréquence d'exécution du bloc de base BB

Pour leur fonctionnement, les deux derniers algorithmes de ce chapitre ont besoin d'une valeur de coût de communication sur chaque arc et pour chaque zone à communiquer. On s'intéresse donc à l'élaboration d'une équation $Cost_{com}(z, e)$ correspondant au coût de communication de la zone z sur l'arc e avant k. Ce coût est fonction de quatre facteurs majeurs :

- la probabilité des accès en écriture à z intervenant avant le passage par k : afin de pouvoir adopter la politique de préchargmement en fonction de la probabilité d'un accès il faut déterminer la probabilité que le BB faisant potentiellement un accès a été exécuté et si l'accès a bien eu lieu avant le codelet ;
- la fréquence relative au codelet des arcs portant les communications et leur probabilité d'être exécutés, il s'agit du nombre moyen de fois où la communication sera lancée entre deux appels de codelet ;
- la distance du préchargement : plus la communication intervient tôt, plus la latence potentiellement masquée est grande. On s'intéresse donc à la distance probable de la communication à l'appel du codelet ;
- les pénalités de redondance : une communication peut être recouverte par une autre à la même zone ; quand on place une communication dans la solution, tous les arcs qui relient les blocs successeurs de la destination de l'arc qui porte la communication, sont sujets à un surcoût probable de recouvrement ; ce coût correspond à l'annulation de la communication en cours, avant de pouvoir la réexécuter avec les données mises à jour.

3.1.2.1 Probabilité et fréquence d'un arc

La fréquence d'un arc $e(BB_o, BB_d)$ est le produit de la fréquence de son origine par la probabilité que l'arc e soit exécuté, notée $P(e)$. Tous ces éléments font partie des éléments fournis par les données spéculatives en entrée.

La probabilité $prob_{exec}(e) = P(e) * P(BB_d|k)$ qu'un arc $e(BB_o, BB_d)$ soit exécuté avant k peut se calculer par un algorithme itératif de réduction. Ceci suppose que la

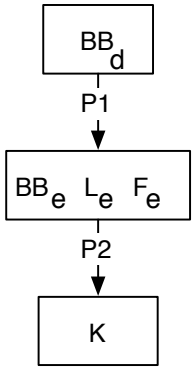

FIG. 3.1 – Cette figure présente le graphe réduit avec les règles de la figure 3.2. Le bloc équivalent noté BB_e a une longueur probable L_e et une fréquence F_e.

partie du CFG considérée pour la réduction est réductible.

Une fois l'algorithme déroulé, on obtient un graphe similaire à celui de la figure 3.1.

Graphe réductible et réduction

La définition 3.9 et le théorème 3.1 présentent les conditions de réductibilité d'un graphe de contrôle de flot [86]. La définition 3.9 est un critère de preuve général. Le théorème 3.1 présente le critère suffisant dans le cadre d'un graphe de contrôle de flot.

Définition 3.9 *Un graphe de contrôle de flot est réductible si et seulement si, il existe une partition de ses arcs en deux ensembles disjoints d'arcs avant et arrière définis ainsi :*

- *les arcs avants forment un graphe acyclique dirigé dont tous les nœuds peuvent être atteints depuis le nœud initial ;*
- *les arcs arrières sont ceux dont la destination domine l'origine.*

Théorème 3.1 *Le CFG d'un programme est réductible si il n'existe pas d'instruction de saut au milieu d'une boucle depuis l'extérieur.*

À partir des structures de contrôle dites structurées (*e.g.* Do-While, If-Then-Else, Continue, Break) il est possible de construire par induction un graphe réductible. L'instruction *Goto* de branchement inconditionnelle n'est pas structurée, cependant avec des tests stricts, son utilisation peut être tolérée. On détermine statiquement qu'elle ne rentre pas en conflit avec le théorème 3.1. Réduire le graphe revient à le « déconstruire » à partir de ses briques de base.

L'algorithme de réduction se base sur les éléments atomiques de la figure 3.2.

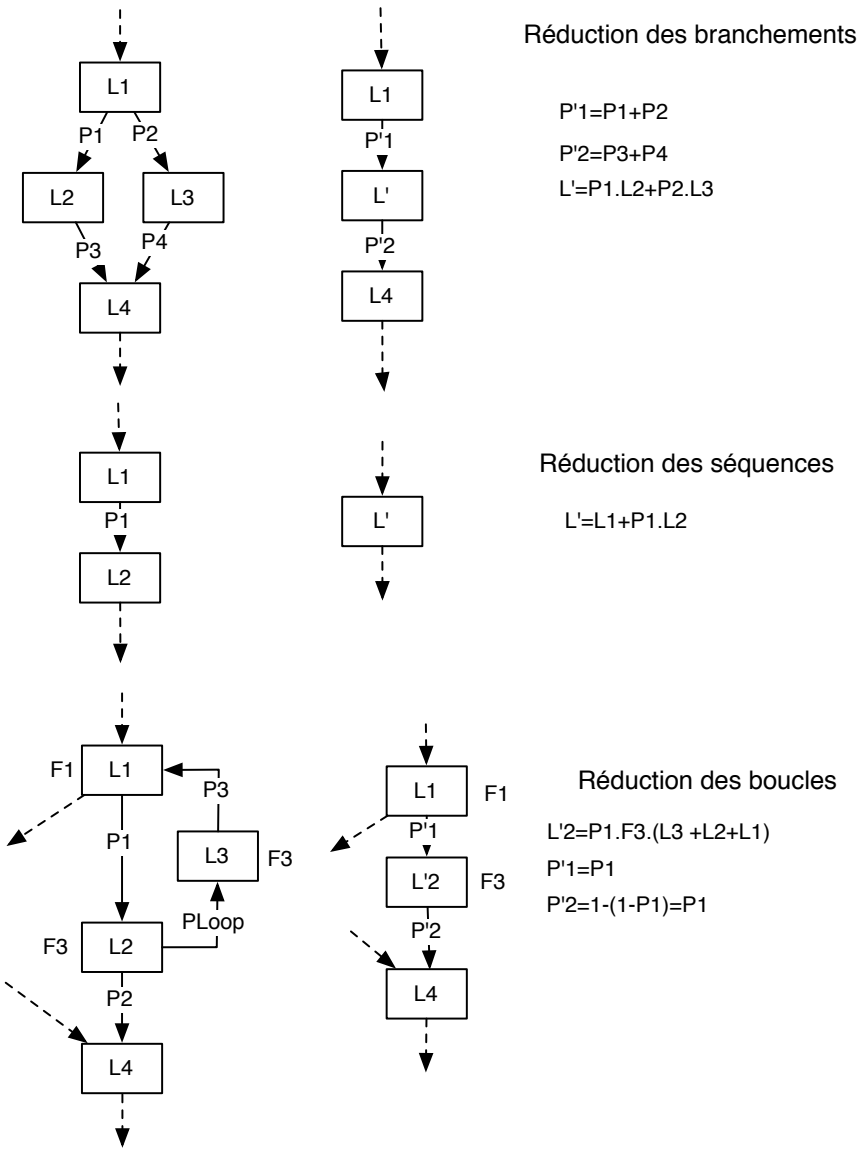

FIG. 3.2 – Ce schéma présente les trois règles de réduction permettant de ramener un graphe (réductible) à un bloc unique. La longueur moyenne probable du bloc équivalent ainsi que la probabilité des arcs sont tenues à jour à chaque étape de réduction.

Calcul par réduction de $prob_{exec}(e) = P(e) * P(BB_d|k)$

La première étape est de se restreindre au sous-graphe des chemins allant de BB_d à k. Pour cela, on supprime les arcs entrants dans BB_d, on supprime les arcs sortants de k et on ne conserve alors que les BB successeurs BB_d ayant k comme successeur.

La seconde étape consiste à réduire tous les chemins de BB_d à k à des chemins de longueur en bloc égale à 1 en appliquant les patrons de la figure 3.2.

La complexité du calcul par un algorithme de la probalité $prob_{exec}(e) = P(e) * P(BB_d|k)$ est $\mathcal{O}(n^2)$, n étant le nombre de nœuds du graphe ayant BB_o comme successeur.

Une solution plus générale peut être cherchée du côté d'outil comme les chaînes de Markov ou les réseaux bayésiens. Cependant le gain attendu ne justifie pas l'utilisation de ces outils complexes à ce stade de l'étude sur l'optimisation des communications.

3.1.2.2 Probabilité d'un accès en écriture avant un arc

Afin de pouvoir effectuer la classification de la section 3.2.1, on doit déterminer la probabilité qu'un accès soit exécuté avant le codelet et porte sur la zone observée.

Au sein d'un BB, par profilage, on connait la probabilité qu'un accès soit sur la zone spéculée. On note $Prob_{access}(z, BB)$ la probabilité d'accès (en écriture) à la zone z dans BB.

La probabilité qu'un accès à la zone z intervienne dans le bloc BB_i avant le codelet k est donnée par l'équation 3.1.

$$prob_{exec\_access}(z, BB_i) = prob_{access}(z, BB_i) * P(BB_i|k) \tag{3.1}$$

L'élément $P(BB_i|k)$ se calcule avec l'algorithme de la section 3.1.2.1.

3.1.2.3 Distance probable de préchargement

La définition 3.10 est la définition de la distance moyenne probable entre deux blocs adaptée à notre fonction de coût.

Définition 3.10 *La distance probable moyenne, Pdistance, entre deux blocs BB_i et BB_j du CFG est la somme des longueurs en instructions (e.g. asm x86) des blocs, pondérée par le produit des probabilités des arcs sur chaque chemin de BB_i à BB_j.*

Le calcul de la distance probable de préchargement dans le cas d'un graphe réductible suit le même algorithme que celui du calcul de la probabilité donné dans la section 3.1.2.1.

La distance probable finale est, dans la figure 3.1, la longueur L_e du bloc équivalent BB_e pondéré par la probabilité $P1$ plus la longueur du bloc BB_d.

3.1.2.4 Coût unitaire d'une communication

Le coût d'une communication se décompose en deux éléments :
- la latence, *latency*, de mise en place de la communication qui est un coût incompressible,
- le coût unitaire de communication d'une unité U de mémoire (l'octet par exemple).

Le coût de la communication $c(z_i, BB_i, k, e(BB_s, BB_d))$ de taille $z_i.size$ est donc donné par l'équation 3.2.

$$C_{abs}(c) = latency + c.z_i.size * cost_{1U} \tag{3.2}$$

La communication $c(z, BB_i, k, e(BB_s, BB_d))$ intervient avec une probabilité $P(e) * prob_{exec\_access}(BB_s|k)$ et masque une latence moyenne proportionnelle à la $Pdistance$ de BB_d à k. Initialement on a $BB_s = BB_i$, *i.e* le préchargement est placé sur les arcs sortants du BB faisant l'accès.

On définit alors le surcoût probable d'une communication par son coût pondéré de sa probabilité et inversement proportionnel à sa $Pdistance$. Ainsi plus une communication est loin de sa cible et exécutée peu de fois, plus son surcoût est faible. La formule du surcoût d'une communication est donnée par l'équation 3.3.

$$Cost\_com(c) = \frac{P(c.e) * prob_{exec\_access}(c.e.BB_s \mid k) * C_{abs}(c)}{Pdistance(c.e.BB_d, k)} \tag{3.3}$$

3.1.2.5 Pénalité de recouvrement

Le dernier surcoût entrant dans l'évaluation du surcoût est celui dit de recouvrement. En effet si un préchargement est lancé alors qu'un autre intervient sur la même zone plus tard dans l'exécution, il faut annuler le préchargement courant, ce qui peut potentiellement impliquer une pénalité. Si la destination d'un arc candidat pour porter un préchargement $c1$ avec $c1.e = e(BB1_o, BB1_d)$, a dans ses successeurs la source d'un arc portant un préchargement $c2$ avec $c2.e = e(BB2_o, BB2_d)$ alors il faut évaluer la probabilité de recouvrement de la communication $c1$ par $c2$ notée $P_{over}(c1, c2)$ et donnée par l'équation 3.4.

$$P_{over}(c1, c2) = P(c1.e) * P(BB2_o|BB1_d) * P(c2.e) \tag{3.4}$$

La probabilité P_{over} se calcule avec le même algorithme de réduction que dans la partie 3.1.2.1.

Un algorithme optimisé gardant en mémoire les formes réduites des différents sous graphes du CFG est alors souhaitable afin de maîtriser la complexité de l'évaluation du coût de préchargement.

Avec un coût de pénalité constant $cost_{over}$, on donne au surcoût de recouvrement de la communication c sur e la valeur de l'équation 3.5.

$$Penalite_{over}(c) = cost_{over} * \sum_{c_i \in C} P_{over}(c.e, c_i.e) \tag{3.5}$$

3.1.2.6 Évaluation du surcoût global

Le surcoût global de communication dans l'application est le coût dynamique probable de l'ensemble des points de préchargement choisis. À chaque communication sont associés d'après les sections précédentes :

- une probabilité d'exécution du préchargement,
- un coût de communication,
- une Pdistance,
- une pénalité probable de recouvrement.

Pour chaque communication c, on a défini un coût $Cost\_com(c)$ qui donne le coût dynamique probable d'une communication sur un arc. Il s'agit de l'équation 3.3.

L'évaluation du surcoût dû à l'ensemble C des communications du programme est donnée par l'équation 3.6

$$Cost(c) = \sum_{c \in C} Penalite_{over}(c) + Cost\_com(c) \tag{3.6}$$

Dans les algorithmes des sections suivantes, le poids d'un arc $e_{candidat}$ est pour une communication c : $Cost\_com(c|c.e = e_{candidat})$

Ainsi ne varie pendant l'algorithme que la partie pénalité de recouvrement du coût de la solution, fonction des arcs retenus.

3.1.3 Définition du problème

La définition du problème reprend les éléments de formalisme de la section 3.1.1. Afin de s'entendre sur la définition du problème il convient de rappeler les éléments et définitions suivantes :

- $EnsSRC(x, k)$ est l'ensemble des BB extérieurs au codelet contenant un accès en écriture sur la zone x de mémoire à communiquer ;
- Une coupe dans un graphe est un ensemble d'arcs qui sépare le graphe en deux parties telles que tout arc ayant une extrémité dans S et l'autre dans $\overline{S}$ est dans cette coupe, *i.e.* tout chemin d'un élément de S à $\overline{S}$ contient un arc dans la coupe ;
- Le graphe $Gk(Vk, Ek)$ est le CFG $G(V, E)$ du programme avec tous les BB du codelet réduit en un seul nœud K ;
- $W(e_i) = Cost\_com(c|c.e = e_i) + Penalite_{over}(c)$, fonction de E_k dans $\mathbb{R}^+$, est la fonction de coût d'une communication sur un arc donné.

La figure 3.3 est une copie modifiée de celle donnée dans le chapitre 1. Elle illustre la définition formelle 3.11 du problème d'optimisation du placement des communications. Seuls les arcs de S vers $\overline{S}$ portent des communications et entrent dans le calcul du surcoût de communication.

Définition 3.11 *Le placement optimal valide des communications pour une zone x dans le graphe $Gk(Vk, Ek)$ muni de la fonction de coût $Cost\_com : E_k \rightarrow \mathbb{R}^+$ est l'ensemble $Com\_Cut \subset E_k$ tel que :*

- *Com_Cut est une coupe du graphe G_k, S et $\overline{S}$ les deux parties résultantes ;*

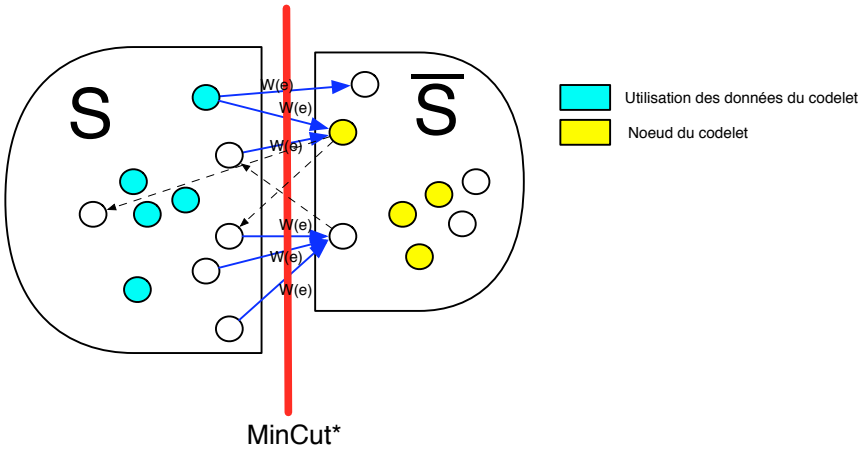

FIG. 3.3 – Le placement des communications de coût minimal entre un codelet et les accès aux variables accédées par le codelet est l'ensemble des arcs de S à $\overline{S}$ dont la somme des poids est minimale. Trouver ce placement revient à trouver l'ensemble S optimal.

- $\forall BB_i \in EnsSRC(x,k),\ BB_i \in S$;
- $k \in \overline{S}$;
- $\sum_{e_i(BB_o,BB_d)\in Com\_Cut, BB_o\in S, BB_d\in\overline{S}} W(e_i)$, *la somme du poids des arcs de* S *vers* $\overline{S}$ *est minimale.*

Outre le cas particulier du graphe dirigé, la différence essentielle de notre problème d'optimisation du placement des communications avec celui classique du MinCut/MaxFlow, est l'influence sur le poids des arcs du recouvrement probable d'une communication par une autre. Sans cela, il existe des solutions optimales en un temps polynomial [83, 84]. Prenant en compte cette réentrance, les suites convergentes des algorithmes itératifs donnant une solution en un temps polynomial perdent leur critère de terminaison [83].

3.1.4 Définition d'une solution

Deux critères permettent de qualifier une solution. Il s'agit du critère nécessaire de validité, et celui souhaitable de qualité en terme de surcoût des communications de la solution calculée.

Dans une première section, cette partie présente la définition formelle du critère de validité d'une solution.

Dans un second temps est discuté la mesure de la qualité d'une solution, nonobstant la complexité de calcul de cette dernière.

3.1.4.1 Condition de validité d'une solution

La définition 3.12 est celle d'une solution valide au problème d'optimisation des communications. Elle traduit l'intuition qu'une solution, pour être valide, doit garantir que quelque soit le chemin suivi à l'exécution entre un accès et le codelet, la donnée sera mise à jour.

Définition 3.12 *Soit Com_Cut une solution valide au placement des communications dans un CFG G(V,E) ayant k comme codelet destination et EnsSRC comme ensemble d'accès en écriture. Com_Cut valide la condition suivante :*

$$\forall x,\ (\forall BB_i \in EnsSRC(x)),\ \forall \text{ chemin de } BB_i \text{ à } k,\ \exists BB_{com} \in Com\_Cut$$

Dans cette définition, Com_Cut n'est pas nécessairement l'unique coupe du graphe contenant l'ensemble des communications pour toutes les solutions valides, il s'agit de la coupe minimale. Dire que Com_Cut est une coupe de G pour BB_i et k n'est donc pas une condition de validité minimale, *i.e* que Com_Cut ne soit pas une coupe de G pour BB_i et k n'implique pas que la solution testée n'est pas valide. La figure 3.4 présente un contre exemple trivial. Afin de rendre cette condition minimale, il faut ajouter que tous les accès doivent appartenir à S, on retrouve alors notre définition du problème. Ceci revient à construire la solution pour prouver sa validité. Dans le cadre d'un problème complexe où l'on fait usage d'algorithme à heuristique, ce n'est évidemment pas acceptable.

Pour tester une solution, l'algorithme consiste donc à faire un parcours acyclique en largeur à partir de chaque accès qui s'arrête si on croise un élément de la solution. Si cet algorithme termine sans trouver la valeur k, c'est que tous les chemins étaient coupés et que par conséquent la solution est valide.

3.1.4.2 Évaluation de la qualité d'une solution

N'ayant pas d'algorithme permettant de construire une solution optimale (à l'exécution), on ne peut pas donner de mesure absolue d'une solution.

Cependant, certaines expérimentations basées sur le profilage des derniers points réels de modification des zones accédées par le codelet, on peut produire une valeur spéculative de la solution optimale et s'y comparer. Une campagne de tests statiquement valides permettrait alors de qualifier dans l'absolu la solution choisie. On fait face au problème complexe de la génération de jeux de données pour le test de programme.

Par mesure réelle, on peut valider le modèle de coût. Une fois ce modèle de coût validé, la comparaison statique des différentes solutions entre elles est possible, grâce à l'évaluation du coût probable des communications. On peut alors choisir parmi les différents algorithmes de construction d'une solution ceux dont l'implantation réelle est potentiellement parmi les plus efficaces.

L'expérimentation finale comparant les résultats réels n'est pas encore possible pendant la rédaction de ce mémoire. La validation des algorithmes n'est donc dans ce chapitre que théorique.

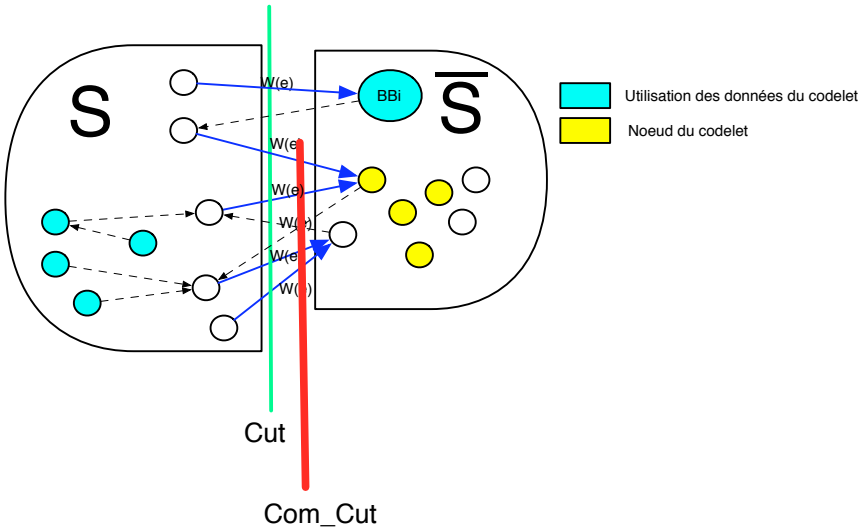

FIG. 3.4 – Dans cet exemple synthétique, *Cut* est une solution valide au problème du placement des communications dont tous les chemins allant d'un accès au codelet passent par un élément de *Com_Cut* sans pour autant avoir $Cut = Com\_Cut$.

Pour le calcul des probabilités, on utilise les données spéculatives fournies par ASTEX. Par conséquent, la fonction de coût utilisant ces probabilités est représentative du coût dynamique des communications dans le programme. Il s'agit là d'un point clef pour le développement d'algorithme efficace de placement des communications.

Le problème de l'évaluation du coût dynamique se posant aussi dans le cadre du placement des barrières de synchronisation [18, 19], il serait éventuellement intéressant de transposer dans ce cadre l'approche consistant à utiliser les données spéculative d'ASTEX.

3.2 Intégration des données spéculatives

Les algorithmes présentés dans ce chapitre se basent sur une analyse statique et des informations dynamiques provenant des profils fournis par ASTEX pour optimiser le placement des communications. Les données dynamiques sont spéculatives. Pour chaque paramètre du codelet extrait, tous les accès possibles avant chaque appel du codelet peuvent se caractériser par un statut statique, un statut dynamique et une fréquence.

Dans un premier temps est décrit dans le section 3.2.1 une classification des accès en fonction de leurs statuts statiques et dynamiques. Le statut dynamique est la traduction des données spéculatives d'ASTEX.

Dans un second temps est expliquée la méthode de calcul des coûts initiaux (*i.e.* le poids des arcs) à partir des données spéculatives de fréquence, de probabilité et de taille des données à communiquer observées par ASTEX.

3.2.1 Classification des accès

Les statuts statiques et dynamiques sont résumés dans le tableau 3.1. Tous les accès sont caractérisés par un doublet (S,D).

S	Statut Statique	D	Statut Spéculatif
1	Valeur accédée	1	Valeur accédée
2	Valeur non accédée	2	Accès à la valeur non vu
3	Indéterminé	3	Pas d'information

TAB. 3.1 – Tableau présentant les différentes valeurs du couple (S,D) des accès, et le statut correspondant.

On peut classer les accès en fonction de (S,D) et appliquer une politique d'optimisation adaptée.

1. {(1, 1 ou 3) ;(3,1)} : il y a un accès sûr ou probable, on doit effectuer une communication ;
2. {(3,2) ;(3,3)} : il peut y avoir un accès mais il est peu probable ou improbable, l'optimisation de ces accès est non prioritaire ;
3. {(2, 2 ou 3)} : il n'y a pas d'accès.

(1,2) et (2,1) sont des combinaisons impossibles.
Dans la suite de ce chapitre, chaque classe peut être identifiée par son numéro dans la liste précédente. La classe 1 représente les accès sûrs ou probables, la classe 2 les accès improbables mais possibles et enfin la classe 3 les accès impossibles.

3.2.2 Calcul des coûts initiaux

Dans les cas où on implante un algorithme nécessitant une fonction de coût, les données spéculatives fournies par ASTEX complètent avantageusement un modèle statique. À partir de ces données, on peut calculer, hors surcoût de recouvrement, la contribution au coût de communication d'un arc si ce dernier est sélectionné dans la solution finale. Une fois ce surcoût initial calculé, est tenue à jour lors de l'exécution de l'algorithme d'optimisation la contribution du recouvrement au surcoût.

Le calcul des coûts initiaux se fait selon le modèle de coût de la section 3.1.2.

Le réalisme du modèle de coût, outre les intrinsèques de construction, dépend de la qualité des données spéculatives et donc des jeux de tests.

3.3 Algorithmes d'optimisation du placement des communications

Dans cette partie sont décrits trois algorithmes utilisant les éléments définis dans les parties précédentes. Le premier est une approche directe hors formalisme, le second une

approche issue de la définition formelle du problème, et enfin le troisième une approche hybride usant d'éléments du formalisme comme la fonction de coût afin de réduire la force des heuristiques adoptées dans le premier algorithme.

La section 3.3.1 présente un algorithme basé sur une interprétation intuitive direct de l'assertion « au plus tôt et le moins de fois ». Il est basé exclusivement sur la classification des accès, le probable recouvrement et des heuristiques fortes sur les points possibles de communication. Il s'agit de déplacer les communications en dehors des boucles et de supprimer les redondances. On évalue pas de fonction de coût.

Le second algorithme est un dérivé de Ford-Fulkerson. Notre problème présentant des similitudes troublantes dans sa définition avec le MinCut/MaxFlow, une adaptation d'une solution polynomiale à ce problème est définie dans la section 3.3.2

Enfin, le dernier algorithme est un algorithme à heuristique proche du premier mais basant ses mouvements de points de communication sur une fonction de coût et plus seulement sur les structures de contrôle seules.

Les différents placements ne peuvent être redondants que s'ils accèdent la même zone de mémoire ; par conséquent, l'algorithme de placement peut se décomposer dans le traitement de groupe d'accès à une même zone de mémoire. Ceci sans influence sur le résultat final. Si une expression d'accès peut adresser différentes zones mémoire, elle est considérée une fois pour chaque zone mémoire qu'elle peut accéder. Ceci autorise un découpage trivial du problème, ce qui diminue sa complexité globale (inégalité entre la somme de puissance et la puissance d'une somme).

Ce découpage est appliqué dans les trois algorithmes présentés. La réunification est présentée dans la section 3.4.1.

3.3.1 Algorithme direct

Cet algorithme est une version retravaillée de celui publié dans algo [41].

Le préchargement de données a pour objectifs :

- de transférer le plus tôt possible les données au coprocesseur ;
- d'éviter de charger à nouveau les données déjà à jour dans le coprocesseur, on ne charge que les données modifiées entre deux appels ;
- de préserver la bande passante de la mémoire et d'éviter d'éventuelles pénalités, des chargements multiples sur les mêmes données.

L'intuition est donc de répondre point par point au trois objectifs précités :

- placer les communications le plus tôt possible après l'accès,
- placer les communications en dehors des structures de répétition, *i.e.* les boucles,
- les rassembler si elles ont de grandes chances d'être redondantes.

3.3.1.1 Définition des politiques de préchargement

Connaissant la probabilité qu'un accès intervienne, et connaissant la probabilité que la communication associée soit recouverte, on choisit l'une des trois politiques de préchargement de la liste suivante :

- Préchargement de type ASAP : l'objectif quand un accès est sûr ou très probable est d'effectuer le préchargement au plus tôt et le moins de fois possible. Dans mon approche publiée dans [41], il est fait l'hypothèse que ce point ASAP est le premier point après les structures de contrôle englobant l'accès mais pas le codelet.
- Préchargement de type GUARD : si la probabilité d'un accès est non négligeable mais avec une probabilité inférieure à un seuil prédéfini, on ajoute un test, éventuellement optimisé, s'assurant que l'accès est effectif avant de communiquer. Au niveau du placement, cette solution est équivalente à ASAP, seul le code généré diffère par l'insertion d'une garde.

Quand le préchargement d'un paramètre a de grandes chances d'être redondant (recouvert à l'exécution par un autre), alors il faut grouper ces deux préchargements et couper les chemins restants avec une politique ASAP. Il n'y a pas besoin, dans ce cas, d'une politique de chargement différente contrairement à ce qui avait été écrit dans [41].

3.3.1.2 Algorithme

L'algorithme proposé fonctionne en trois étapes. La première initialise le placement des communications relativement à leur classification. La seconde déplace les préchargements en dehors des boucles et supprime ceux devenus redondants. La dernière étape, dite de raffinement, regroupe les préchargements probablement redondants et duplique le préchargement initial pour fermer les chemins peu probables restant. L'algorithme est détaillé sur toutes ses étapes dans la suite de cette section.

On itère ensuite entre les deux dernières étapes jusqu'à l'obtention d'un point fixe.

La figure 3.5 présente un exemple de déroulage de l'algorithme.

1. Initialisation La première étape présentée dans la colonne de gauche est l'initialisation. Les accès classés 1, *i.e.* dans $\{(3,1);(1,1)\}$, héritent d'une politique ASAP. Les accès classés 2, *i.e.* dans $\{(3,2);(3,3)\}$, héritent d'une politique GUARD.

2. Simplification La seconde étape déplace un point de communication en dehors d'une boucle. On supprime alors toutes les communications devenues redondantes et ce quelque soit leur politique. La communication $C1$ est redondante avec $C2$ si $C1$ a $C2$ comme prédécesseur dans le graphe de contrôle de flot inverse (*i.e.* où le sens des arcs est inversé). On parle de postdominateur.

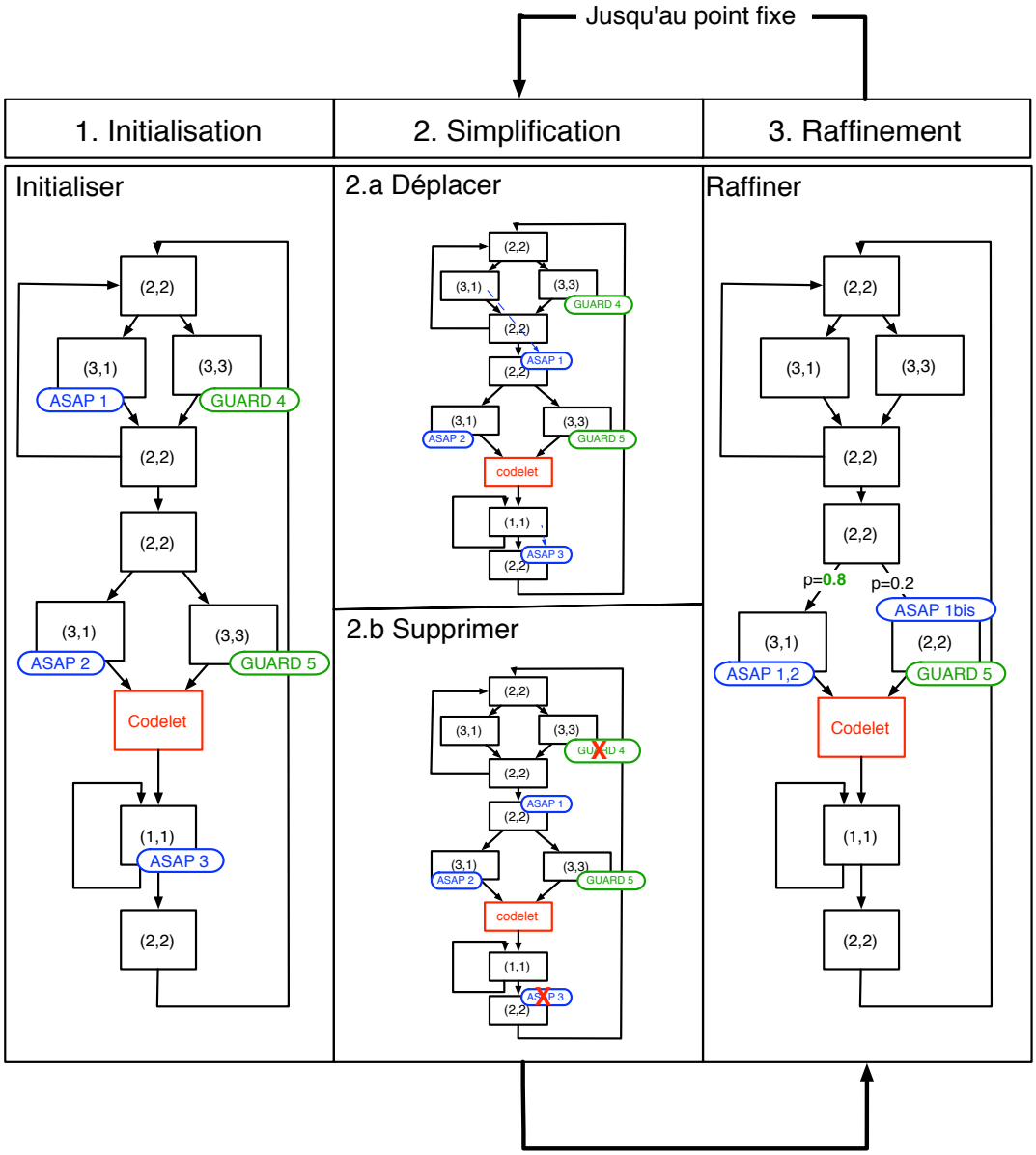

FIG. 3.5 – Exemple illustrant le déroulage de l'algorithme direct d'optimisation des communications.

Dans la figure 3.5, l'étape de simplification correspond à la colonne du milieu. Une fois la communication $ASAP1$ du graphe exemple déplacée en dehors de la boucle qui le contient, $GUARD4$ et $ASAP3$ deviennent redondants, on les supprime dans l'étape 2.*b*.

3. Raffinement La dernière étape, afin d'éviter un maximum de redondance à l'exécution, fusionne les communications avec celles qui probablement les recouvrent.

Afin de garder la condition de validité assurant que tous les chemins d'un accès au codelet donnent lieu à une communication, les chemins ne menant pas au point de fusion sont coupés au plus tôt par une communication ASAP.

Dans l'exemple de la figure 3.5, cette étape est représentée dans la colonne de droite. $ASAP1$ a 80% de chance d'être redondante avec $ASAP2$. Ce cas trivial de calcul de probabilité se généralise en utilisant l'algorithme de la section 3.1.2.1 afin de calculer $P(ASAP2|ASAP1)$. Dans l'exemple, cette probabilité est estimée forte. On fusionne $ASAP1$ et $ASAP2$. On ajoute au plus tôt une communication $ASAP1bis$ sur le chemin laissé libre.

Rebouclage Tant qu'il y a un préchargement dans une boucle différente d'une boucle englobant le codelet, on le déplace. Au pire, tous convergent vers le codelet ou du moins le premier prédecesseur du codelet successeur de tous les points d'accès. L'algorithme a donc nécessairement un point fixe.

L'algorithme présenté ici paye sa simplicité relative due aux heuristiques fortes (*e.g.* pas de communication dans les boucles) par un certain nombre d'inconvénients.

- L'absence de fonction de coût ne permet pas l'évaluation statique de la qualité d'une solution ;
- Le seuil de probabilité pour juger de l'importance de la redondance est fixe ;
- Des placements potentiellement optimaux sont ratés (*e.g.* boucle avec peu d'itérations mais un corps très long en temps d'exécution pourrait amortir une pénalité). Un déroulage du dernier tour de chaque boucle basé sur des données spéculatives (quand c'est possible) pourrait éventuellement amoindrir ce problème ;
- Dans certains cas, en particulier si le contrôle est complexe, les communications auront toutes tendance à glisser dans le voisinage immédiat du codelet, ce qui est contre-productif. L'exemple de la figure 3.5 en est une illustration.

3.3.2 Adaptation de Ford-Fulkerson

L'utilisation de Ford-Fulkerson a déjà été abordée dans la section 1.5.3. Une fois l'algorithme déroulé, l'ensemble MinCut forme l'ensemble des arcs portant une communication.

Pendant l'algorithme de Ford-Fulkerson, afin de conserver sa condition de terminaison, le poids des arcs déjà saturés n'est pas réactualisé, ce qui revient à ignorer une partie des redondances. La solution peut, dans certains cas où l'heuristique de convergence n'est pas utilisée, être optimale.

Pour les autres redondances, contrairement à l'algorithme de la section 3.3.1, la redondance est traitée implicitement et de manière exacte par l'utilisation de la fonction de coût. Il n'y a plus de phénomène de seuil, la réponse est graduelle.

Par ailleurs, aucun arc n'est favorisé pour le placement d'une communication. Si le coût estimé à partir des données spéculatives est moindre, rien n'empêche une communication de prendre place dans une boucle et éventuellement d'être redondante.

Cependant, il y a des cas où la solution est sous-optimale. Si on ignore une redondance affectant un arc déjà saturé, le coût de la solution est faussé. Afin de minimiser les occurrences de ce problème, on peut choisir pour la construction de la série des chemins augmentant une série particulière. L'algorithme de Dikjstra [84] pour construire le plus court chemin non saturé est un choix pertinent. Partir du plus court chemin entre le codelet vers le plus long pour la saturation permet de réduire la probabilité d'occurrence du problème. En effet, avoir moins de successeurs aux communications déjà placées tend à diminuer la probabilité qu'une communication soit placée parmi eux. Par ailleurs, on favorise l'optimisation des chargements les plus proches du codelet en les traitant de manière prioritaire.

3.3.3 Algorithme à heuristique et fonction de coût

Afin de répondre au problème soulevé par l'algorithme de la section 3.3.1 tout en conservant une complexité raisonnable, un autre algorithme à heuristique intégrant la modélisation du coût et s'abstrayant des boucles est développé dans cette partie.

Bien que produisant des solutions théoriquement raisonnables, la complexité de l'adaptation de l'algorithme de Ford-Fulkerson et l'absence de garantie sur la qualité d'une solution, en particulier en présence de redondance, motivent l'élaboration d'une solution intermédiaire.

Dans la première section de cette partie est abordée l'étape d'initialisation de l'algorithme, en particulier la simplification nécessaire du CFG et ses conséquences.

Dans un second temps est décrit l'algorithme de placement.

3.3.3.1 Initialisation

Afin de pouvoir placer des communications, y compris dans les boucles, et de pouvoir traiter la redondance, l'algorithme présenté dans cette partie simplifie le graphe de contrôle de flot. La diminution du nombre de nœuds et d'arcs amène par ailleurs un gain appréciable quant au coût de l'algorithme malgré sa complexité.

Cette section présente dans l'ordre les étapes d'initialisation de l'algorithme.

1. Comme pour les autres algorithmes, on commence par la classification des accès. Tous les accès de catégorie 3 de la section 3.2.1 sont exclus.
2. On supprime les arcs sortant du BB codelet. Ceci a pour effet de « casser » les boucles englobant le codelet. Par ailleurs, est considérée comme redondante une communication intervenant plusieurs fois entre deux appels au codelet ; considérer les boucles englobant le codelet est une erreur. Enfin, aucune communication ne

peut être portée par ses arcs car ils dominent tout le code pouvant être exécuté entre deux appels au codelet, ce qui inclut les accès.

3. On supprime les nœuds qui n'ont pas le BB codelet dans leur successeur. Cette simplification est évidente, si le codelet n'est pas exécuté après un bloc ; charger les données est inutile.
4. On calcule les coûts initiaux des arcs par réduction ainsi que la matrice des distances probables de BB à BB. Les arcs contenus dans des boucles ne contenant pas le codelet sont considérés comme auto-redondants. Encore une fois, un perfectionnement consisterait à se baser sur les données spéculatives pour dérouler le dernier tour de boucle quand cela est possible. Ainsi, ce facteur de redondance inévitable sans déroulage pourrait disparaître.
5. On inverse le CFG (on retourne les arcs) et supprime tous les arcs arrières encore dans le graphe en considérant le BB du codelet comme origine du graphe conformément à la définition 3.9. En inversant à nouveau le graphe, on a conservé tous les chemins acycliques qui mènent d'un accès éventuel au codelet.
6. Si tous les chemins d'un accès A au BB codelet croisent un accès sûr ou probable alors A est supprimé (redondance évidente). A domine B dans le graphe inverse.
7. On place initialement les communications sur les arcs issus des BB avec accès sûr ou probable.

À ce stade, tous les arcs sont descendants, il n'y a plus de cycle dans le graphe, seul le facteur de redondance dans le calcul du coût va varier lors de la suite de l'algorithme. La figure 3.6 reprend celle de la section 1.5. Elle présente la forme du graphe obtenu à cette étape. Le trait rouge est un exemple de coupe possible. On cherche alors la coupe minimale.

3.3.3.2 Algorithme

Une fois les différentes valeurs initialisées et le graphe simplifié, l'optimisation itérative du placement commence. Les étapes suivantes forment dans l'ordre l'algorithme d'optimisation.

1. Pour toutes les communications non placées ;
2. On commence par la communication non placée de coût le plus élevé (heuristique) ;
3. On cherche en profondeur (recursivement) une position moins coûteuse (en cas de branchement elle se duplique) ;
4. Si on croise une autre communication, on les fusionne temporairement et on continue la descente ;
5. À l'issue de cette descente, on place la communication dans la position la moins coûteuse trouvée ;
6. Si cette position est différente de la position initiale, toutes les communications placées redeviennent non placées et les valeurs de coût sont remises à jour (redondance éventuelle) ;

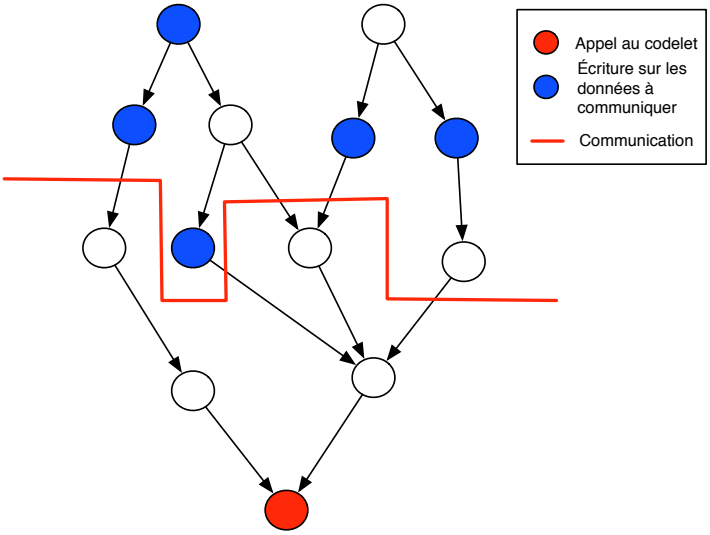

FIG. 3.6 – Après simplification dans l'algorithme 3.3.3, le *CFG* d'une application devient unilatéral et acyclique. Le noeud rouge est le point d'appel au codelet. Les noeuds bleus modifient les zones mémoire nécessaires au codelet. On cherche dans la suite de l'algorithme la géométrie de la ligne rouge.

7. On recommence jusqu'au point fixe.

On ne déplace, dans un premier temps, que les communications issues d'accès sûr ou probable.

L'algorithme converge car le graphe, après simplification, est acyclique, unilatéral et fini (on ne peut que « descendre » et il y a une position « plancher » qui est le BB codelet k).

On lance ensuite l'algorithme sur les communications peu probables ; si dans la descente on croise une communication sûr ou probable redondante, la fusion est immédiate. Pour les communications restantes on génère à leur position la moins coûteuse la garde permettant de choisir si oui ou non elles doivent être exécutées.

3.4 Placement final des communications

Tous les algorithmes précédents fournissent en sortie une liste de couples communication-arc afin de permettre leurs translation dans le code original, et ce pour chaque zone mémoire considérée.

Dans un premier temps, après le placement des communications, peut intervenir une dernière optimisation, dite de *coalesing*, permettant de regrouper, si c'est intéressant, les communications à des zones différentes. En particulier si elles sont adjacentes en mémoire. Cette optimisation est détaillée dans la section 3.4.1.

Enfin la solution sur le CFG est translatée en instruction ou pragma réel dans le code source. Dans notre cas, il s'agit de pragma HMPP [42]. Cette translation est l'objet

de la section 3.4.2.

3.4.1 Groupement des communications

Le placement des communications ayant été mis en place indépendamment pour chaque zone mémoire, certains arcs portent des communications sur plusieurs zones différentes, certaines communications de zones mémoire adjacentes peuvent être dans un voisinage proche. Rassembler les communications permet de mutualiser le surcoût de mise en place des communications et éventuellement d'exploiter leur localité si l'architecture y est sensible.

Dans un premier temps, on évalue grâce au modèle de coût, sur le chemin de l'accès au codelet, la possibilité pour une communication d'être groupée avec les communications successeurs à des zones adjacentes. On part de la communication la plus proche vers la plus éloignée du codelet selon le critère de la distance probable. Si une position de coût inférieur est retenue, on groupe. Partir de la communication la plus proche vers la plus éloignée permet de traiter tous les regroupements en une seule passe.

Si, sur l'architecture cible, il y a un intérêt éventuel à regrouper des communications sur des zones non adjacentes, on reproduit dans un second temps le même algorithme. Afin de raffiner ce traitement, on pourrait éventuellement étudier la possibilité de modifier l'allocation dans la mémoire du processeur principal afin d'obtenir un ordre d'allocation qui favorise la localité pour les communications.

3.4.2 Translation de la solution dans le programme original

Une fois l'ensemble des arcs du CFG portant une communication déterminé, il faut transcrire ce placement dans le code source original de l'application.

La section 2.4.1.3 montre l'utilisation des directives de préchargement HMPP dans le code C original. L'implantation physique étant la charge de l'outil HMPP, seul l'emplacement du point de communication reste à déterminer.

Le choix retenu est l'implantation en tête du BB destination de l'arc portant la communication. En effet, dans le cas où plusieurs arcs sortants d'un bloc portent une communication, un seul des chemins sortants d'un bloc est exécuté (sauf optimisation particulière liée aux exécutions spéculatives). Par conséquent, seule une des communications est exécutée.

Le placement en queue de bloc prédécesseur avant le branchement final n'est pas sûr dans le cas général. Si un accès à la zone mémoire cible de la communication est effectué dans la condition de branchement final du BB prédécesseur, on ne peut pas insérer la communication avant. La possibilité de traiter ce cas particulier avec un gain supérieur au surcoût éventuellement induit semble restreinte. Sauf cas particulier, il est préférable d'utiliser la solution d'insertion en tête du bloc successeur.

3.5 Premières expérimentations : le cas du "GPGPU"

Le logiciel ASTEX n'est pas encore prêt à recevoir l'implantation de l'optimisation des communications. Seules sont calculées aujourd'hui les entrées-sorties spéculatives des codelets. L'instrumentation des chemins et accès mémoire inter-codelet est une étape importante dans la suite du développement de l'outil. Ces premières expérimentations ont pour but d'explorer à la main l'influence des communications et surtout de leurs possibles optimisations. En particulier, il s'agit de déterminer dans le cadre du GPGPU l'influence des trois axes d'optimisation :

- charger les données au plus tôt,
- éviter les copies redondantes entre deux appels,
- ne recopier que la partie modifée de la mémoire entre deux appels du codelet.

Pour ces expérimentations préliminaires, l'application de test choisie est l'application *FT* des *NAS parrallel benchmark*. Il s'agit d'un calcul de FFT3D implanté à l'aide de trois FFT 1D. On effectue l'implantation sur GPU. On fait ensuite varier la taille du problème (dimension de la matrice cubique de 16 à 256) et la méthode d'optimisation.

Le GPU utilisé est une carte NVIDIA 8800 avec 1Go de mémoire principale. La machine hôte est un Pentium D à 2.80GHz. La connexion entre le CPU et le GPU est de type PCIe 8x. Dans les deux cas d'optimisation le codelet est implanté à l'aide de CUDA [61]. ASTEX est utilisé pour détecter le codelet principal. Il s'agit de la fonction $fftz2$ qui représente 67% du temps total d'exécution de l'application pour une taille de matrice en entrée de 256 x 256 x256.

L'objectif n'est pas de se comparer à d'autres implantations spécialisées de la FFT sur GPU mais de mesurer le gain lié à l'optimisation des communications.

On utilise deux niveaux d'optimisation notés *opt* et *opt*2 :

- *opt* : le déroulage manuel de l'algorithme présenté dans [47] ; dans le cas de l'application FT, il ne présente pas de différence avec celui de la section 3.3.1 ; l'optimisation reste locale, c'est-à-dire dans la fonction faisant l'appel au codelet.
- *opt*2 : l'optimisation manuelle et inter-procédural , *a priori* optimale, des communications ; elle permet de mesurer la performance absolue des différents algorithmes présentés.

Pour chaque niveau d'optimisation sont optimisés le volume des données à copier et le préchargement.

La sous-section 3.5.1 présente les résultats obtenus.

3.5.1 Résultats

La figure 3.7 représente, pour des tailles restreintes de problème (16 à 64), le temps d'exécution par rapport au temps de référence (CPU) des différentes versions de FT sur GPU. Sans optimisation, les surcoûts des communications restent toujours supérieurs au temps d'exécution sur CPU seul, ce qui n'autorise aucun speed-up. On obtient même un facteur de ralentissement de 37.

Le CPU charge lors de l'exécution les données de la mémoire principale ; son temps de calcul, chargement mémoire compris, est inférieur au temps de communication vers

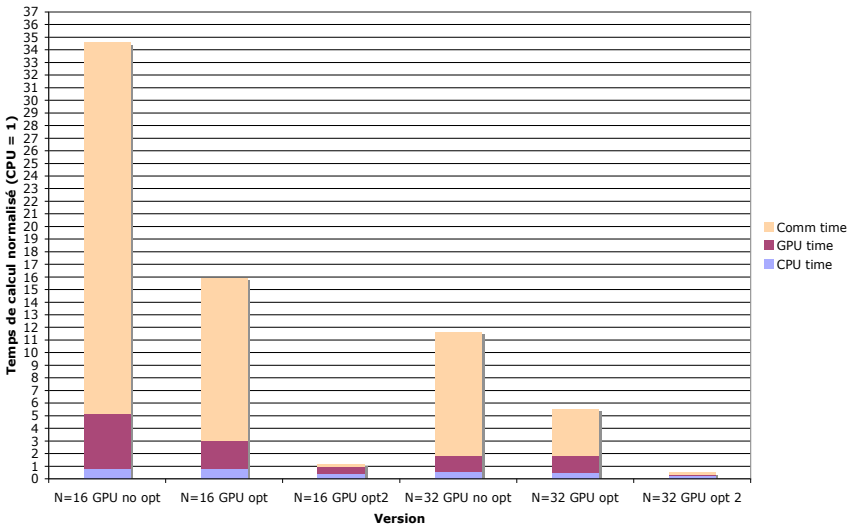

FIG. 3.7 – Histogramme des performances des versions GPU de FT par rapport à la version CPU (temps de référence égal à 1). Ce graphique présente les résultats pour la version non optimisée et les deux versions optimisées pour des tailles allant de 16 à 32. Le seuil critique permettant d'obtenir du speed-up est franchi dans la version la plus optimisée à partir d'une taille de problème égale à 32.

le GPU dans la méthode sans optimisation. On communique donc plus de mémoire que nécessaire. Dans la version *opt* on ne recopie que les zones nécessaires.

Pour la version optimisée simple, *opt*, on a aussi un temps de communication superieur à celui de l'exécution sur CPU. Sa décroissance rapide permet de faire l'hypothèse de l'absoption d'un coût fixe élevé de mise en place des communications entre le CPU et le GPU élevé. Diminuer le volume des communications n'est donc pas suffisant, il faut diminuer leur nombre. Les méthodes de *coalescing* de la partie 3.4.1 ont donc du sens sur GPU. L'importance de ce surcoût fixe important doit être intégré dans le modèle de coût pour l'applicabilité des codelets d'ASTEX.

Enfin la version *opt*2 obtient un speed-up à partir de la taille 32. Malgré une optimisation optimale, il existe toujours un seuil de granularité minimal pour les codelets. Un codelet de granularité en temps d'exécution faible ne fonctionne donc jamais sur un GPU. Par ailleurs, l'optimisation agressive des communications est essentielle pour abaisser ce seuil de granularité.

La figure 3.8 présente la suite de la figure 3.7, c'est-à-dire le temps d'exécution des différentes versions GPU de FT mais pour les grandes tailles de problème (64 à 256).

Le speed-up obtenu avec la version *opt*2 à partir des problèmes de taille 128 commence à être sensible. On a alors un facteur supérieur à 2 de speed-up sur l'application globale. Sachant qu'on porte sur le GPU 67% de l'application initiale, le speed-up maximum d'après la loi d'Amdahl est d'environ 3. L'accélération obtenue par l'optimisation *opt*2 représente un facteur 11 sur le codelet initial sur GPU et 33 pour la version *opt*

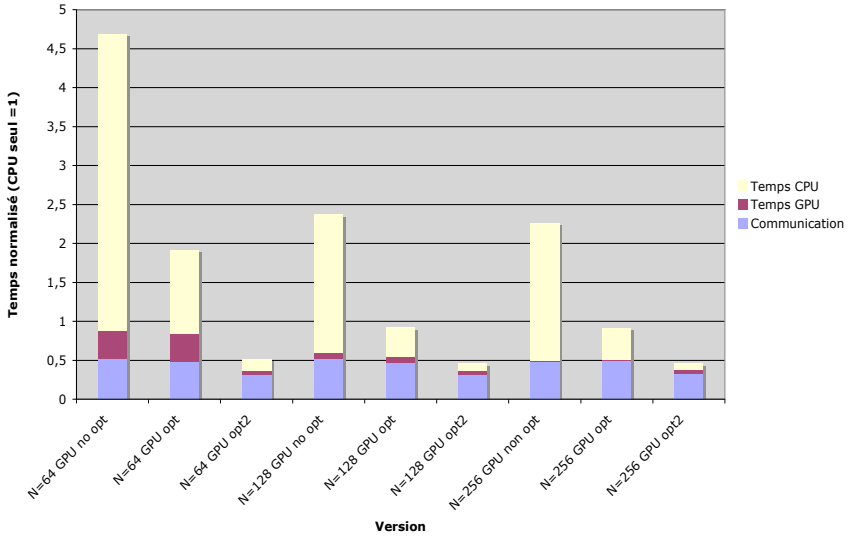

FIG. 3.8 – Histogramme des performances des versions GPU de FT par rapport à la version CPU (temps de référence égal à 1). Ce graphique présente les résultats pour la version non optimisée et les deux versions optimisées pour des tailles allant de 64 à 256. Malgré l'augmentation de taille du problème, le speed-up pour la version non optimisée et la version simple est nul ou très limité. La version *opt*2 présente un speed-up maximal de 2,17 sur un maximum de 3 d'après la loi d'Amdhal.

alors que le speed-up sur l'application globale est d'à peine 10%. Ces résultats montrent qu'il est nécessaire de produire un compromis entre optimisation des communications et optimisation du code sur le GPU.

La version avec une optimisation locale *opt* a toujours un surcoût de communication décroissant, ce qui tend à confirmer l'absorption d'un coût fixe par l'augmentation du coût variable lié à la complexité de l'application. Dans le cas de *opt*2, au delà d'une taille critique (environ 64), ce temps de communication représente une portion quasiment proportionnelle au temps d'exécution. C'est cette proportionnalité qui forme l'hypothèse du modèle de coût d'ASTEX où granularité et volume de communication sont linéairement liés. Sa plage d'utilisation est donc restreinte dans ce cas pour les tailles supérieures à 64.

Le graphique 3.9 présente pour trois versions implantées (CPU, GPU *opt* et GPU *opt*2), la courbe du temps d'exécution de l'application FT en fonction de la taille du problème. Le temps d'exécution est lié à la complexité et donc au volume des données traitées. L'algorithme étant le même dans les trois cas, la complexité reste identique.

Le temps d'exécution de la version *opt* est toujours sensiblement équivalent au temps sur CPU seul. Le gain sur le temps de calcul est consommé par les communications. Implanter l'algorithme de cette manière sur le GPU n'a *a priori* pas d'intérêt.

Dans la version *opt*2, le gain en temps de calcul dû au parallélisme du GPU se traduit par un retard net en croissance de la courbe du temps d'exécution. La courbe est la

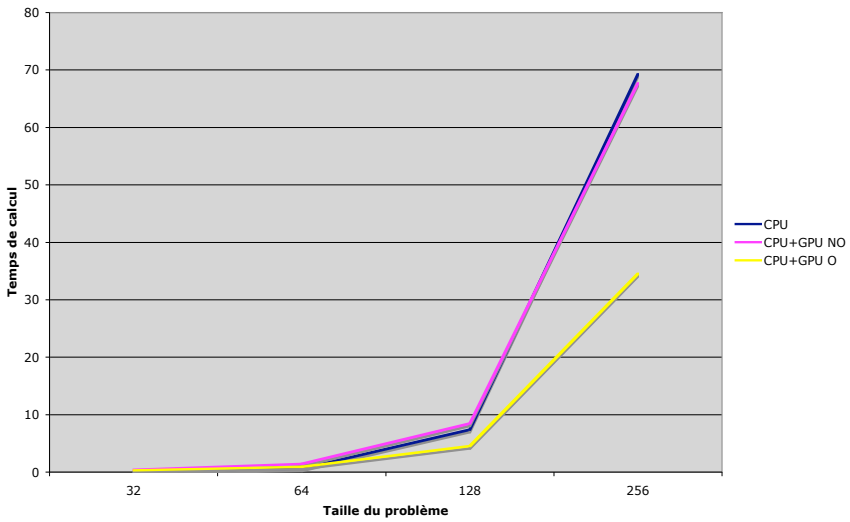

FIG. 3.9 – Courbe du temps d'exécution de l'application FT des NAS en fonction de la taille du problème pour trois versions différentes : CPU seul, opt et opt2.

même à une homothétie près. La complexité ne change pas mais l'utilisation du GPU permet de la répartir. Sur cette section de code, nonobstant les surcoûts d'exécution, le calcul à effectuer séquentiellement est divisé par le degré de parallélisme effectivement exploité du GPU.

L'histogramme 3.10 présente la répartition du temps de calcul entre GPU, CPU et communications pour les trois versions implantées sur le GPU.

Dans la version non optimisée, la proportion du temps de communication est prépondérante et varie peu. Il n'existe pas de taille critique au delà de laquelle on peut envisager un speed-up.

Dans les deux versions optimisées, la proportion du temps de calcul décroît linéairement. Cependant une fois le dégré maximum de parallélisme atteint sur le GPU le nombre de communications recommence à augmenter. Dans le cas de *opt*, cette décroissance s'observe à partir de la taille 256. Les deux courbes sont représentées en rouge sur la figure 3.10.

Dans le cadre de l'utilisation des GPU comme coprocesseur, la latence du code sur le CPU entre deux appels du codelet est négligeable devant les coûts de communication. Par conséquent, les deux politiques d'optimisation les plus importantes dans ce cadre sont la diminution du volume des communications et de leur nombre, en particulier en évitant la mise à jour de zone résidente. La classification des accès selon leur probabilité joue donc un rôle clef.

Dans la solution *opt*2, l'appel au codelet a été réordonné afin d'optimiser la localité

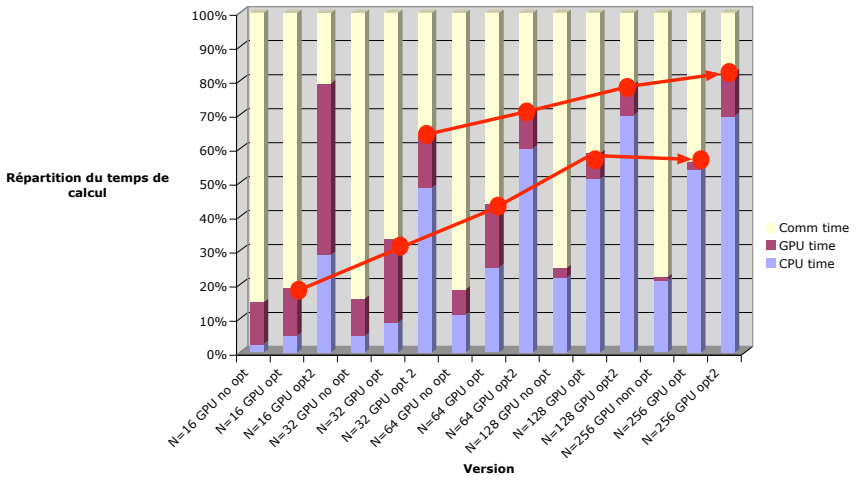

FIG. 3.10 – Histogramme de la répartition du temps de calcul dans les différentes versions exécutées. On retrouve en rouge l'évolution en $1/x$ de l'influence des communications.

mémoire. Ce réordonnancement crée de nouvelles zones résidentes. Enfin, pour les accès peu probables la génération des tests de spécialisation efficaces est essentielle. En déterminer la forme optimisée est une perspective possible dont il existe des précédents dans la littérature [77].

Dans la suite des expérimentations, il faudrait idéalement conduire des expérimentations sur un système hétérogène supportant un grain plus fin pour les codelets afin de déterminer l'influence dans ce cadre du préchargement ; le CELL semble être un exemple possible.

3.6 Conclusions et perspectives

Les algorithmes et expérimentations présentés dans ce chapitre sont à un stade préliminaire. Les conclusions sur les algorithmes sont théoriques et celles sur les expérimentations se limitent au cadre des GPU.

L'algorithme direct est d'une complexité relativement faible et simple à mettre en œuvre. Son utilisation limitée dans l'optimisation de la latence masquée par le préchargement n'est pas un facteur pénalisant dans le cadre du GPGPU.

Les deux algorithmes utilisant la fonction de coût, bien que fournissant une solution de meilleure qualité, possèdent eux aussi des avantages et des inconvénients.

En premier lieu, le calcul de la fonction de coût est particulièrement complexe et est

tributaire de la qualité du profil spéculatif. L'algorithme à heuristique tire partie d'une simplification poussée du graphe pour limiter le nombre de coûts à évaluer pendant l'initialisation et le reste de l'algorithme. Sur des applications de taille raisonnable, ou en application locale, il devient exploitable.

Le paramétrage de la fonction de coût en fonction des architectures et des applications est un problème complexe non encore abordé.

Enfin, ces algorithmes ne garantissent pas en échange de leur complexité l'optimalité de leur solution.

Dans la pratique, ces algorithmes sont tous utilisables. Les graphes de contrôle de flot à traiter atteignent rarement le millier de BB et la limite quant à la réductibilité du graphe n'est que très rarement franchie [86]. Il peuvent donc tous trois raisonnablement être mis en œuvre. Seule l'expérimentation sur des codelets de grains fins usant de la latence du calcul sur le CPU pour masquer les communications permettra de les discriminer.

Les données issues d'Astex permettent de traiter le problème de manière spéculative. L'approche hybride permet de traiter efficacement le code C et les alias, sans faire usage d'analyse statique complexe et éventuellement non concluante. Les solutions sont en théorie de qualité mais la mesure reste une question ouverte.

Les expériences préliminaires montrent l'intérêt d'une stratégie d'optimisation des communications. L'influence sur le GPU du recouvrement par le calcul du CPU de la latence des communications est négligeable. C'est la réduction des communications et la détection de zones résidentes qui comptent et c'est ce qui en partie est implanté dans la spécialisation. En intégrant les données spéculatives du flot de contrôle et des accès mémoire en dehors du codelet, l'algorithme direct d'optimisation des communications complète avantageusement le profil spéculatif des données en entrée-sortie déjà construit par Astex. De fait, les données non modifiées ne sont *a priori* pas mises à jour inutilement.

Dans le cadre d'une granularité plus fine, l'intérêt du préchargement pour masquer les latences est probablement non négligeable. Il faut tester notre approche sur ce type de couple application-système.

Par manque d'expérimentation, déterminer les améliorations nécessaires aux algorithmes est prématuré. On peut cependant envisager quelques éléments.

Améliorer la qualité du profil spéculatif : dans sa version actuelle, Astex ne détermine plus quelles parties des zones mémoire sont accédées. Quand les zones sont de grande taille et accédées par morceaux, cette information devient indispensable.

L'utilisation d'historique dynamique de l'exécution permet de prédire les zones mémoire à copier et calculer une corrélation entre les différents paramètres et leurs valeurs spéculatives. Ces informations permettent d'ajouter une probabilité à la recopie qui aide à affiner l'approximation de l'ensemble des données à copier avant l'appel suivant au codelet. Quand la probabilité d'une zone mémoire associée à l'historique courant est

faible, on teste sa nécessité avant de l'exécuter. L'information de probabilité est devenue dynamique. Afin de mettre en œuvre ce système, il existe de nombreux outils : les historiques (cf. 2.2.1.3), les matrices de corrélation, les chaînes de Markov, les réseaux Bayésien,...

Il est possible, sur la base de transformation de code, de créer des opportunités nouvelles d'optimisation des communications. Il s'agit en particulier de créer des zones résidentes et de diminuer les communications et les dépendances en réordonnançant et en redécoupant les codelets. C'est l'optimisation qui a été effectuée à la main dans la version la plus optimisée de FT dans la section 3.5.1. Ce procédé a déjà été partiellement implanté par Guillaume Papauré dans un outil développé à l'INRIA de Rennes sur la base logiciel d'ASTEX mais indépendant de ce dernier. Dans le cadre de l'utilisation d'OpenMP, l'outil détecte des congestions dans les flots de données. L'intégration dans ASTEX de ce profilage du flot de données *a posteriori* est une voie intéressante pour le raffinement des codelets et de leur ordonnancement dans le cadre de l'optimisation des communications.

Avant d'investir plus de temps dans ces algorithmes, il faut trouver leurs domaines d'application puis faire leur intégration dans ASTEX. Une campagne de tests pour choisir, éventuellement automatiquement, les algorithmes et leurs paramètres en fonction des architectures cibles, des applications et leurs profils spéculatifs, est nécessaire.

Conclusion générale

Dans ce document est présenté mon travail de thèse. Il s'agit de traiter du problème du partitionnement automatique d'applications séquentielles en codelets spéculatifs pour les architectures hétérogènes à mémoires distribuées. Dans un premier temps ont été établis l'état de l'art sur les architectures, les systèmes *multithreads* et leurs optimisations, en particulier l'optimisation des communications. J'ai ensuite présenté un modèle spéculatif pour le partitionnement automatique d'application en codelet, et son implantation, Astex. La seconde contribution porte sur l'étude du problème de l'optimisation des communications et en particulier de leur placement. Il s'agit d'un problème critique pour l'exploitation de coprocesseurs hétérogènes. Pour chaque contribution ont été établie des conclusions et perspectives.

Dans cette conclusion à mon document de thèse, je résume les différentes contributions apportées sur le sujet. Dans un second temps, je présente les perspectives, pour le modèle de partitionnement spéculatif et pour l'optimisation des communications. Enfin, une conclusion générale fait le bilan sur la situation actuelle d'Astex et son impact sur la communauté.

Contributions

Cette section présente un résumé des contributions apportées par mon travail de thèse au domaine du partitionnement automatique.

Il s'agit dans un premier temps du modèle d'extraction spéculatif et de son implantation logicielle : Astex.

Dans un second temps est résumé l'apport de mon travail à la formalisation du problème d'optimisation du placement des communications.

Modèle spéculatif pour le partitionnement d'application et logiciel

L'objectif principal de mon travail de thèse était la mise au point d'un processus automatique de partitionnement d'application. La première étape est l'élaboration d'un modèle pour la caractérisation des partitions.

Dans cette thèse, j'ai introduit la notion de *codelet*. C'est l'unité de base de la partition. Il s'agit d'une fonction pure contenant le noyau de calcul à extraire, *i.e.* une fonction ayant une seule entrée et une seule sortie ne faisant appel à aucun autre élément extérieur que ces paramètres.

Le processus élaboré base l'extraction des codelets sur la spéculation. Il s'agit d'extraire d'un nombre restreint d'exécutions les informations utiles et/ou nécessaires pour un partitionnement efficace. Ces informations sont spéculatives. La représentation de ces informations est possible grâce à la mise au point d'un modèle spéculatif. Il s'agit d'une contribution originale de mon approche.

L'usage de la spéculation apporte de nombreux avantages. En particulier celui de fournir les informations suffisantes pour répondre au problème posé par l'utilisation d'architecture à mémoire distribuée et/ou non cohérente. Astex est actuellement à ma connaissance la seule approche de partitionnement automatique traitant de ce problème particulier sans adjonction de contrainte matérielle.

Enfin, le processus se déroule en préambule de la phase de compilation du programme. Ainsi, l'approche est indépendante de l'architecture, elle supporte les architectures hétérogènes. De plus, elle ne fait appel à aucun support matériel. À la compilation, sur le support des informations spéculatives, sont générées une ou plusieurs versions de chaque codelet pour chaque architecture visée.

Une fois le modèle élaboré, un processus logiciel, Astex a été construit. Par instrumentation, profilage et traitement des résultats, est construit sur le modèle spéculatif, une partition de l'application en entrée.

La complexité acceptable de la détection et la taille maitrisée des informations de profilage rendent le processus de calcul des codelets compatible avec les applications de relativement grande taille. Les résultats obtenus par expérimentations valident l'approche d'Astex pour le partitionnement automatique d'application.

Dans sa version actuelle, le logiciel Astex ne reprend pas toutes les possibilités du modèle spéculatif explorées et validées dans le prototype de recherche. Il est cependant parfaitement fonctionnel et entame depuis 2009 son exploitation commerciale. Du point de vue du logiciel industrialisé, de nombreux développements restent à faire.

Formalisation du problème d'optimisation des communications et algorithme

Il a été établi par expérimentation que les communications dans l'utilisation de coprocesseur étaient le principal facteur limitant dans son exploitation efficace. L'optimisation des communications est donc un point critique.

Dans un premier temps, le processus de partitionnement implanté par Astex apporte une réponse partielle au problème. En spéculant sur les zones réellement accédées on évite la copie de toutes les zones potentiellement accédées. Avec l'utilisation de pointeur et les alias possibles, ce résultat est indispensable.

Dans un second temps a été élaboré et prototypé une méthode pour la détection des zones potentiellement résidentes en mémoire. L'utilisation de profil spéculatif restreint sur l'utilisation des paramètres des codelets entre les appels est une bonne solution.

Enfin, le possible recouvrement des latences des communications nécessaires pour le reste du code de l'application est en cours d'élaboration. Il s'agit du problème du placement des communications, et de son optimisation. Il y a en effet deux problèmes à traiter : la validité d'un placement et son efficacité.

Un placement est valide si tous les éléments nécessaires au codelet sont garantis d'être à jour dans la mémoire locale lors de son appel. L'efficacité appelle quant à elle l'élaboration d'une fonction de coût pour être évaluée.

Validité et efficacité ont donc nécessité la formalisation du problème d'optimisation du placement des communications. Ce problème n'a au préalable jamais fait l'objet d'un formalisme poussé dans la littérature.

Une fois le problème formalisé, sa proximité avec le mincut, le maxcut, et l'optimisation des barrières de synchronisation, ont inspiré trois différents algorithmes valides mais dont l'efficacité reste à déterminer par l'expérimentation.

Perspectives

Le travail de thèse effectué a fourni des modèles et outils solides et efficaces pour aborder les problèmes liés à l'utilisation automatique de systèmes hétérogènes à mémoire distribuée.

Deux voies sont possibles pour les perspectives :
- étendre la base du modèle et l'efficacité du logiciel qui l'implante,
- construire de nouvelles utilisations des données constitutives du modèle spéculatif.

Extension du modèle spéculatif et de ses applications

Dans la conclusion du chapitre 2 ont été établies des perspectives pour l'évolution d'Astex et de son modèle :
- l'optimisation des tests de spéculations,
- l'utilisation de l'outil pour déterminer l'affinité entre architecture et application,
- l'ordonnancement des tâches,

- l'allocation des ressources (*affinity scheduling*),
- l'utilisation de l'outil de partitionnement pour une compilation « différenciée »,
- la gestion du parallélisme, il s'agit notamment de gérer les problèmes de synchronisation entre les codelets,
- la gestion de structures de données plus complexes par la mise en place de mécanismes de spéculation et de translations d'adresse nécessaires à la gestion d'accès comportant de multiples indirections.

L'étape actuelle dans la prospection sur les applications du profil spéculatif généré par ASTEX est de tester l'utilisation sur le code partitionné et spécialisé d'un compilateur auto-adaptatif : gcc milepost [71].

Lors de discussions scientifiques avec des architectes, ces derniers ont montré un intêret certain pour l'outil et ses résultats dans le cadre du *co-design* ; cette voie est à explorer.

Dans le développement de nouvelles optimisations avec ASTEX, la prochaine étape est l'optimisation des communications. Elle bloque actuellement sur la simplification excessive de la spéculation sur les zones mémoire dans la version actuelle du logiciel comparée au prototype de 2007, en particulier l'absence de profilage sur les accès en dehors des codelets. Cette situation est appelée à évoluer rapidement.

Intégration et étude des algorithmes d'optimisation des communications

Par manque d'expérimentation, déterminer les améliorations nécessaires aux algorithmes est prématuré. Dans la conclusion du chapitre 3 ont été détaillées les perspectives et évolutions suivantes :

- adapter et améliorer la qualité du profil spéculatif sur l'utilisation de la mémoire
- étudier des transformations de code pour l'optimisation des communications (*e.g.* redécoupage des codelets)
- étudier l'ordonnancement spatial et temporel des codelets pour limiter les communications (zones résidentes, *streaming*, *affinity scheduling* ...).

L'étude de ces possibles perspectives demande en préalable l'implantation effective dans ASTEX d'un module d'optimisation des communications.

Conclusion

Les bases théoriques et l'outil construit ont produit des résultats aujourd'hui reconnus. L'utilisation commerciale des codelets et profils spéculatifs fournis par ASTEX, en particulier par Intel, a commencé en 2009. L'outil est destiné à être un élément durable des outils de compilation de CAPS entreprise.

Des approches similaires sont en train d'être développées dans d'autres laboratoires et entreprises, ce qui confirme la qualité de l'approche d'ASTEX et sa contribution à l'art.

ASTEX étant un produit pérenne, je ne doute pas de la qualité de ses futurs développements. Les résultats déjà acquis fournissent une base solide de travail pour les futurs travaux de recherche dans la compilation pour les *manycores*, sujet critique encore très ouvert.

Glossaire

adaptabilité
: (d'une application) Il s'agit de la capacité d'adaptation dynamique d'une application à son contexte d'exécution, 9

alias
: On dit qu'une variable d'un programme est un alias d'une autre si elles désignent toutes deux le même élément de la mémoire physique, 11, 28, 33, 36, 45, 46, 54, 61, 66

alignement des données
: L'alignement des données est relatif à l'adresse physique des variables. Il s'agit de savoir si elles sont alignées sur les lignes de cache, les pages, ou toutes autres structures liées au fonctionnement de l'*architecture*, 28

API
: *Application Programing Interface*, est la couche logicielle d'interface permettant l'utilisation de composants matériels ou logiciels extérieurs au programme, 27, 165

architecture
: Ce terme désigne en informatique l'environnement matériel d'exécution des programmes, 165

BB
: C'est l'abréviation utilisée pour *Bloc de Base* ou *Basic Bloc* en anglais, 29, 90, 165

Bloc de Base
: C'est un groupe d'instructions du programme consécutives sans instruction de saut, 29, 165

C
: Le C est un langage de programmation impératif plutôt bas niveau et faiblement typé ; d'autres langages comme *C++* et JAVA ont une syntaxe proche du C, 11, 165

C++
: C'est un langage objet compatible avec le langage *C*, 27, 165

CAPS entreprise
: *CAPS entreprise* est une start-up issue de l'équipe de recherche ALF (anciennement CAPS) à l'INRIA de Rennes, 55, 165

CFG
: *Control Flow Graph*, c'est le graphe de contrôle de flot, c'est-à-dire un graphe où chaque noeud est un *BB* et chaque arc correspond à une alternative d'un branchement, 29, 30, 32, 33, 37, 38, 47, 50, 131, 148, 179, 180, 182

codelet
: Un codelet est une *fonction pure* qui est l'unité de base du partitionnement d'application, 12, 14, 29, 160

Control Flow
: Il s'agit de la locution anglaise correspondant au *contrôle de flot*, 29, 165

contrôle de flot
: Il s'agit du chemin suivi dans les structures de contrôle,i.e. boucle, branchement conditionnel, appel de fonction, saut,..., 165

DSP
: *Digital Signal Processing*, se dit d'un processeur parallèle ou unité d'un processeur dédié aux calculs parallèles, 11, 25

environnement d'exécution
: Il s'agit du contexte logiciel et matériel de l'exécution formé par le *système*, l'*architecture*, mais aussi de potentiels composants logiciels apportés par l'utilisateur au moyen d'*API* ; c'est le cas de *HMPP*, 60, 166

exécution dans l'ordre
: Il s'agit d'un mode d'exécution des processeurs où les instructions sont exécutées dans l'ordre du programme en entrée, 165

flot de donnée
: C'est l'utilisation faite des données dans l'exécution d'un programme, 54

fonction pure
: C'est une fonction autosuffisante du code, c'est à dire qu'elle ne fait pas appel à des fonctions extérieures, et toutes les variables utilisées sont locales ou des paramètres de la fonction, 29, 165

FPGA
: *Field Programmable Gate Array*, est un type de processeur qui contient des parties reprogrammables, 25, 28

GPGPU
: *General Purpose computing on a GPU*, correspond à l'utilisation d'un processeur graphique comme unité de calcul généraliste, 12, 25, 26, 56

GPU
: *Graphical Processing Unit*, Un processeur parallèle dédié aux calculs graphiques, 10, 11, 21, 26, 27, 35, 165

granularité
: c'est le nom donné au paramètre représentant la taille des éléments d'un ensemble. Ex : la granularité d'une tâche est plus élevée que celle d'une instruction, 18, 27, 44, 45

HMPP
: *Heterogeneous Manycore Parallel Programing*, c'est une interface de programmation développée par *CAPS entreprise* pour les *architectures* multi-processeurs hétérogènes, 12, 45, 55, 62, 65, 88, 91, 92, 94, 123, 165

hotpath
: La traduction littérale est chemin chaud. Il s'agit d'un chemin parmi les plus usités du code, c'est-à-dire une séquence de *BBs*, et toutes ses instances dans la *trace* d'exécution ont un temps d'exécution jugé important, 78, 79, 90, 95, 105, 181

hotspot
: C'est une portion de code d'une application, sans distinction de chemin, parmi les plus usitées lors de l'exécution, encore appelée point chaud ; les *hotpaths* sont un sous-ensemble de l'ensemble des hotspots, 120

HPC
: *High Performance Computing*, il s'agit du domaine de l'informatique en charge des calculs numériques de grande taille qui demandent des performances élevées, 55

ILP
: *Instruction Level Parallelism*, c'est l'abréviation anglaise correspondant au parallélisme d'instruction, 19, 22, 166

in order
: Il s'agit de la locution anglaise correspondant à l'*exécution dans l'ordre*. Cette notion s'oppose à l'exécution de type *out of order*, 31

ISA
: *Instruction Set Architecture*, est la description de l'ensemble des instructions accessibles au programmeur et exécutables pour un processeur donné. La locution française consacrée est « jeu d'instructions », 44, 62

JIT
: *Just In Time*, est une abréviation désignant l'utilisation d'un mécanisme intervenant sur le déroulement de l'exécution d'un programme pendant l'exécution de ce dernier. On parle de méthode à la volée. Par exemple la compilation de type JIT, 26, 44

legacy codes
: Il s'agit de l'ensemble des programmes qui sont les applications minimum que doit pouvoir effectuer l'*architecture* pour être valide dans son domaine, 28

nid de boucles
: On appelle nid de boucles l'imbrication stricte de plusieurs boucle dans une application, 17

ordonnancement
: C'est le procédé en charge de la répartition des tâches ou instructions sur les différentes unités ou sur différents processeurs, 30, 31, 34, 48, 166

out of order
: Il s'agit de la locution anglaise correspondant à l'*exécution dans le désordre*, 165

padding
: (de la mémoire), il s'agit de l'action visant à ajouter des octets vides de mémoire entre les variables de manière à les aligner, on peut traduire ce terme par «allocation de zone tampon», 100

portabilité
: (d'un programme, d'un code). Il s'agit de la capacité d'un programme à être exécuté sur différents systèmes et architectures sans modification majeure, 9

pragma
: Il s'agit d'une directive insérée dans un code et destinée à orienter le travail du compilateur ; son interprétation est située dans le compilateur à l'étage du pré-processeur, 92

profils d'exécution
: Il s'agit de l'ensemble des données représentatives de l'exécution issue d'une ou d'un ensemble de *traces* d'un programme, 60

RAW
: *Read After Write*, se dit d'une dépendance de type lecture après écriture, 34, 37, 38, 179

RISC
: *Reduced Instruction Set Computer*, c'est un jeu d'instructions pour les processeurs composés d'un nombre relativement faible d'instructions simples, 22, 25

run-time
: Il s'agit de la locution anglaise correspondant à l'*environnement d'exécution*, 44, 123, 127

scheduling
: Il s'agit de la locution anglaise correspondant à l'*ordonnancement*, 31

SIMD
: *Single Instruction Multiple Data*, il s'agit d'un processeur exécutant en parallèle la même instruction sur des données différentes, 11, 19–21, 25–27, 31, 166, 179

SMT
: *Simultaneous Multi-Threading*, il s'agit d'un procédé permettant l'exécution simultanée de plusieurs *threads* dans un même processeur, 22

SPMD
: *Single Process Multiple Data*, il s'agit d'un système où plusieurs processeurs autonomes exécutent simultanément le même programme. Il n'y a pas de synchronisation implicite sur chaque instruction comme dans le *SIMD*., 47, 54

système
: Le système est le substrat logiciel d'exécution des applications utilisateur, 165

trace
: On appelle trace un ensemble d'éléments représentatifs d'une exécution du programme ; par exemple, la liste des *BBs* traversés lors une exécution constitue une trace du programme, 165

VLIW
: *Very Long Instruction Word*, c'est un jeu d'instructions pour les processeurs ; Chaque instruction comprend une commande par unité de processeur qui sera exécutée en parallèle des autres sous instruction de l'instruction VLIW, l'*ILP* est géré par le compilateur, 11, 25, 31

WAR
: *Write After Read*, se dit d'une dépendance de type écriture après lecture, 34, 37, 179

Bibliographie

[1] J. Steffan and T Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98 : Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, page 2. IEEE Computer Society, 1998.

[2] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *ASPLOS-X : Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 184–196. ACM Press, 2002.

[3] J. T. Oplinger, D. L. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *PACT '99 : Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303. IEEE Computer Society, 1999.

[4] S. J. Patel and S. S. Lumetta. rePLay : A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6) :590–608, 2001.

[5] M. Chen and K. Olukotun. TEST : a tracer for extracting speculative threads. In *CGO '03 : Proceedings of the international symposium on Code generation and optimization*, pages 301–312. IEEE Computer Society, 2003.

[6] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, K. Szabo, D. Lyonnard, and G. Nicolescu. A Multi-Processor SoC Platform and Tools for Communications Applications. 2004.

[7] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA : High-Level Synthesis of Nonprogrammable Hardware Accelerators. *J. VLSI Signal Process. Syst.*, 31(2) :127–142, 2002.

[8] F. Warg and P. Stenström. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *IPDPS '03 : Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 12.2. IEEE Computer Society, 2003.

[9] J. Dou and M. H. Cintra. Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization. In *IEEE PACT*, pages 203–214, 2004.

[10] D. Robbins. POSIX threads explained. *IBM web site : http ://www-106.ibm.com/developerworks/library/l-posix2/*, 2000, cited 6. Aug. 2009.

[11] G. Pokam and F. Bodin. An Offline Approach for Whole-Program Paths Analysis using Suffix Arrays. In *LCPC '04 : Languages and Compilers for Parallel Computing*, 2004.

[12] A. Djabelkhir and A. Seznec. Characterization of embedded applications for decoupled processor architecture. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization*, 2003.

[13] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19(2) :111–170, 2001.

[14] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4) :1319–1360, 1994.

[15] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[17] Mihalis Yannakakis. A polynomial algorithm for the min-cut linear arrangement of trees. *J. ACM*, 32(4) :950–988, 1985.

[18] M. F. P. O'Boyle, L. Kervella, and F. Bodin. Synchronization minimization in a spmd execution model. *J. Parallel Distrib. Comput.*, 29(2) :196–210, 1995.

[19] Alain Darte and Robert Schreiber. A linear-time algorithm for optimal barrier placement. In *PPoPP '05 : Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 26–35, New York, NY, USA, 2005. ACM.

[20] Dhruva R. Chakrabarti and Prithviraj Banerjee. Global optimization techniques for automatic parallelization of hybrid applications. In *ICS '01 : Proceedings of the 15th international conference on Supercomputing*, pages 166–180, New York, NY, USA, 2001. ACM.

[21] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *PLDI '96 : Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 1996. ACM.

[22] Thomas Fahringer and Eduard Mehofer. Problem and machine sensitive communication optimization. In *ICS '98 : Proceedings of the 12th international conference on Supercomputing*, pages 117–124, New York, NY, USA, 1998. ACM.

[23] Harold N. Gabow. Perfect arborescence packing in preflow mincut graphs. In *SODA '96 : Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 528–538, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[24] E. A. Stöhr and M. F. P. O'Boyle. A graph based approach to barrier synchronisation minimisation. In *ICS '97 : Proceedings of the 11th international conference on Supercomputing*, pages 156–163, New York, NY, USA, 1997. ACM.

[25] F. Harrary R. Z. Norman and D. Cartwright. *Structural Models : An Introduction to the Theory of Directed Graphs*. John Wiley and Sons, New York, NY, USA, 1965.

[26] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation : long-range prefetching of delinquent loads. In *ISCA '01 : Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM.

[27] V. Chaudhary F. Liu. Extenting openmp for heterogeneous chip multiprocesseur.

[28] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading : A platform for next-generation processors. *IEEE Micro*, 17(5) :12–19, 1997.

[29] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11) :32–38, Nov. 2005.

[30] Wayne Wolf. The future of multiprocessor systems-on-chips. In *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 681–685, New York, NY, USA, 2004. ACM.

[31] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1) :29–63, 2003.

[32] R.W.M. Jones and P.H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. *The Third International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.

[33] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *CGO '04 : Proceedings of the international symposium on Code generation and optimization*, page 39, Washington, DC, USA, 2004. IEEE Computer Society.

[34] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7) :33–38, July 2008.

[35] James Burns and Jean-Luc Gaudiot. Smt layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(2) :142–155, 2002.

[36] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. Ilp versus tlp on smt. In *Supercomputing '99 : Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, New York, NY, USA, 1999. ACM.

[37] Erich Elsen, Mike Houston, V. Vishal, Eric Darve, Pat Hanrahan, and Vijay Pande. N-body simulation on gpus. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM.

[38] John Whaley and Christos Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP '05 : Proceedings of the 2005 International Conference on Parallel Processing*, pages 147–156, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Eric Petit, Francois Bodin, Guillaume Papaure, and Florence Dru. Astex : a hot path based thread extractor for distributed memory system on a chip. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 141, New York, NY, USA, 2006. ACM.

[40] Eric Petit, Guillaume Papaure, Florence Dru, and François Bodin. Astex : a hot path based thread extractor for distributed memory system on a chip. In *High-Performance Embedded Architecture and Compilation Industrial Workshop (HiPEAC)*, 2006.

[41] E. Petit, F. Bodin, and R. Dolbeau. An hybrid data transfer optimization for gpu. In *Compilers for Parallel Computers (CPC2007)*, 2007.

[42] F.Bodin and R.Dolbeau. Hmpp programming tool home page. In *http ://www.caps-entreprise.com/hmpp.html*.

[43] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3) :1–15, 2008.

[44] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1) :179–196, Jan. 2006.

[45] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06 : Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[46] Joachim Clabes, Joshua Friedrich, Mark Sweet, Jack DiLullo, Sam Chu, Donald Plass, James Dawson, Paul Muench, Larry Powell, Michael Floyd, Balaram Sinharoy, Mike Lee, Michael Goulet, James Wagoner, Nicole Schwartz, Steve Runyon, Gary Gorman, Phillip Restle, Ronald Kalla, Joseph McGill, and Steve Dodson. Design and implementation of the power5 microprocessor. In *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 670–672, New York, NY, USA, 2004. ACM.

[47] R. Kalla, Balaram Sinharoy, and J.M. Tendler. Ibm power5 chip : a dual-core multithreaded processor. *Micro, IEEE*, 24(2) :40–47, Mar-Apr 2004.

[48] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara : a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2) :21–29, March-April 2005.

[49] A. Gara et al. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2/3) :195–212, 2005.

[50] ClearSpeed Technology plc. Csx processor architecture whitepaper. In *www.clearspeed.com/docs/resources/*, 2005.

[51] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research : A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[52] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *PPOPP '95 : Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 144–155, New York, NY, USA, 1995. ACM.

[53] K. Kennedy and N. Nedeljkovic. Combining dependence and data-flow analyses to optimize communication. In *in Proceedings of the 9th International Parallel Processing Symposium*, pages 340–346, 1995.

[54] J. D. Owens et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[55] Richard Vincent Bennett, Alastair Colin Murray, Björn Franke, and Nigel Topham. Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. In *LCTES '07 : Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 83–92, New York, NY, USA, 2007. ACM.

[56] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC '72 : Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136, New York, NY, USA, 1972. ACM.

[57] Thi Viet Nga Nguyen and François Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3) :527–570, 2005.

[58] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05 : Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society.

[59] nones Carlos García Qui Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler : an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05 : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, New York, NY, USA, 2005. ACM.

[60] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of gpus using the rapidmind development platform. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[61] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2) :40–53, 2008.

[62] Mitrionics. Mitrionic home page. In *www.mitrionics.com*, 2008.

[63] A. Richards. The codeplay sieve c++ parallel programming system. In *http ://www.codeplay.com/downloads_ public/sievepaper-2columns-normal.pdf*, 2006.

[64] Jon Stokes. Peakstream unveils multicore and cpu/gpu programming solution. In *http ://arstechnica.com/news.ars/post/20060918-7763.html*, 2006.

[65] Aaftab Munshi. Opencl parallel computing on the gpu and cpu. In *SIGGRAPH'08*, 2008.

[66] Intel Corp. www.intel.com/technology/architecture/downloads/quad-core-06.pdf. 2008.

[67] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla : A unified graphics and computing architecture. *IEEE Micro*, 28(2) :39–55, 2008.

[68] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[69] AMD White Paper. Multi-core processors - the next evolution in computing. 2005.

[70] AMD White Paper. Ati ctm guide : Technical reference manual. 2006.

[71] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. Milepost gcc : machine learning based research compiler. 2008.

[72] Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan, and Dhabaleswar K. Panda. Performance analysis and evaluation of pcie 2.0 and quad-data rate infiniband. In *HOTI '08 : Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, pages 85–92, Washington, DC, USA, 2008. IEEE Computer Society.

[73] E. Petit and F. Bodin. Extracting speculative threads using traces for soc with astex. In *Compilers for Parallel Computers (CPC2006)*, 2006.

[74] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[75] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO'08*, pages 330–341. IEEEACM, 2008.

[76] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss : a programming model for the cell be architecture. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[77] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS '07 : Proceedings of the 21st annual international conference on Supercomputing*, pages 263–273, New York, NY, USA, 2007. ACM.

[78] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6) :1115–1145, 1995.

[79] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76, New York, NY, USA, 1987. ACM.

[80] Ron Y. Pinter. Optimal layer assignment for interconnect. *Adv. VLSI Comput. Syst.*, 1(2) :123–137, 1984.

[81] Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Oper. Res.*, 36(3) :493–513, 1988.

[82] Martin Skutella. Semidefinite relaxations for parallel machine scheduling. *Foundations of Computer Science, Annual IEEE Symposium on*, 0 :472, 1998.

[83] D.R. Fulkerson and L.R. Ford. Maximal flow through a network. *Canadian Journal of Mathematics*, 8 :399–404, 1956.

[84] Marie-Claude Gaudel et Michèle Soria Christine Froidevaux. *Types de données et algorithmes*. McGraw-Hill, Collection Informatique, 1990.

[85] Alan Chun Wai Leung. Automatic parallelization for graphics processing units in jikesrvm. Master's thesis, University of Waterloo, May 2008.

[86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[87] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *PLDI '09*. ACM, 2009.

[88] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95 : Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[89] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *ISCA '92 : Proceedings of the 19th annual international symposium on Computer architecture*, pages 58–67, New York, NY, USA, 1992. ACM.

[90] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (spsm) architecture : compiler-assisted fine-grained multithreading. In *PACT '95 : Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 109–121, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[91] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. *Int. J. Parallel Program.*, 27(1) :1–19, 1999.

[92] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03 : Proceedings of the ninth*

ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 13–24, New York, NY, USA, 2003. ACM.

[93] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 4(2) :12, 2007.

Table des figures

Résumé

Devant les difficultés croissantes liées au coût en développement, en consommation, en surface de silicium, nécessaires aux nouvelles optimisations des architectures monocœur, on assiste au retour en force du parallélisme et des coprocesseurs spécialisés dans les architectures. Cette technique apporte le meilleur compromis entre puissance de calcul élevée et utilisations des ressources.

Afin d'exploiter efficacement toutes ces architectures, il faut partitionner le code en tâches, appelées codelet, avant de les distribuer aux différentes unités de calcul. Ce partionnement est complexe et l'espace des solutions est vaste. Il est donc nécessaire de développer des outils d'automatisation efficaces pour le partitionnement du code séquentiel. Les travaux présentés dans cette thèse portent sur l'élaboration d'un tel processus de partitionnement. L'approche d'Astex est basée sur la spéculation, en effet les codelets sont construits à partir des profils d'exécution de l'application.

La spéculation permet un grand nombre d'optimisations inexistantes ou impossibles statiquement. L'élaboration, la gestion dynamique et l'usage que l'on peut faire de la spéculation sont un vaste sujet d'étude. La deuxième contribution de cette thèse porte sur l'usage de la spéculation dans l'optimisation des communications entre processeur et coprocesseur et traite en particulier du cas du GPGPU, i.e. l'utilisation d'un processeur graphique comme coprocesseur de calcul intensif.

Mots clés : compilation, optimisation, spéculation, parallélisme, architecture multicoeur, coprocesseur, communication, GPU, *manycore*, HMPP, Astex

Abstract

In light of the increase of development cost, power consumption and silicon area for new single-core architecture optimisations, the new way for performance improvements leads to multicore architectures, with parallel programming and specialised coprocessors. They give the best trade-off between high computing performance and required resources.

In order to efficiently address this new kind of architecture, applications have to be split into tasks, also called codelets, which will be mapped onto the different computing units of the host system. Code partitioning is complex and the solution space to explore is wide. It becomes necessary to develop automatic and efficient tools to generate codelets from sequential application. The purpose of this thesis is to propose such a code-partitioning process. The Astex approach is based on speculation ; the codelet detection and construction phases use speculative data from execution profiles of the application.

Speculation allows the compiler to handle a number of optimisations which would have been impossible or unavailable without speculative data. The way to collect the speculative data, how to handle them during the execution, and how to use them are new challenges for the compiler framework. My second contribution deals with the data transfer optimisation between the processor and the coprocessor by using speculation. The case study focuses on GPGPU, i.e. the usage of graphics processor for general-purpose computing.

Keywords : compiler, optimisation, speculation, parallelism, multicore, coprocessor, communication, GPU, *manycore*, HMPP, Astex