Méthode d'estimation de performance logicielle : application au développement rapide de code optimise pour une classe de processeurs dsp

Alain Pegatoquet

Thèse

Présentée devant

L'UNIVERSITE DE NICE-SOPHIA ANTIPOLIS
Ecole Doctorale des Sciences Pour l'Ingénieur

Pour obtenir le titre de

DOCTEUR EN SCIENCES

mention

SCIENCES POUR L'INGENIEUR

par

Alain PEGATOQUET

Titre de la thèse :

Méthode d'estimation de performance logicielle : application au développement rapide de code optimisé pour une classe de processeurs DSP

Soutenue le 27 octobre 1999 devant le jury :

M. François Charot	Rapporteur
M. Jean-Luc Philippe	Rapporteur
M. Daniel Dours	Président - Rapporteur
M. Michel Auguin	Directeur de thèse
M. Emmanuel Gresset	Membre invité
Mme. Cecile Belleudy	Membre invité

Remerciements

Cette thèse est le résultat de quatre années de collaboration entre l'entreprise VLSI Technology et le laboratoire I3S de l'Université de Nice-Sophia Antipolis.

Je tiens tout d'abord à remercier Michel AUGUIN, mon directeur de thèse, et Emmanuel GRESSET, mon responsable industriel, qui ont défini le cadre de ce travail et avec qui ce fut un réel plaisir de travailler. Mon travail a bénéficié de leur expérience, leurs compétences et leur enthousiasme. Leur disponibilité, leurs encouragements et leur patience m'ont donné l'assurance nécessaire à la réalisation de ce travail. Je leur en suis très reconnaissant. Je tiens à exprimer ma gratitude à Emmanuel Gresset pour m'avoir permis de présenter mes travaux dans plusieurs conférences et d'accepter de faire partie de mon jury de thèse.

Je remercie Daniel DOURS d'avoir rapporté sur mon mémoire de thèse et de me faire l'honneur de présider mon jury de thèse.

François CHAROT et Jean-Luc PHILIPPE m'ont fait également l'honneur de rapporter sur ma thèse. Je leur suis très reconnaissant d'avoir consacré une partie de leur temps à cette tâche.

Je remercie aussi tous les membres du laboratoire I3S, de l'Equipe MOSARTS et plus particulièrement Cécile BELLEUDY d'avoir accepté de faire partie de mon jury de thèse.

Je remercie également Daniel ABECASSIS, Directeur du groupe EPG, pour l'intérêt qu'il a porté à mes travaux et pour m'avoir accueilli au sein de son équipe. J'ai bien sûr une pensée pour toutes les personnes du groupe EPG et de VLSI Technology qui m'ont à un moment ou un autre consacré un peu de leur temps.

Je tiens à remercier tout particulièrement Guy GOGNIAT pour ses encouragements, ses conseils avisés et sa bonne humeur. Ce fut vraiment très agréable de travailler à ses cotés.

Je tiens à remercier Fernand BOERI pour son aide si précieuse et pour m'avoir permis de faire de l'enseignement dans le département GEii de l'IUT de Nice. Merci également à Charles André pour m'avoir accepté dans son DEA.

Une pensée toute particulière pour Remi, Moliv et Arnaud, pour leur contribution dans ce projet et avec qui il fut très enrichissant de travailler.

Je remercie aussi Patrick BALESTRA (« *el goleador* ») pour son aide précieuse et pour m'avoir accueillie au sein de la très célèbre équipe de football de l'AVDSA.

Enfin, je ne peux oublier mes parents, ma famille et mes amis qui m'ont sans cesse encouragé, fait confiance et donné les moyens de suivre les études que j'avais entreprises.

Table des matières

Table des Figures

Liste des Tables

Introduction

Les systèmes numériques prennent une place de plus en plus importante dans notre vie quotidienne. De la machine à café jusqu'au satellite en passant par le téléphone portable ou la télévision, tous ces systèmes intègrent des puces renfermant dans seulement quelques millimètres carré de silicium une complexité étonnante. Les ASIC (*Application Specific Integrated Circuit*) sont des circuits intégrés dédiés exclusivement à un type d'application. Les circuits de ce type permettent d'atteindre les meilleurs rapports coût (taille), performance et consommation, mais leur comportement est totalement figé. Afin de supporter des modifications de spécification de l'application ou pour permettre d'exécuter une gamme d'application plus vaste, les circuits intègrent de plus en plus de processeurs programmables. Avec ces circuits, une simple modification du programme suffit pour tenir compte des évolutions. En contrepartie, les processeurs programmables sont plus complexes que les circuits dédiés puisqu'une partie significative de la logique est consacrée au stockage et au contrôle pour l'exécution des programmes. Les systèmes à base de processeurs programmables sont généralement décomposés en deux grandes classes : les systèmes généraux et les systèmes embarqués. Les systèmes généraux sont constitués à partir de processeurs standards et correspondent par exemple aux ordinateurs personnels de type PC. Les processeurs spécialisés, contrairement aux processeurs généraux, sont conçus pour exécuter efficacement les programmes pour lesquels ils sont prévus, et peuvent avoir des performances dégradées pour d'autres. Dotés d'unités de calcul adaptées, de chemins de données optimisés et d'un jeu d'instructions spécialisé, ces processeurs permettent d'atteindre d'excellents compromis performance, surface et consommation.

En 1997, seulement 2% des quelques deux milliards de microprocesseurs vendus dans le monde étaient destinés aux systèmes généraux, les 98% restants sont intégrés dans des systèmes embarqués. Un système embarqué peut être défini par les caractéristiques [Cam96] suivantes:

- Il est dédié à une application particulière.
- Il doit respecter des contraintes temps-réel souvent strictes et réagir à des événements externes.
- Son fonctionnement correct est essentiel en raison de l'impact sur l'environnement extérieur.

Les systèmes embarqués réalisent typiquement deux types de traitement : les applications de contrôle et les applications de traitement du signal. Parmi les processeurs spécialisés, les DSP (*Digital Signal Processor*) sont voués aux calculs intensifs inhérents aux applications de traitement du signal alors que les microcontrôleurs sont conçus pour exécuter efficacement des programmes de contrôle. Le marché des DSP programmables est en pleine croissance puisque le nombre de processeurs de ce type vendu dans le monde est passé de 132,7 millions en 1995 à 440,3 millions en 1998[1], pour atteindre selon les prévisions le milliard d'unités en 2001. Ces dernières années, de nouveaux types de processeurs ultra spécialisés sont apparus, les ASIP (*Application Specific Instruction set Processor*). Dans [Mar95c], P. Marwedel situe les ASIP entre les ASIC et les processeurs spécialisés comme les DSP. Les ASIP ciblent en effet, non plus un domaine d'application (comme les DSP), mais en général moins d'un dizaine de programmes très spécifiques.

[1] Source In-Stat.

Les difficultés liées à la programmation de processeurs DSP

Les difficultés liées aux processeurs spécialisés programmables touchent à leur conception et à leur programmation. Dans ce document nous nous intéressons plus particulièrement aux problèmes des systèmes embarqués à base de DSP et à leur difficulté de programmation. De tels systèmes se caractérisent en effet par des contraintes de temps et de taille du code très fortes et font une part de plus en plus importante au logiciel par rapport au matériel. Actuellement, 60% du temps de développement de tels systèmes est consacré au codage logiciel.

Historiquement, le codage d'une application sur un processeur DSP s'effectuait manuellement. Tout d'abord, les constructeurs ont le plus souvent négligé l'investissement nécessaire au développement de compilateurs efficaces au profit de la conception même du circuit. Ce manque d'investissement s'explique par les faibles volumes des DSP qui ne justifiaient pas encore l'effort de conception d'un compilateur efficace. La complexité réduite des applications comme des architectures DSP permettait de plus une programmation en assembleur relativement aisée sur la base de fonctions de bibliothèques optimisées.

Mais la baisse du prix du matériel et les possibilités d'intégration que procurent les technologies sous microniques (actuellement 0,15 micron), permettent de concevoir des circuits contenant jusqu'à plusieurs millions de transistors par millimètre carré. Tirant profit de ces progrès, les constructeurs proposent des processeurs sans cesse plus performants pouvant exécuter plusieurs centaines de millions d'opérations par seconde. Les processeurs embarqués sont ainsi capables d'exécuter des applications de traitement du signal de plus en plus complexes. Mais cette puissance s'accompagne également d'un accroissement de la complexité de l'architecture cible [Pau97]. En conséquence, si pour les DSP du début des années 90 il est relativement aisé d'écrire du code assembleur tirant au maximum profit de l'architecture, ceci est beaucoup plus difficile avec les architectures parallèles récentes de type *dual-MAC* ou *VLIW*. Ces dernières possèdent en effet des contraintes liées au nombre élevé d'étages de pipeline (*delay slot*) ou encore des restrictions sur les chemins de données qui rendent considérablement plus complexes leur programmation.

Les coûts de développement élevés associés au codage manuel d'applications sur DSP et la pression sans cesse plus forte du « *time-to-market* » rendent cette situation de plus en plus inacceptable pour les entreprises et militent en faveur d'une approche de haut niveau basée sur l'utilisation de compilateurs [Woo98][Man99]. La situation a ainsi brusquement évoluée ces dernières années donnant aux compilateurs une place de plus en plus prépondérante dans le choix d'un DSP [Gal97]. Outre ces raisons, une telle approche augmente également la portabilité, la facilité de maintenance et le déboguage du code. Cette prise de conscience est générale dans le monde industriel. D'ailleurs Texas Instruments avec le processeur C6x[Tur97] ou Lucent avec le Sabre[Bod97] ont d'ores et déjà demandé à leurs utilisateurs d'abandonner l'assembleur au profit du langage C. L'utilisation d'un compilateur est indispensable pour ces deux processeurs en raison de leur complexité architecturale, donc de leur difficulté de programmation. Pour les processeurs à usage général, l'approche de haut niveau est courante puisque les compilateurs sont en général efficaces. Cela ne justifie pas pour autant leur utilisation dans des systèmes embarqués à base d'algorithmes de traitement du signal car leurs performances (nombre de cycles et taille de la mémoire programme) pour ce type d'applications sont souvent insuffisantes. Les DSP ont été conçu pour répondre aux contraintes fortes des systèmes embarqués, mais paradoxalement, leur haut degré de

spécialisation en font des cibles particulièrement délicates pour la génération de code optimisé.

Les problèmes de génération de code efficace pour DSP suscitent un intérêt, certes récent, mais croissant de la part des industriels comme de la communauté scientifique [Ara95a][Gal97][Cra97]. L'inefficacité des compilateurs C actuels pour DSP oblige en effet tout ou partie de l'application à être codée en assembleur. Cela reste malheureusement le plus souvent la seule solution envisageable pour respecter les fortes contraintes de coût, de performance et de consommation. Or le codage et l'optimisation d'un code assembleur est un travail long et fastidieux. Pour l'application G.728 [Rec94] (codeur/décodeur de parole) par exemple, l'implémentation d'une solution valide et optimisée sur le processeur OakDSPCore a demandé cinq mois de travail. De plus, pour tirer le meilleur profit des performances d'un DSP, il est souvent nécessaire d'avoir une bonne connaissance de l'architecture du processeur (afin d'utiliser de manière efficace le jeu d'instructions) mais aussi de l'application cible.

Le besoin de nouveaux outils

La réduction du temps de développement d'un code assembleur optimisé pour DSP passe par une approche de plus haut niveau. Il s'agit en effet d'utiliser plus systématiquement le compilateur afin d'écrire le moins possible de code assembleur. Or, si les compilateurs pour DSP sont globalement inefficaces, nous savons également qu'il est possible d'améliorer de manière significative les performances du code assembleur généré en modifiant le code C d'origine pour le compilateur cible (i.e. l'architecture cible). Les modifications peuvent être effectuées naïvement sur la totalité du code C. Cependant, pour les applications de traitement du signal les parties critiques en temps d'exécution représentent généralement une faible partie de la taille totale du code. Ce sont typiquement les instructions appartenant à des boucles imbriquées. Partant de ce constat, le travail d'optimisation doit évidemment s'effectuer prioritairement sur ces parties critiques afin d'éviter de perdre inutilement du temps sur des parties de code ayant peu d'influence sur les performances globales de l'application.

En l'absence d'outil adapté, la localisation de ces parties critiques s'effectue généralement par une simulation du code assembleur généré par le compilateur. Malheureusement, le principal inconvénient de cette approche est les temps de simulation très longs Il faut en effet compter plusieurs heures voire plusieurs jours de simulation pour une application complète avec une séquence de test représentative (plusieurs secondes de paroles dans le cas d'un algorithme de compression comme G.728).

Contributions de la thèse

Ces problèmes ont motivé l'élaboration de nouveaux outils permettant d'accélérer ce processus. C'est dans ce contexte que nous situons le travail présenté dans ce mémoire. Il s'agit en effet de concevoir et de réaliser un outil permettant d'utiliser au maximum le compilateur du DSP cible afin de réduire les temps de développement dus au codage en assembleur « à la main ». Nous proposons pour cela d'utiliser des méthodes d'estimation logicielle qui permettent de déterminer, à partir d'une description de haut niveau de l'application, les performances du code assembleur généré sans utiliser de simulateur de

niveau instruction. Malheureusement, dans le cas des DSP, cette estimation représente le plus souvent une mesure de l'inefficacité du compilateur C. En effet, les performances obtenues ne sont pas représentatives de ce que pourrait faire un programmeur expérimenté en écrivant le code assembleur manuellement. Nous proposons ainsi d'estimer également les performances d'un code assembleur optimisé c'est-à-dire comme s'il avait été écrit par un programmeur expérimenté.

Les objectifs de cette thèse sont d'apporter des contributions dans le domaine de l'estimation de performance d'un code optimisé, sujet qui a priori n'avait pas encore été traité. Le premier travail accompli a été de comprendre les problèmes liés à la génération de code pour DSP. Cette réflexion a donné naissance à une approche originale basée sur une représentation intermédiaire orientée schéma de calcul DSP et une technique pour décrire le processeur cible. Les autres contributions de ce travail concernent également la modélisation des contraintes de parallélisme au niveau du modèle du processeur cible ainsi que l'ordonnancement des tâches du graphe représentant l'application traitée adapté au type de processeur. L'outil mis en œuvre se base sur un modèle d'estimation multicible, où pratiquement seule la spécification du processeur change.

Les intérêts d'une méthode d'estimation d'un code assembleur optimisé sont multiples et dépassent le simple cadre du développement d'applications sur un processeur DSP.

- Tout d'abord, l'estimation d'un code assembleur optimisé représente un guide pour le programmeur lors de l'optimisation du code C. En effet, en comparant ces estimations avec les performances du code assembleur généré par le compilateur, il est possible de juger de l'efficacité de ce dernier mais aussi de déterminer quelles fonctions il est souhaitable de coder manuellement afin de respecter les contraintes temps-réel de l'application. Nous montrons cela dans le septième chapitre de ce rapport.

- Il est possible aussi sans aucune implémentation en assembleur de connaître rapidement si le DSP est en mesure d'exécuter en temps-réel l'application. Cette mesure est très utile pour les industriels car elle permet, tôt dans la conception du système final, de faire le choix du DSP cible le mieux adapté.

- Parfois, les contraintes sont telles qu'une solution logicielle doit faire place à une solution mixte matérielle/logicielle. Une expérimentation menée pour l'application G.721 [Rec84] sur le PineDSPCore et le OakDSPCore a montré l'intérêt d'une solution mixte [Peg98b]. La recherche de coprocesseurs accélérant l'exécution de certains traitements est beaucoup plus précise si elle se base sur des estimations d'une exécution optimisée sur le DSP.

- Les estimations logicielles sont également très utilisées par les outils de *co-design* afin de caractériser les éléments constituant leur bibliothèque de fonctions spécifiques (e. g. FFT) ou d'applications complètes (e. g. AC-3). Or, la qualité du partitionnement dépend en partie de la qualité des estimations logicielles (et matérielles). Si le partitionnement logiciel/matériel se base sur des estimations calculées à partir du code assembleur généré par le compilateur, il est fort probable d'aboutir à une solution sous-optimale. En effet, si le compilateur génère un code assembleur dont le temps d'exécution est deux fois supérieur à une implémentation manuelle, il est évident que l'outil de partitionnement aura tendance à allouer plus de tâches du coté matériel, augmentant ainsi la surface du

système final. Si au contraire l'outil de partitionnement dispose d'estimations logicielles d'un code assembleur optimisé, les solutions fournies ne pourront que se rapprocher d'une solution optimale. Notre outil d'estimation (VESTIM) est utilisé aujourd'hui pour fournir à l'outil de partitionnement CODEF [Bia98][Bia99a][Bia99b] (CODesign Environment Framework) des performances sur le OakDSPCore et le PalmDSPCore de VLSI Technology.

Structure du document

Le second chapitre de cette thèse présente tout d'abord les propriétés architecturales des processeurs DSP. On décrit ensuite la génération de code dans son ensemble, puis on aborde plus particulièrement les problèmes soulevés lorsque la cible est un processeur DSP.

Le troisième chapitre s'intéresse aux méthodes d'estimations logicielles existantes. Après une présentation des différents types de mesures de performances pour les systèmes embarqués, on décrit le modèle couramment utilisé pour calculer le temps d'exécution d'un programme. On introduit alors des techniques d'analyses statiques, dynamiques et mixtes qui permettent de déterminer le nombre d'exécutions de chaque instruction. On présente finalement des méthodes pour le calcul du nombre de cycles machine nécessaire à l'exécution d'une instruction.

Le quatrième chapitre présente les principales règles de réécriture du code C pour optimiser le code fourni par le compilateur, ainsi que le flot général de la méthode d'estimation. Le choix d'une approche basée sur l'exécution d'une séquence de test est justifié.

Le cinquième chapitre décrit le moyen utilisé pour récupérer les informations dynamiques puis le niveau de description intermédiaire sur lequel se base notre méthode d'estimation. A partir d'expérimentations effectuées sur différentes applications, nous montrons l'intérêt de modifier cette représentation intermédiaire afin de s'approcher d'un modèle de calcul orienté DSP.

Le sixième chapitre décrit en détail le modèle d'estimation multicible utilisé. On définit tout d'abord l'algorithme d'ordonnancement par liste, puis on montre qu'il est nécessaire d'adapter l'annotation du graphe de flots de données, la gestion de la liste des tâches ordonnançables ainsi que le traitement des composantes connexes aux propriétés du DSP cible. On décrit alors le modèle de description du processeur cible utilisé. On termine cette partie en décrivant le processus d'estimation à l'aide d'exemples représentatifs, en montrant en particulier comment sont modélisées les contraintes de parallélisme

Le septième chapitre regroupe les résultats des estimations obtenues pour le OakDSPCore en utilisant notre méthode d'estimation logicielle. Ces mesures concernent des applications complètes dans le domaine de la compression d'un signal de parole ou audio mais aussi d'une application modem. Elles mettent en évidence la précision des résultats mais aussi leurs dépendances liées en particulier aux modifications algorithmiques. La fin de ce chapitre est consacrée aux premières expérimentations effectuées sur le PalmDSPCore.

Ce mémoire se termine par une conclusion où nous essayons de tirer un bilan du travail présenté et de dégager des perspectives de recherche basées sur l'utilisation de méthodes d'estimations logicielles.

Les processeurs DSP

Les processeurs DSP sont des entités spécialisées dans le traitement d'applications de traitement du signal temps-réel comportant des calculs arithmétiques. Ces applications cibles opèrent de façon régulière et systématique sur les données. L'implémentation de ces deux propriétés (régularité et répétitivité) est caractéristique des processeurs de traitement numérique du signal.

Cette partie se propose de décrire les principales caractéristiques architecturales utilisées pour obtenir de hautes performances pour ce type d'applications. Nous verrons que l'évolution des architectures DSP est liée aux algorithmes de traitement du signal, mais aussi aux progrès technologiques récents qui permettent d'intégrer sur une puce de nombreuses unités fonctionnelles. Enfin, nous discuterons aussi des contraintes imposées aux programmeurs et aux concepteurs d'outils de haut niveau (compilateur en particulier) par les caractéristiques architecturales spécifiques des processeurs DSP. Cette étude se base principalement sur les articles de E.A. Lee [Lee88][Lee89] ainsi que des rapports d'analyse des performances d'un grand nombre de DSP [Bdt95] [Bdt99].

2.1. Architecture de base pour exécuter un programme de calcul

Les processeurs sont tous basés sur le modèle de Von Neumann (Figure 1) qui représente l'architecture simplifiée pour exécuter un programme de calcul.

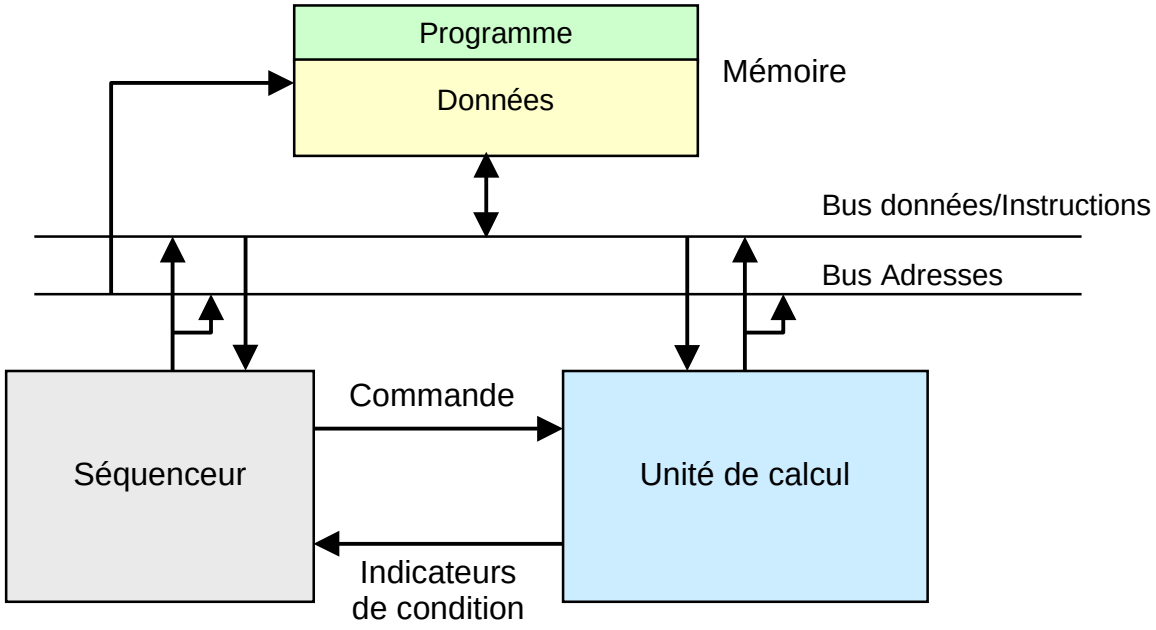


Figure 1 : Architecture Von Neumann

Cette architecture se compose de trois parties :

- une **partie opérative** qui effectue les calculs,
- un **séquenceur** qui contrôle l'exécution du programme,
- et une **mémoire unique** qui stocke les données, les résultats et les instructions du programme.

Les microprocesseurs traditionnels ont généralement une bande passante mémoire limitée et obtiennent de hautes performances avec des instructions utilisant des registres et des caches de données et instructions séparés. Or les applications de traitement du signal

opèrent sur de nombreuses données stockées en mémoire, le plus souvent sous forme de tableau. Afin d'améliorer les performances, le nombre d'échanges avec la mémoire a été augmenté en découpant celle-ci en plusieurs blocs.

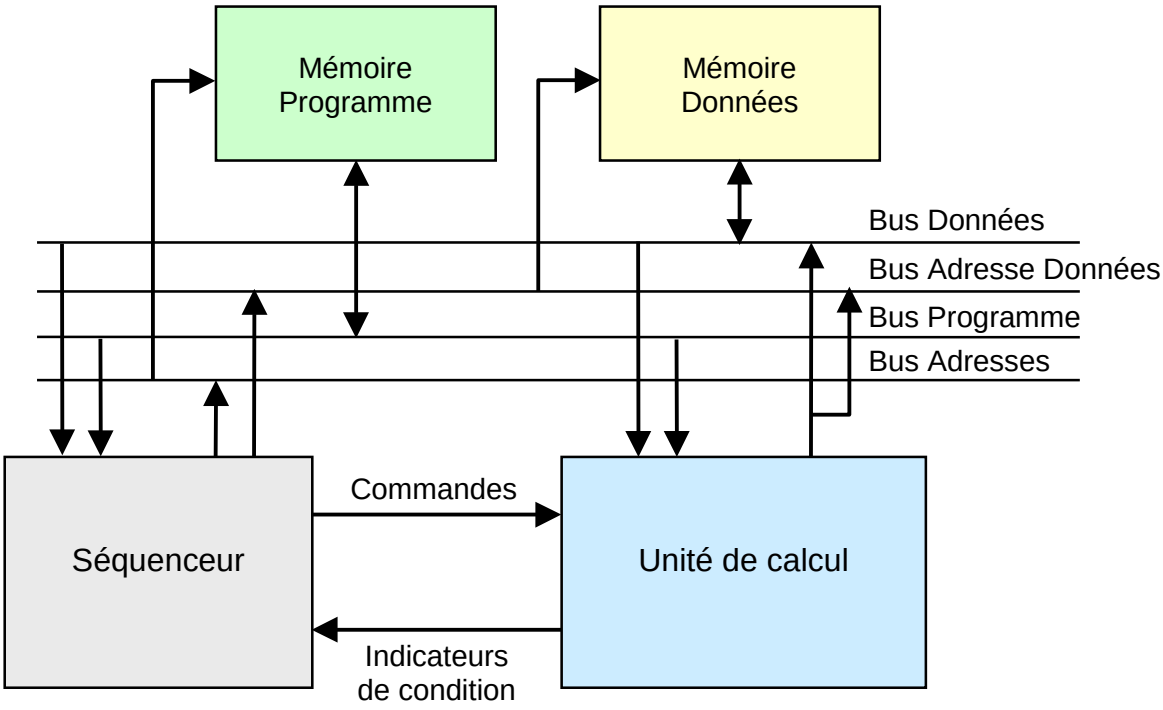


Figure 2 : Architecture Harvard

L'architecture Harvard de base, montrée sur la Figure 2, se compose ainsi d'une mémoire pour les instructions et d'une mémoire pour les données. L'architecture Harvard « modifiée » opère un découpage de la mémoire données en plusieurs bancs, de manière à augmenter encore le nombre d'échanges simultanés avec la mémoire. Les DSP n'utilisent en général pas de mémoires caches[2]. Ces processeurs ont en effet souvent une mémoire programme de taille limitée (quelques kilo octets) contrairement aux processeurs RISC. De plus, la fréquence d'horloge d'un DSP est généralement plus lente que celle d'un RISC ce qui permet l'emploi de mémoires plus lentes, adaptées à la vitesse du processeur. Enfin, les contrôleurs de cache conduisent à une augmentation de la surface de silicium par rapport à une approche multi-bancs.

On répartit généralement les coefficients (ou constantes) et les variables du programme dans deux bancs de mémoires différents, de manière à acheminer dans le même cycle une instruction et deux données. Ceci implique néanmoins l'existence de trois bus desservant les trois blocs mémoires et la gestion des adresses relatives à chacun de ces blocs. Le calcul des adresses pour les données est effectué parallèlement aux traitements par une unité spécialisée : le générateur d'adresse.

Le caractère systématique et répétitif des algorithmes de traitement du signal a été utilisé pour améliorer les performances de cette architecture par l'ajout d'un compteur de répétition et de toute la logique associée pour une gestion matérielle des boucles, c'est-à-dire sans pénalité temporelle, si ce n'est lors de l'initialisation de la boucle.

[2] Le DSP16xxx [Bdt99] de Lucent inclut un cache de 31 mots d'instructions alors que le ZSP16401 [Bdt99] possède un cache de données.

Enfin, une accélération de la fréquence de travail est obtenue en étalant sur plusieurs cycles d'horloge les différentes étapes d'élaboration d'un résultat. C'est la technique du pipeline dont le nombre d'étages varie d'un processeur à l'autre.

La plupart des DSP du commerce sont des déclinaisons de cette architecture mais comportent toujours les unités matérielles de base que nous venons de décrire et que nous détaillons dans le paragraphe suivant.

2.2. Caractéristiques architecturales des DSP

2.2.1. Représentation des nombres dans les DSP

Les DSP utilisent deux types de représentation numérique : la virgule (ou point) fixe et la virgule flottante. Le principal avantage des DSP à virgule flottante est que l'utilisateur n'a pas à se soucier de la dynamique et de la précision des nombres manipulés. Les processeurs à virgule flottante sont néanmoins plus coûteux, consomment plus et sont généralement plus lents que les processeurs en point fixe à technologie équivalente. Ces inconvénients font que les DSP en point fixe dominent le marché des processeurs embarqués.

2.2.1.1. Représentation en arithmétique point fixe

La programmation de processeurs en arithmétique point fixe demande une vigilance constante du programmeur afin de s'assurer que la dynamique et la précision plus restreinte des calculs ne nuisent pas à la fidélité du signal traité. Ceci passe par exemple par une normalisation des nombres pour s'assurer qu'il n'y a pas de dépassement de capacité (*overflow*). En arithmétique point fixe, les nombres sont codés soit en arithmétique entière entre une valeur maximale positive et une valeur minimale négative (complément à 2), soit en arithmétique fractionnaire entre -1.0 et +1.0 (non compris), soit comme une combinaison d'une partie entière et d'une partie fractionnaire (représentation Q_n). Dans la pratique, la plupart des DSP en point fixe supportent l'arithmétique fractionnelle et entière. La première est plus utile pour les algorithmes de traitement du signal alors que la seconde est adaptée aux opérations de contrôle ou de calculs d'adresse.

2.2.1.2. Largeur des données

La largeur naturelle des données[3] traitées par le processeur représente la largeur des données que les bus et les chemins de données du processeur peuvent manipuler en une seule instruction. Pour les processeurs en point fixe, la taille de données la plus commune est de 16 bits. Il existe cependant des processeurs dont la largeur des données est égale à 20 voire 24 bits. La taille des données a un impact important sur le coût du processeur car elle influence considérablement la taille du composant, le nombre de broches nécessaires, ainsi que le nombre et la taille des mémoires connectées au DSP. Le choix de la bonne largeur de données dépend en grande partie de la classe d'applications cibles et des contraintes de performances attendues. En effet, la complexité algorithmique (i.e. le nombre de MIPS[4]) et/ou le temps de codage augmente quand une application manipule des données avec une précision supérieure à celle du DSP. C'est ce qui ce passe généralement pour les applications audio (AC-3 par exemple [Ac3]) avec les DSP 16 bits points fixes.

[3] Définition utilisée dans [Bdt95].
[4] Million of Instruction Per Second

2.2.1.3. Extensions de la précision des nombres

Afin de répondre aux exigences de précision et d'éviter les dépassements de capacité, plusieurs mesures ont été prises pour les DSP. Tout d'abord, les accumulateurs et l'unité arithmétique et logique (ALU) utilisent généralement une largeur de données supérieure à la largeur naturelle. Pour le OakDSPCore [Oak96] par exemple, des bits d'extensions sont ajoutés : les accumulateurs et l'ALU ont une largeur de 36 bits. Il est ainsi possible d'effectuer jusqu'à 15 additions sur des données de 32 bits sans se soucier des problèmes de dépassement de capacité.

Ensuite, lorsque la taille de la donnée dépasse la largeur naturelle des données du processeur, le processeur peut être amené à effectuer des calculs en double précision de manière à garder une précision acceptable des signaux. Les calculs en double précision sont effectués soit en utilisant une suite d'instructions en simple précision (très long), soit à l'aide d'instructions spécifiques du processeur cible. Un exemple typique pour les DSP est l'ajout d'instructions dont les opérandes peuvent être signés ou non signés pour la multiplication en double précision. Néanmoins, l'utilisation d'instructions en double précision ne doit concerner qu'une faible partie de l'application. Dans le cas contraire il est préférable d'opter pour un processeur avec une largeur naturelle de données supérieure.

Enfin, les processeurs DSP disposent généralement d'unités de saturation à la sortie de l'ALU et entre les accumulateurs et le bus de données. Quand un dépassement de données se produit, l'unité de saturation affecte la plus grande ou la plus petite valeur possible. Ceci évite les erreurs de signe ou limite les effets des erreurs de calcul.

2.2.2. Les chemins de données

2.2.2.1. Evolution des architectures DSP

Hormis l'usage intensif des techniques de pipeline et de découpage des mémoires, les chemins de données ont été adaptés aux types de calcul des applications de traitement du signal. L'une des premières innovations architecturales typique des DSP fut l'intégration d'un multiplieur/accumulateur matériel rapide dans le chemin de données. Les calculs font plus généralement partie intégrante de l'exécution de chaque instruction. L'organisation de l'architecture est étudiée de manière à ce que le temps d'exécution d'une instruction soit égal à celui de l'unité de calcul matériel. Les DSP intègrent également des unités matérielles spécifiques, utiles dans certains traitements. Par exemple, un DSP possède généralement des unités dédiées à la détection d'exposant et aux décalages de longueur variable (*barrel shifter*). Ces deux unités sont très utiles lors de conversions de nombres entiers en format flottant (et inversement), très fréquentes pour les applications audio impliquant des calculs de transformée de Fourier.

Les DSP sont également capables d'activer en parallèle plusieurs unités fonctionnelles. Par exemple, lors d'une instruction de multiplication-accumulation (MAC), le multiplieur, l'ALU et les deux générateurs d'adresses sont activés en parallèle. Néanmoins, avec l'arrivée d'applications toujours plus complexes, ce parallélisme est devenu insuffisant pour une exécution en temps-réel. Les possibilités d'intégration qu'offrent les progrès technologiques ont permis ces dernières années de concevoir de nouvelles architectures DSP avec plus de ressources matérielles et donc plus de parallélisme. Les choix architecturaux sur l'organisation des chemins de données utilisant l'ensemble de ces ressources diffèrent

selon les constructeurs. On peut néanmoins dire qu'il existe deux tendances : l'approche VLIW et l'approche multi-MAC.

2.2.2.2. L'architecture DSP de base

Le premier type d'architecture DSP vise les calculs dont la base est la multiplication/accumulation, opération élémentaire de la plupart des transformations en traitement du signal. Pour un filtre à réponse impulsionnelle finie (FIR) l'algorithme est :

$$y(n) = \sum_{i=0}^{k-1} a(i) \times x(n-i)$$

La représentation correspondante sous forme de graphe flot de données (DFG) est montrée sur la Figure 3 pour un filtre FIR d'ordre 5.

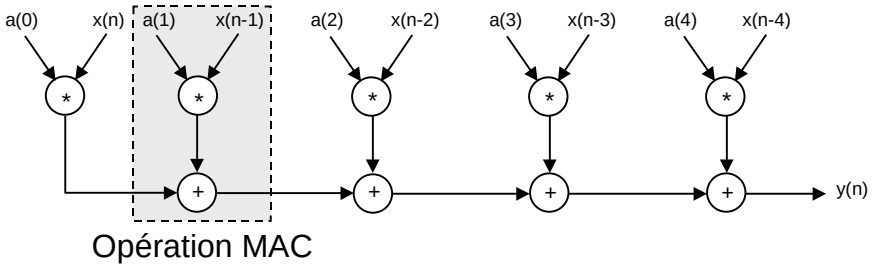


Figure 3 : Graphe flot de données d'un filtre FIR à 5 étages

Pour réaliser un filtre FIR à N étages en N cycles, le processeur doit effectuer dans le même cycle la lecture de deux données, la multiplication de ces dernières et l'addition (accumulation) avec le résultat précédent (opération MAC). Le schéma de principe de la partie opérative d'un DSP réalisant ce type de traitement est montré sur la Figure 4 :

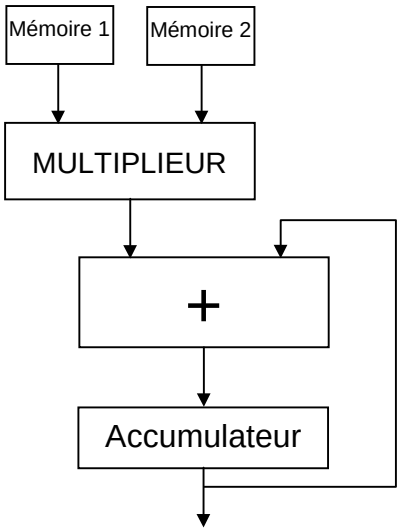


Figure 4 : Partie opérative d'un DSP de base

De nombreux DSP du commerce sont basés sur ce modèle d'architecture :

- Famille TMS320Cxx
- DSP16 et DSP32 de AT&T
- Famille ST18 de SGS-Thomson
- PineDSPCore et OakDSPCore de DSP Group

2.2.2.3. Extension adaptée aux calculs de la transformée de Fourier rapide

Les concepteurs de DSP ont cherché également à modifier les chemins de données afin d'optimiser les calculs relatifs à la transformée de Fourier, utile pour les applications qui travaillent dans le domaine fréquentiel. Les DSP adaptés à l'exécution de cette transformée utilisent l'algorithme rapide (FFT) de Cooley-Tukey qui s'applique lorsque le nombre d'échantillons N est une puissance de 2. La complexité de la transformée est ainsi réduite de N^2 à $N\log_2 N$. Il est fondé sur une fragmentation à base 2 au cours du temps par étapes successives. Le calcul rapide de cette transformée se compose d'une suite de papillons à deux entrées (radix-2) que l'on peut modéliser de la manière suivante (Figure 5):

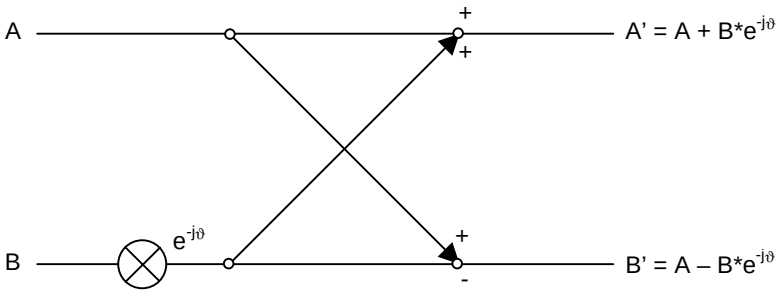


Figure 5 : Papillon de FFT radix-2 (Décimation en temps)

Si

$A = A_r + j*A_i$
$B = B_r + j*B_i$
$A' = A'_r + j*A'_i$
$B' = B'_r + j*B'_i$
$e^{-j\vartheta} = W_r + j*W_i$

Alors

$A_r' = A_r + (B_r*W_r - B_i*W_i)$
$A_i' = A_i + (B_r*W_i + B_i*W_r)$
$B_r' = A_r - (B_r*W_r - B_i*W_i)$
$B_i' = A_i - (B_r*W_i + B_i*W_r)$

Ainsi, quand les coefficients ne sont pas des valeurs singulières, le calcul de chaque papillon peut nécessiter :

- 6 opérations d'addition ou soustraction
- 4 multiplications
- 6 lectures mémoires
- 4 écritures mémoires

Le graphe flots de données de la Figure 6 représente une décomposition possible des traitements pour le calcul d'un papillon.

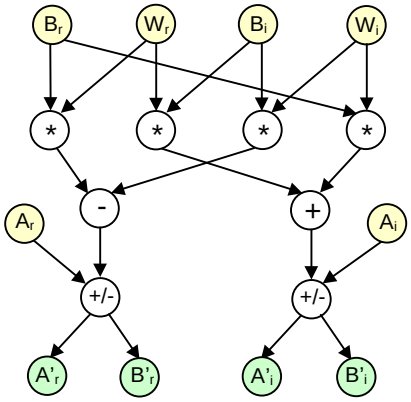


Figure 6 : Graphe de flots de données de la transformée de Fourier rapide

La partie opérative d'un DSP réalisant une FFT utilisant le DFG de la Figure 6 comme modèle de calcul est représentée par la Figure 7. On remarque que ce modèle d'architecture est une extension des chemins de données définis par la Figure 4 pour les calculs dont la base est la multiplication/accumulation.

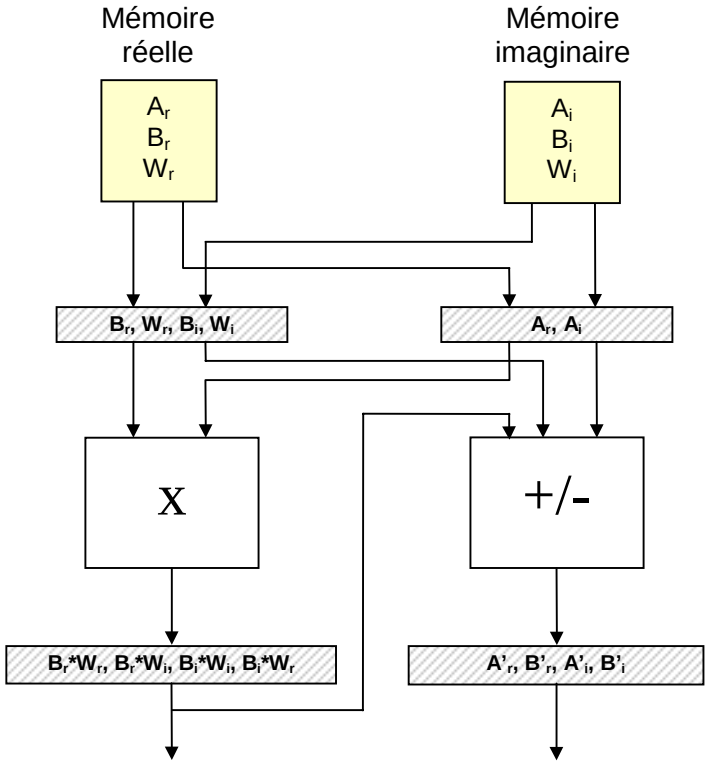


Figure 7 : Partie opérative d'un DSP optimisé pour le calcul de FFT

Les DSP du commerce basés sur cette architecture sont par exemple les Motorola 56000 et 96002.

Si l'on considère à présent que le processeur cible dispose d'une unité arithmétique et logique à trois entrées, on peut aboutir à un nouveau graphe flot de données montré sur la Figure 8.

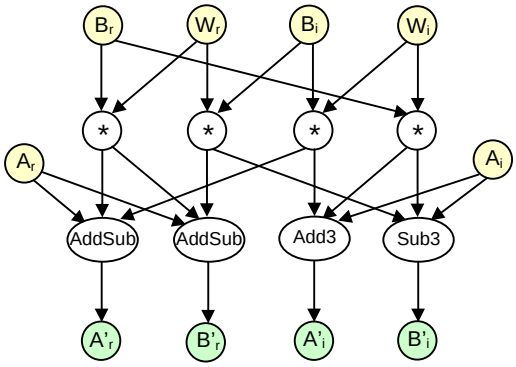


Figure 8 : Graphe de flots de données modifié de la transformée de Fourier rapide

Le PalmDSPCore [Ova98] conçu conjointement par DSP Group et VLSI Technology a une partie opérative basée sur ce modèle de calcul (Figure 9).

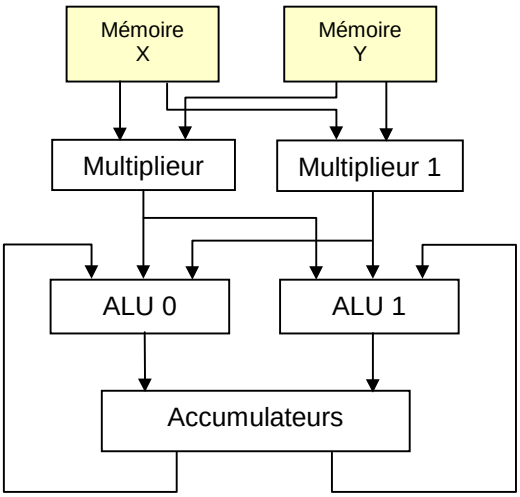


Figure 9 : Partie opérative simplifiée du PalmDSPCore

Remarque : Les accès mémoires (six lectures et quatre écritures au maximum pour le calcul d'un papillon) représentent le véritable goulot d'étranglement d'une implémentation rapide du papillon de FFT.

2.2.2.4. Les architectures orthogonales

Dans une architecture orthogonale les chemins de données permettent de mettre en relation n'importe lequel des registres avec n'importe laquelle des unités de calcul. Pour cela, les architectures de ce type disposent d'une file de registres assurant un espace d'échanges suffisant pour les unités fonctionnelles. La Figure 10 illustre le principe architectural de ce type de processeur. Les processeurs basés sur une architecture orthogonale ont pour objectif d'élargir encore l'éventail de leurs applications cibles. Leur comportement est en effet souvent très bon sur la plupart des algorithmes de traitement du signal.

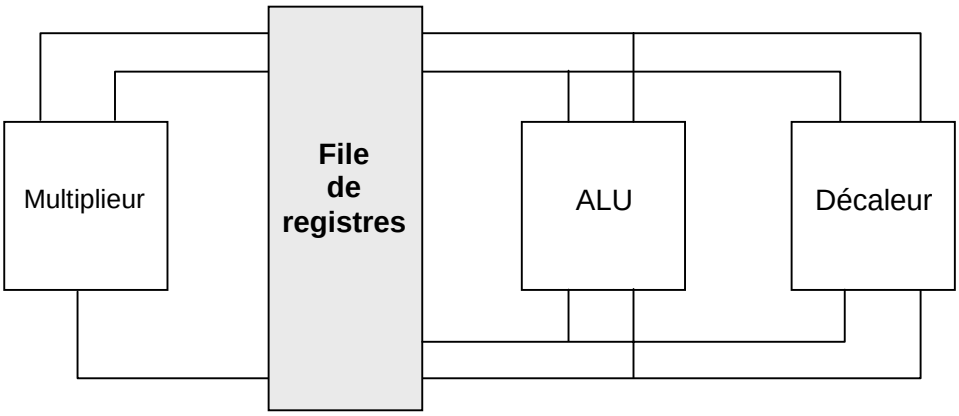


Figure 10 : Architecture orthogonale

La famille 21020 d'Analog Devices [AnDev] est basée sur ce modèle d'architecture. Une architecture orthogonale a également pour avantage d'être particulièrement adaptée aux techniques de compilation développées pour les processeurs RISC. Malheureusement, la file de registres et le nombre de bus reliant l'ensemble des unités de calcul augmentent considérablement la taille de la puce et donc son coût. De plus, les circuits basés sur ce modèle consomment plus d'énergie et fonctionnent (à technologie équivalente) généralement à une fréquence d'horloge inférieure (facteur deux typiquement) à celle des modèles de processeurs vus précédemment. Ces raisons expliquent leur faible utilisation dans les applications à fortes contraintes de coût et de consommation.

2.2.2.5. Les architectures à parallélisme étendu

Les DSP possédant des architectures à parallélisme étendu sont des processeurs possédant un plus grand nombre d'unités fonctionnelles que les processeurs de première génération définis précédemment. Ces DSP intègrent également de nouveaux chemins de données, plus de registres et d'avantages de fonctionnalités.

Les architectures multi-MAC

Les architectures à deux MAC (appelées « *dual-MAC* ») ont deux chemins de données en parallèle, chacun des chemins de données étant identique à celui décrit par la Figure 4. On obtient ainsi des modèles d'architectures semblables au modèle défini par la Figure 9. Les chemins croisés entre la sortie des multiplieurs et l'entrée des ALU permettent un gain de performance mais aussi une plus grande souplesse d'écriture des programmes. Les DSP du commerce basés sur ce modèle d'architecture sont par exemple le PalmDSPCore de VLSI Technology [Ova98], le Sabre (ou DSP16000) de Lucent Technology [Bdt99] ou le

LodeDSPCore de TCSI [Bdt95]. Le DSP SC140 [Ele99] est composé quant à lui de quatre unités de multiplication-accumulation en parallèle.

Les architectures VLIW

De manière schématisée, les architectures VLIW (**V**ery **L**ong **I**nstruction **W**ord) [Kie98] [Ell85] correspondent à des processeurs superscalaires[5] avec une partie contrôle de type microprogrammée. L'objectif de telles architectures est de rendre plus souple l'exploitation du parallélisme possible entre les unités fonctionnelles en fonction du parallélisme de l'application. Sur la Figure 11 la mémoire de microcode contient les commandes des différentes unités fonctionnelles : unités d'adressage, mémoires de données, ALU, multiplieur et séquenceur.

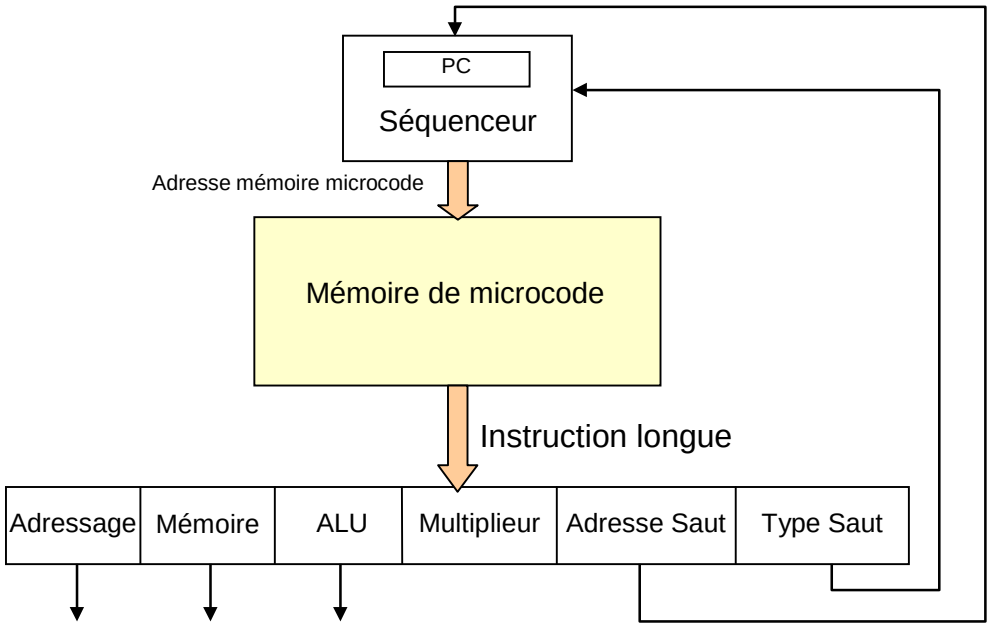


Figure 11 : Principe de l'architecture VLIW

Le rôle du compilateur est prépondérant puisque le microprogramme optimisé (issu de la phase d'ordonnancement) est le résultat de la phase de génération de code (instruction longue). Les architectures VLIW ont pour principaux inconvénients la taille de leur mémoire programme (rendant encore délicate leur utilisation dans des applications embarquées) et la complexité de conception d'un compilateur optimisé. Quelques références de processeurs DSP basés sur un modèle d'architecture VLIW sont le Trimedia [Sak98] et le R.E.A.L. [Kie98] de Philips, le Carmel [Suc98] de Siemens (en fait une déclinaison du modèle VLIW) ou le TMS320C62xx de Texas Instruments [Tur97]. Le cœur de l'architecture de ce dernier, montré sur la Figure 12, se compose de deux chemins de données points fixes, d'une unité de contrôle du programme et des interfaces mémoires. Chaque chemin de données possède quatre unités de calcul, une file de 16 registres généraux de 32 bits et des bus pour acheminer des données de la mémoire vers le chemin de données. En utilisant les unités d'exécution en parallèle, le processeur peut exécuter jusqu'à huit instructions 32 bits par cycle d'horloge. Les unités d'exécutions de chaque chemin de données se composent des unités L, S, M et D.

- L : ALU 40 bits supportant la saturation.

[5] Un processeur superscalaire peut exécuter plusieurs opérations par cycle.

- S : ALU 32 bits, décaleur de 32 bits, générateur de constante et instruction de branchements.
- M : Multiplieur 16x16
- D : Additionneur/Soustracteur 32 bits utilisé pour la génération des adresses (linéaire et modulo).

Notons la possibilité de liens croisés entre les deux chemins de données.

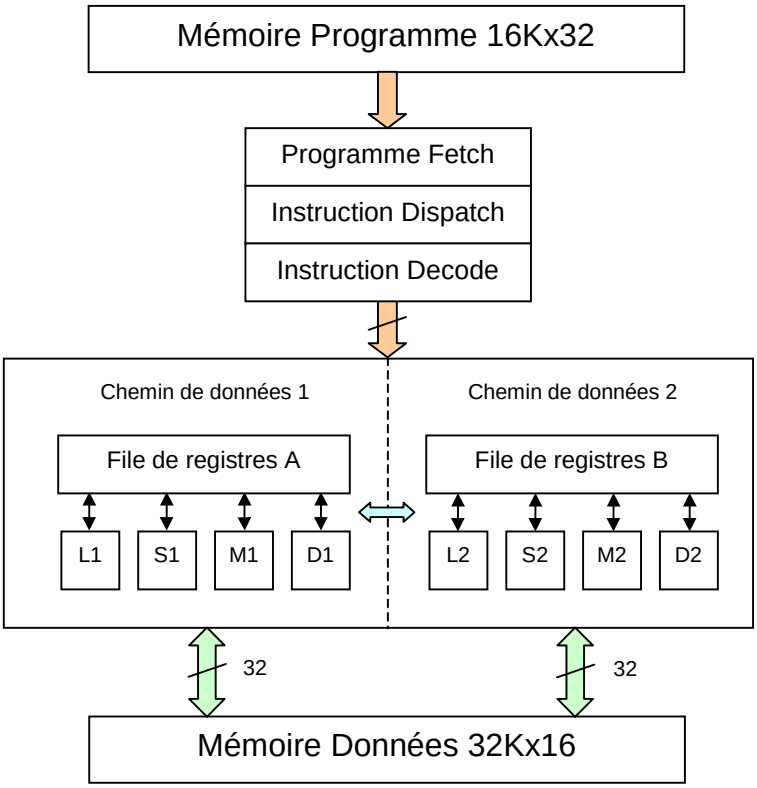


Figure 12 : Architecture du TMS320C62xx

2.2.3. L'architecture mémoire

2.2.3.1. Quelques déclinaisons du modèle de base

L'augmentation de la bande passante mémoire nécessaire à l'accélération des traitements peut prendre diverses formes. Les contraintes de coût (i.e. de surface) et de consommation des systèmes embarqués contraignent néanmoins les concepteurs dans leur choix. Les solutions retenues sont le plus souvent des mémoires multi-ports ou le découpage de la mémoire données en plusieurs bancs. La Figure 13 montre quelques déclinaisons de l'architecture Harvard de base rencontrées sur des processeurs DSP du commerce.

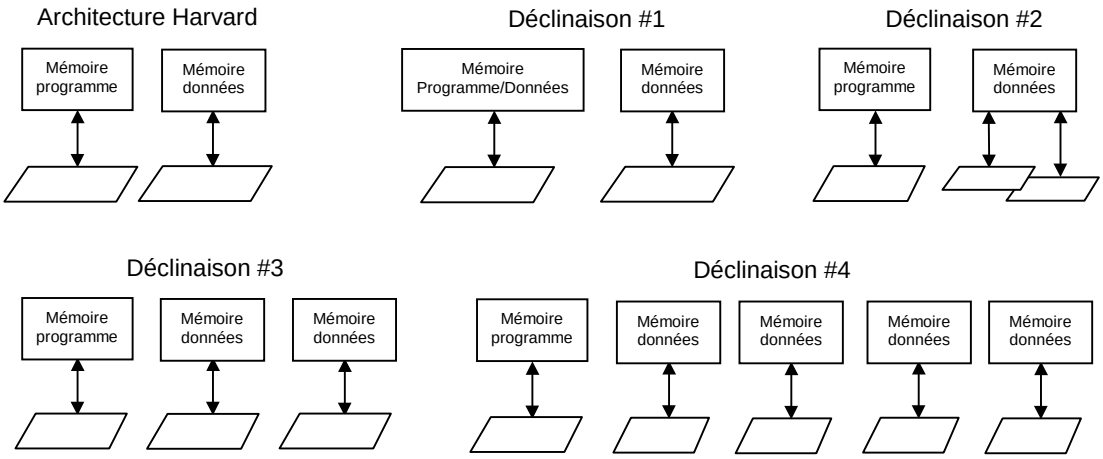


Figure 13 : Différentes alternatives d'organisation de la mémoire

Dans la déclinaison #1, le bus réservé au programme est partagé avec le bus réservé aux données (les coefficients généralement). Cette organisation mémoire implique qu'une instruction à deux opérandes mémoires nécessite deux accès à la mémoire données, plus un accès à la mémoire programme pour s'exécuter. Les processeurs TMS320C20 et TMS320C25 de Texas Instrument utilisent ce modèle d'organisation mémoire. Comme le temps d'accès à la mémoire est égal au temps de cycle d'une instruction, une instruction nécessitant deux opérandes mémoires s'exécute en deux cycles. Dans le DSP32 et DSP32C basé sur ce même modèle, le temps d'accès mémoire est égal à la moitié du temps de cycle d'une instruction de base. Il est ainsi possible d'exécuter en un cycle une instruction nécessitant deux, voire trois données en mémoire.

La déclinaison #2 utilisée dans le Fujitsu MB86232 se caractérise par une mémoire données multi-ports. Bien que perçue comme une solution coûteuse, l'avantage de ce modèle est de permettre des instructions multi-opérandes sans se soucier de séparer les données dans différents bancs de mémoires.

Une autre alternative est représentée par la déclinaison #3. Dans ce cas il est possible de lire une instruction et deux données simultanément en un cycle si le temps d'accès mémoire est égal au temps de cycle d'une instruction. Les architectures basées sur ce modèle sont par exemple le DSP56001 de Motorola ou le PineDSPCore et OakDSPCore de DSP Group. En contrepartie, le programmeur doit porter une attention toute particulière à l'arrangement des données en mémoire pour tirer au maximum profit des accès parallèles.

La déclinaison #4 utilisée dans le DSPi d'Hitachi ou le PalmDSPCore de VLSI Technology et DSP Group, utilise quatre bancs de mémoire données. Comme un cycle

d'instruction est égal à celui d'un accès mémoire, quatre données peuvent être lues dans le même cycle (parfois sous certaines conditions). L'accroissement du nombre de bancs de mémoire s'accompagne malheureusement d'une augmentation de la complexité pour distribuer efficacement les données dans les différents bancs. Ceci est valable pour le programmeur comme pour le concepteur du compilateur.

2.2.3.2. Mémoire interne ou externe ?

Les cœurs de DSP sont typiquement utilisés dans les applications à très fortes contraintes de coût et dont l'objectif est de fournir un système complet sur une puce (*system-on-chip*). Si plusieurs bancs de mémoires ou des mémoires multi-ports sont implémentés à l'extérieur de la puce, le nombre de broches de la puce nécessaire pour les bus devient excessif. Pour ces raisons la mémoire des DSP est généralement sur la puce. De manière générale, les DSP sont conçus avec suffisamment de mémoire interne dans chaque banc pour le type d'applications visées et les bus internes sont multiplexés vers l'extérieur. Mais l'utilisation d'un DSP pour une application différente peut nécessiter l'ajout d'une mémoire externe qui implique des temps d'accès plus longs que ceux relatifs à la mémoire interne. Dans ce cas la répartition du code et des données entre ces mémoires doit être optimisée.

2.2.4. Les modes d'adressage mémoire

Nous venons de voir que l'utilisation de plusieurs bancs de mémoire ou des mémoires multi-ports augmente la bande passante c'est-à-dire le nombre d'accès simultanés à la mémoire. Il reste le problème de spécifier (si possible dans la même instruction) l'ensemble des adresses pour accéder à ces données. Ceci est d'autant plus important que la charge de calcul des adresses dépasse largement la charge de calcul sur les données [Lie97] pour les applications de traitement du signal. Une solution universelle est l'utilisation d'un mode d'adressage indirect.

2.2.4.1. Mode d'adressage indirect

Dans ce cas la donnée est pointée en mémoire via un registre d'adresse (principe équivalent à ceux des pointeurs en C). Pour cela les DSP disposent d'un ensemble de registres d'adresse. Leur nombre est souvent très restreint afin de limiter le nombre de bits nécessaires au codage du registre dans l'instruction. La mise à jour des registres d'adresse correspondant à l'accès de la composante suivante (post incrément par exemple) est réalisée par un opérateur cablé : le générateur d'adresse. Ce dernier exploite la propriété de régularité des traitements.

La plupart des générateurs d'adresse actuels ne se limite pas à la simple opération de post incrémentation de l'adresse. Le mode d'adressage modulo permet de réduire le coût associé au traitement de tampon (i.e. *buffer*) circulaire (implémentant une ligne à retard) très utilisé dans les algorithmes de filtrage. La Figure 14 illustre le principe d'un buffer circulaire.

Ce buffer contient M+1 emplacements mémoires contigus commençant à l'adresse mémoire L et se terminant à l'adresse mémoire L+M. Quand un registre pointant sur L+M est incrémenté de un, son adresse suivante est L. De même, si un registre pointant sur L est décrémenté de un, ce dernier pointe alors sur L+M. Ce mécanisme est réalisé automatiquement par la plupart des DSP, c'est-à-dire sans l'ajout d'instructions, si ce n'est lors de la phase d'initialisation.

Le mode d'adressage indexé dont dispose de nombreux DSP est une forme d'adressage indirect. La donnée est accédée en mémoire via un registre de base auquel on ajoute un index. Ce mode d'adressage est particulièrement utile dans les algorithmes de recherche d'éléments particuliers dans un tableau, comme la recherche du meilleur index d'un dictionnaire dans la norme G.728 [Rec94].

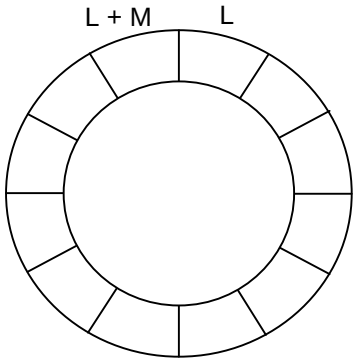


Figure 14 : Principe du buffer circulaire

Notons enfin que certains générateurs d'adresse possèdent également la logique nécessaire à la gestion automatique du mode d'adressage bit inversé utilisé pour accélérer les accès aux données après (ou avant) le calcul d'une FFT.

Remarque : La génération des adresses des instructions dans la mémoire programme est réalisée par le séquenceur.

2.2.4.2. Autres modes d'adressages

Le mode d'adressage mémoire indirect ne permet pas toujours des traitements optimisés. Les DSP possèdent généralement deux autres modes d'adressages.

- Des formes d'adressage immédiat : la donnée est placée dans l'instruction elle-même. Le mode d'adressage immédiat est utile pour le chargement d'adresse dans des registres.
- Des modes d'adressage mémoire direct où l'adresse de la donnée en mémoire est spécifiée dans l'instruction.

Cependant ces modes d'adressage requièrent généralement des instructions codées sur deux mots mémoires et s'exécutent en deux cycles. On parle de mode d'adressage immédiat (respectivement direct) *long*. Certains DSP peuvent néanmoins avoir des modes d'adressages direct et immédiat codés sur un seul mot mémoire et s'exécutant en un cycle machine. Dans le cas du OakDSPCore par exemple, le mode d'adressage direct *court* utilise pour cela une technique de mémoire paginée : la mémoire est divisée en page de 256 mots de 16 bits. L'adressage direct s'effectue alors en deux étapes :

- Spécification du numéro de la page stockant les données à traiter,
- Spécification du déplacement (*offset*) à l'intérieur de la page de la donnée à pointer.

Comme chaque page contient 256 données, le déplacement est codé sur 8 bits ce qui laisse 8 bits pour coder le reste de l'instruction.

2.2.5. Les structures de contrôle

Bien que les DSP ne soient pas des processeurs dédiés à l'exécution d'applications orientées contrôle (on laisse cela au processeur RISC), l'efficacité des structures de contrôle mises à la disposition du programmeur peuvent influencer considérablement les performances globales.

2.2.5.1. Les boucles matérielles

En premier lieu, tout DSP possède un compteur de répétition permettant d'utiliser la propriété de répétitivité inhérente aux applications de traitement du signal. Le compteur de répétition permet de répéter une séquence d'instructions sans y faire figurer d'instructions de contrôle de boucle. Pour cela, le compteur de répétition réalise sous forme cablée le décompte des itérations et la reprise en début de séquence tant que le décompte ne parvient pas à zéro. Ainsi, la perte de temps dans la gestion d'une partie itérative se limite aux cycles d'horloge perdus lors de l'initialisation de la boucle. Ce mécanisme est particulièrement rentable pour les boucles courtes fréquentes en traitement du signal.

2.2.5.2. Les instructions de branchement

La seconde structure de contrôle de base concerne les instructions de branchement. Ces instructions peuvent être d'autant plus coûteuses que le nombre d'étages de pipeline du DSP est élevé. Pour améliorer les performances liées aux problèmes d'aléas de pipeline (paragraphe 2.2.6.2), les DSP peuvent utiliser des techniques développées depuis de nombreuses années pour les processeurs à usage général. Les deux approches les plus fréquemment utilisées sont le branchement retardé et la prédiction de branchement. Notons que les DSP disposent généralement d'instructions conditionnelles qui permettent d'éviter l'utilisation d'instructions de branchement.

Enfin, une gestion efficace des interruptions peut aussi être un facteur non négligeable pour obtenir de bonnes performances, en particulier les mécanismes de sauvegarde et de restauration de contexte.

2.2.6. La structure pipeline

2.2.6.1. Mécanisme du pipeline

L'instruction MAC commune à tout DSP peut réaliser dans le même cycle la lecture de deux opérandes en mémoire, leur produit, l'addition avec l'accumulateur et le post incrément des deux registres d'adresse (plus éventuellement l'écriture du résultat en mémoire). Il est évident que si toutes ces opérations devaient s'effectuer séquentiellement à l'intérieur de l'instruction, le nombre de cycles d'horloge par instruction serait bien supérieur, ce qui limiterait très fortement l'intérêt des DSP. Une exécution rapide est obtenue en décomposant l'exécution d'une instruction en plusieurs étapes ou actions qui sont, pour un modèle simple :

- recherche de l'instruction : Instruction Fetch
- décodage de l'instruction : Decode
- lecture des opérandes : Operand Fetch
- exécution de l'instruction : Execute
- écriture du résultat : Modify

C'est le mécanisme du pipeline. La Figure 15 montre comment on peut structurer la partie contrôle du processeur : la structure pipeline est construite en plaçant des registres (les boîtes hachurées) qui définissent les étages du pipeline.

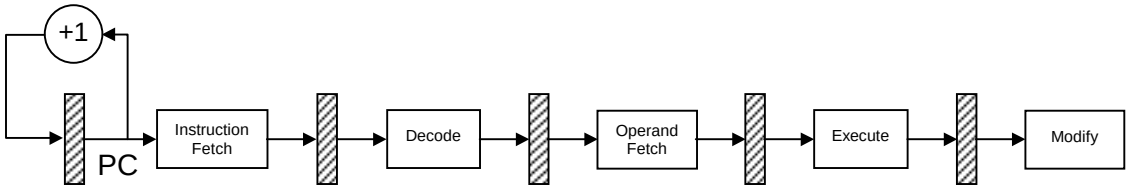


Figure 15 : Modèle simple de pipeline

Ainsi, la lecture des opérandes de l'instruction courante (*Operand Fetch*) s'effectue en même temps que le décodage de l'instruction suivante (*Decode*) ou de l'exécution de l'instruction précédente (*Execute*). De cette manière, les instructions sont entrelacées tout en laissant l'impression que chaque instruction se termine avant que la prochaine ne commence. Une autre manière de représenter le mécanisme du pipeline est montrée sur la Figure 16.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5				
Instruction Fetch	Insn1	Insn2	Insn3	Insn4	Insn5				
Decode		Insn1	Insn2	Insn3	Insn4	Insn5			
Operand Fetch			Insn1	Insn2	Insn3	Insn4	Insn5		
Execute				Insn1	Insn2	Insn3	Insn4	Insn5	
Modify					Insn1	Insn2	Insn3	Insn4	Insn5

Figure 16 : Structure pipeline à 5 étages

Au cycle 5, cinq instructions différentes sont en cours d'exécution, chacune à des étapes différentes du pipeline. Généralement la période du processeur est égale à la période d'un étage de pipeline et le temps d'exécution d'une instruction est égal à la profondeur du pipeline multiplié par la période du processeur.

2.2.6.2. Aléas de pipeline

Tant qu'il n'y a pas de dépendance entre une instruction et la précédente (aléas de contrôle ou de structure) ou de dépendance entre les opérandes (aléas de données), la totalité des étages de pipeline sont remplis. Dans le cas contraire, ces aléas conduisent à des pertes de performance. Réduire l'effet des aléas de pipeline sur les performances peut se gérer matériellement. La prédiction des branchements utilisée par le TMS320C6X [Tur97] par exemple est une solution matérielle puisqu'un tampon de prédiction de branchement est inclu dans l'étage « *Instruction Fetch* » du pipeline.

Mais il est aussi possible d'optimiser logiciellement l'utilisation des emplacements laissés vides par les aléas de pipeline [Goo97], par exemple par des techniques de pipeline logiciel (*software pipelining*). La technique du branchement retardé utilisée par le PalmDSPCore [Ova98] ou le ZSP16401 [Bdt99] fait partie de ces méthodes et consiste à exécuter l'instruction destination du branchement, N cycles après l'instruction de branchement. Il reste ainsi N-1 emplacements libres (*delay slot*) que le compilateur se doit d'utiliser aux mieux.

2.2.7. Jeu d'instructions des DSP

Les processeurs DSP utilisés dans les systèmes embarqués ont le plus souvent le programme de l'application qui réside en mémoire interne (sur la puce). De manière à réduire la surface de la puce mais aussi la consommation d'énergie liée aux accès à la

mémoire programme (dépendante de la largeur des bus), les concepteurs ont tendance à réduire la largeur d'un mot d'instructions.

En d'autres termes il s'agit de faire des choix lors de l'encodage du jeu d'instructions. Les principales approches utilisées à cet effet sont les suivantes:

- Réduction du nombre d'opérations : par exemple, le DSP16xx de Lucent Technologies ne possède pas d'instruction de rotation.
- Réduction du nombre de modes d'adressage : le nombre de possibilités de mise à jour dans un mode d'adressage indirect peut être limité par exemple. Il est également possible de limiter l'usage de certains modes d'adressage pour des classes d'instructions ou des instructions spécifiques.
- Restrictions sur les opérandes sources et destinations : par exemple, une instruction de décalage de l'accumulateur avec le TMS320C5x de Texas Instruments prend la valeur du décalage dans un registre dédié, et non dans un registre quelconque ou une valeur immédiate qui serait encodée dans le mot de l'instruction.
- L'utilisation de bits de mode est aussi utilisée par le TMS320C5x pour différencier les opérations de décalage logique et arithmétique. La valeur d'un bit dans un registre de contrôle détermine en effet le mode de décalage.

Ces techniques permettent de réduire le nombre d'instructions à encoder et donc la largeur des mots d'instruction. Une instruction qui est contrainte sur le choix de ses opérandes en entrée ou en sortie, ou sur les modes d'adressage est dite hétérogène. Par opposition, les processeurs à usage général sont dits orthogonaux (ou homogènes) car ils possèdent généralement un jeu d'instructions régulier et uniforme. Mais les processeurs orthogonaux ont des largeurs de mot d'instruction plus importantes car ils nécessitent un encodage indépendant des opérations et des opérandes à l'intérieur d'un mot d'instruction. Un inconvénient majeur de l'encodage du jeu d'instructions est de restreindre généralement le niveau de parallélisme offert par le processeur. En effet, l'encodage peut ignorer des chemins de données existants afin de réduire la largeur du mot d'instruction. Notons que le choix des instructions à encoder (i.e. déterminer un jeu d'instructions à partir d'une largeur de mot limitée) est un travail délicat. La méthode couramment utilisée consiste à utiliser des *benchmarks*[6] représentatifs des parties critiques de la classe d'applications visée par le processeur [Ziv96b].

Enfin, la réduction de la largeur du mot d'instruction passe aussi par l'utilisation d'un nombre réduit de registres spécialisés, contrairement aux processeurs à usage général qui ont une architecture plus régulière et utilisent des registres non spécialisés groupés dans une file de registres. On appelle registre spécialisé, un registre réservé à un usage particulier, c'est-à-dire pour lequel il existe des contraintes sur son utilisation. Par exemple, toute opération arithmétique ou logique exécutée par le OakDSPCore a pour registre destination les accumulateurs et uniquement ceux-ci. Plus généralement les DSP possèdent plusieurs types de registres spécialisés comme les registres d'adresse, les pointeurs de pile ou encore les registres pour les formes itératives. On dit aussi que les DSP (à l'exception de la famille 21020 d'Analog Devices [AnDev]) ont un jeu de registres hétérogène.

En conclusion, la réduction de la largeur d'un mot d'instruction limite le parallélisme potentiel du processeur mais rend également plus complexe la programmation du processeur. Enfin, comme nous le verrons plus en détail dans le chapitre suivant,

[6] Séries de tests

l'hétérogénéité est néfaste pour le développement de compilateurs efficaces pour des cibles DSP.

2.2.8. Présentation des cœurs de DSP OakDSPCore et PalmDSPCore

Les deux cœurs de DSP OakDSPCore [Oak96] et PalmDSPCore [Ova98] sont les processeurs sur lesquels portent plus spécifiquement l'étude. Un cœur de DSP est un processeur élémentaire ne comportant que les unités matérielles strictement nécessaires à l'exécution d'applications de traitement du signal. Les cœurs sont donc des macrocellules qu'un client peut intégrer dans son propre circuit avec d'autres blocs matériels définis par ses soins, comme par exemple un convertisseur analogique-numérique.

2.2.8.1. Le OakDSPCore

Le OakDSPCore [Oak96] est un cœur de processeur DSP 16 bits (largeur naturelle des données) à virgule fixe. Il est optimisé pour les algorithmes de recherche dans un dictionnaire (quantification vectorielle), de modulation/démodulation et de filtrage adaptatif. La partie opérative du OakDSPCore est identique au modèle décrit par la Figure 4. Le processeur est capable d'effectuer en un cycle une multiplication et une accumulation (instruction MAC). La mémoire donnée est divisée en deux bancs XRAM et YRAM (déclinaison #3 de la Figure 13), ce qui permet de lire deux données dans le même cycle si un mode d'adressage indirect est utilisé.

L'architecture du OakDSPCore, schématisée par le bloc diagramme de la Figure 17, se compose de trois principales unités qui sont actives en parallèle:

- L'unité de calcul (CBU) dont le bloc diagramme détaillé est présenté sur la Figure 18,
- L'unité de calcul des adresses (DAAU),
- L'unité de contrôle du programme (PCU).

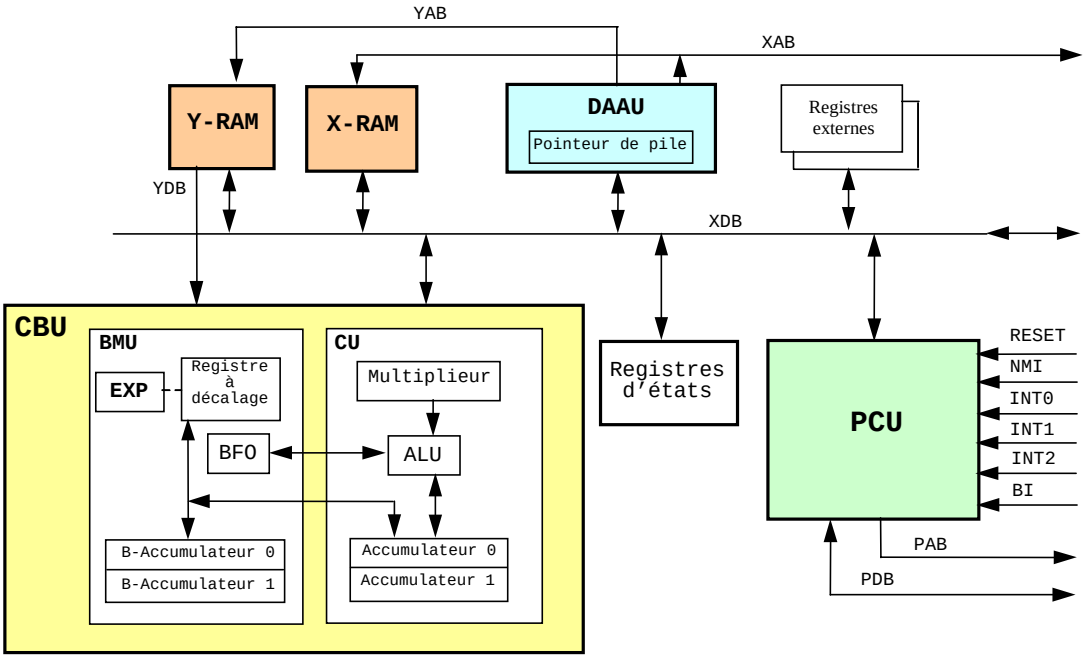


Figure 17 : Bloc diagramme du OakDSPCore

Le DAAU est l'unité dédiée aux calculs des adresses. Le OakDSPCore permet l'utilisation de quatre modes d'adressage différents :

- Immédiat,
- Direct,
- Indexé (où rb est le registre de base),
- Indirect en mode linéaire ou modulo. En fonction de la taille du tableau, il existe pour le mode modulo des contraintes sur l'adresse de base des données en mémoire.

Le DAAU possède également un registre de localisation du maximum ou du minimum d'un tableau (instructions min et max). La gestion des boucles est effectuée matériellement par le PCU. Le niveau maximum d'imbrication est quatre, et il existe deux types d'instructions de boucle : « rep » permet de répéter une seule instruction alors que « bkrep » permet de répéter plusieurs instructions. Le registre LC est un compteur initialisé au début de chaque boucle par le nombre d'itérations.

L'unité de manipulation de bits (BMU) possède deux blocs matériels très utiles pour les conversions flottantes et la normalisation des nombres entiers : le registre à décalage et l'unité de détection d'exposant (EXP sur la Figure 18). Utilisées conjointement, ces unités permettent d'effectuer une conversion flottante en seulement trois cycles. Les opérations sur les bits sont effectuées par BFO.

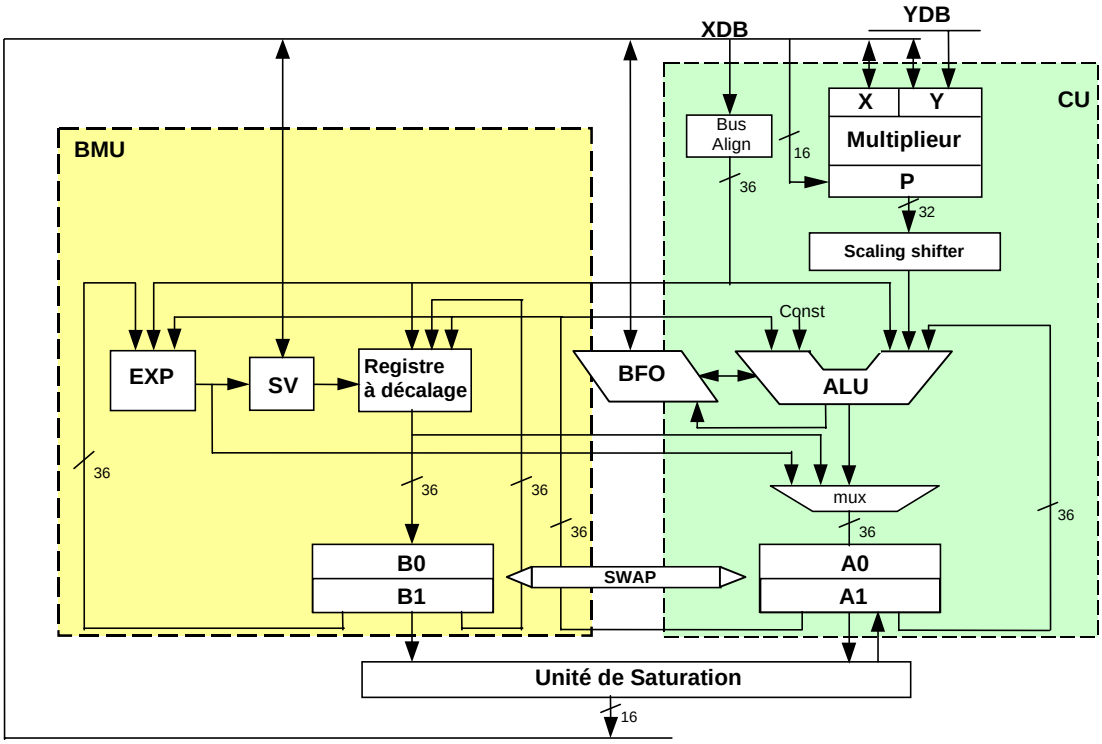


Figure 18 : Bloc diagramme détaillé de l'unité de calcul et de manipulation de bits

L'unité de calcul (CU) possède un multiplieur capable de différencier des opérandes signés et non signés. Ceci permet d'optimiser les multiplications en double précision, fréquentes dans les applications audio par exemple. Selon la valeur du registre PS (non représenté sur la Figure 18), le *scaling shifter* permet de décaler automatiquement le résultat de la multiplication (i.e. le registre P) de un bit vers la droite ou un ou deux bits vers la gauche avant son utilisation par l'unité arithmétique et logique. L'unité arithmétique et logique (ALU) effectue des calculs sur des données de 32 bits et dépose ces résultats dans l'un des deux accumulateurs (A0 ou A1). Notons qu'il n'existe pas de chemin direct du registre P vers la mémoire donnée.

Notons aussi que le OakDSPCore possède huit registres externes spécialement conçus pour une interface efficace avec l'extérieur du cœur, et plus particulièrement pour la conception de coprocesseurs [Peg98b].

Le OakDSPCore est un processeur comportant peu de parallélisme au niveau instruction. En fait, les instructions de type multiplication/accumulation (MAC) représentent le parallélisme maximum de ce processeur, à savoir : deux lectures simultanées en mémoire (adressage indirect obligatoire), une multiplication et une accumulation. Notons que deux lectures en mémoire sont possibles si et seulement si une donnée réside dans le banc XRAM, l'autre dans le banc YRAM. Le nombre d'étages de pipeline étant limité, les instructions ne sont pas contraintes par d'éventuels aléas de pipeline, ce qui simplifie l'écriture du code et le travail du compilateur.

Enfin, le OakDSPCore a été conçu pour les applications en télécommunication à faible consommation et avec de fortes contraintes de coût (i.e. surface). Pour répondre à ces exigences, les concepteurs ont conçu un DSP hétérogène tant au niveau de son jeu d'instructions que de ses registres. Ces derniers sont peu nombreux et généralement spécialisés. Les instructions quant à elles peuvent comporter de nombreuses différences sur les modes d'adressage possibles, la nature des opérandes ou l'emploi ou non de champ conditionnel. Ces choix architecturaux sont responsables en partie de l'inefficacité du compilateur C.

2.2.8.2. Le PalmDSPCore

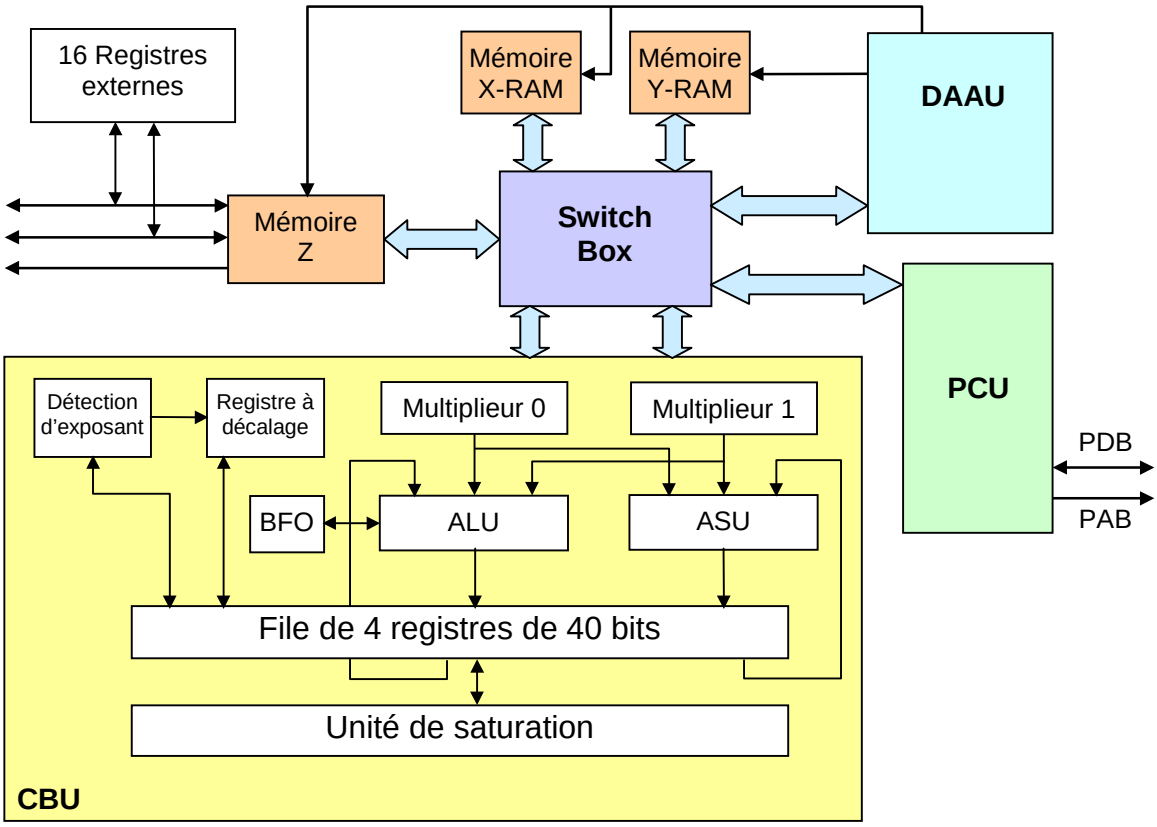


Figure 19 : Bloc diagramme du PalmDSPCore

Le PalmDSPCore [Ova98] est un cœur de processeur DSP 16, 20 ou 24 bits (largeur naturelle des données) à virgule fixe. Il a été conçu pour de plus vastes applications que le

OakDSPCore mais avec des contraintes de coût et de consommation toujours fortes. La partie opérative du PalmDSPCore est basée sur le modèle illustré par la Figure 9. Notons que l'ALU est une unité arithmétique et logique alors que l'ASU ne peut effectuer que des opérations d'addition ou de soustraction. Le processeur est capable d'effectuer en un cycle deux multiplication/accumulation (instruction MAC) et les chemins croisés à la sortie des multiplieurs peuvent être fort utiles (calcul d'une FFT par exemple).

Ce processeur a une compatibilité ascendante vis-à-vis du OakDSPCore. C'est pour cette raison que nous retrouvons les mêmes unités matérielles que dans ce dernier. Dans cette partie nous décrivons les nouveautés du PalmDSPCore par rapport au OakDSPCore.

L'organisation de la mémoire donnée est équivalente à la déclinaison #4 de la Figure 13[7]. En réalité, le PalmDSPCore ne possède que deux bancs de mémoire XRAM et YRAM, mais ces dernières peuvent être vues comme deux bancs distincts, un pour les adresses paires et l'autre pour les adresses impaires. Si les données sont à des adresses consécutives en mémoire, il est possible dans le même cycle de :

- lire quatre données ou de
- lire deux données et d'en écrire deux.

L'architecture du PalmDSPCore, schématisée par le bloc diagramme de la Figure 19, se compose, comme pour le OakDSPCore, de trois principales unités qui sont actives en parallèle:

- L'unité de calcul (CBU),
- L'unité de calcul des adresses (DAAU),
- L'unité de contrôle du programme (PCU).

Au niveau du DAAU, hormis un plus grand nombre de registres d'adresses (8 au lieu de 6), le PalmDSPCore permet l'utilisation d'un mode d'adressage bit inversé très utile pour le calcul de la FFT.

L'unité de calcul (CBU) est tout d'abord organisée autour de quatre accumulateurs de 40 bits regroupés dans une file de registres. Ceci permet une plus grande souplesse dans la programmation et facilite le travail du compilateur. Le PalmDSPCore possède également 16 registres externes. L'interface a été modifiée pour optimiser les transferts avec d'éventuels coprocesseurs. Le PalmDSPCore est un processeur comportant beaucoup de parallélisme au niveau instruction. Le nombre d'étages de pipeline étant égal à 6, la technique du branchement retardé est utilisée afin d'optimiser les performances. C'est au compilateur, ou en dernier lieu au programmeur, d'utiliser efficacement les emplacements libres. La sortie des deux multiplieurs possède des chemins croisés ce qui permet une plus grande souplesse d'utilisation. Les registres X0, Y0 et X1, Y1 sont les registres d'entrées des multiplieurs.

Le PalmDSPCore possède un jeu d'instructions moins hétérogène que le OakDSPCore car la taille d'un mot d'instruction est passée à 32 bits (contraintes d'encodage moins fortes). En particulier, les instructions utilisent plus régulièrement les différents modes d'adressage. L'organisation en file de registres des accumulateurs amène aussi une plus grande orthogonalité.

Enfin, le PalmDSPCore possède des instructions très spécifiques pour :

[7] La mémoire Z est plus lente et est dédiée aux entrées-sorties.

- Le calcul de la FFT
- Le calcul de Viterbi
- La quantification de vecteurs
- Etc.

2.2.9. Conclusion

L'extension du marché des DSP a entraîné une spécialisation de ces processeurs pour les besoins d'applications spécifiques (ou classe d'applications). La spécialisation se caractérise le plus souvent par l'inclusion de coprocesseurs, l'ajout de nouveaux chemins de données ou la définition de nouvelles instructions qui augmentent les performances pour certains types d'algorithmes. Le degré de spécialisation varie d'un processeur à un autre. Il peut être important comme dans le cas du Motorola DSP56305, clairement dédié aux stations de bases GSM, ou encore pour les membres de la famille de processeurs ZR38xxx de Zoran conçus exclusivement pour la décompression des applications audio AC-3. Au contraire, les processeurs de la famille du TMS320C54x de Texas Instruments ou le OakDSPCore de DSPGroup sont des processeurs adaptés aux applications de télécommunication sans fil sans pour autant être spécifiques à une application particulière.

La spécialisation des processeurs DSP, même pour une classe d'application, s'accompagne généralement d'une augmentation de la complexité et d'une hausse de la spécificité des unités matérielles, des chemins de données ou des modes d'adressage. Cette spécialisation est malheureusement en partie à l'origine des problèmes rencontrés par les compilateurs pour générer un code assembleur efficace.

Une autre tendance est la prolifération de nouvelles architectures pour les processeurs de traitement du signal. Non seulement les processeurs à usage général intègrent des unités matérielles afin d'exécuter plus efficacement les algorithmes de traitement du signal, mais les processeurs DSP intègrent aussi des fonctionnalités pour optimiser le traitement de tâches normalement prises en compte par les processeurs RISC. Les exemples suivants sont représentatifs.

- Le DSP 568xx de Motorola inclut des caractéristiques de microcontrôleurs.
- Les extensions MMX du Pentium d'Intel apportent aux architectures 80x86 les performances des DSP en points fixe.
- Le SH-DSP d'Hitachi et le HyperStone sont des processeurs hybrides RISC/DSP.
- Le Picollo, un coprocesseur DSP conçu pour fonctionner avec le cœur de microcontrôleur RISC ARM7 de ARM Ltd.
- Le TMS320C62xx utilise une architecture VLIW afin d'obtenir un degré de parallélisme important, inhabituel pour les algorithmes de traitement du signal.

La diversité des applications de traitement du signal offre de nombreuses possibilités d'approches architecturales et ceci devrait continuer. Cependant, cette diversité rend encore plus difficile le reciblage des compilateurs en vue d'une compilation efficace pour une classe de processeurs. Un compilateur reciblable est plus que jamais réservé à une famille restreinte d'architecture. En contrepartie, la prolifération de nouvelles architectures ainsi que l'augmentation de leur complexité renforcent l'intérêt de développer des outils efficaces. Il paraît en effet difficile d'utiliser certains processeurs DSP introduits récemment sans outil de qualité. Ceci est plus particulièrement vrai pour les compilateurs. Le TMS320C62xx fait partie de ces processeurs pour lesquels sans outil de compilation efficace [Bdt99] la production d'un code optimisé est une tâche fastidieuse et peu envisageable dans de nombreux cas.

Génération de code pour DSP

2.3. Les principales raisons d'inefficacité des compilateurs C actuels pour DSP

De nombreuses études et expérimentations sur différents types de processeur DSP [Lev97][Ziv94][Wil96] ont été menées ces dernières années mettant en évidence les problèmes d'inefficacité de la génération de code des compilateurs C pour ces processeurs. Nous avons mené également nos propres expérimentations avec le Oak C Compiler (OCC) [Occ96] amenant aux mêmes conclusions. Nous distinguons trois raisons principales à l'inefficacité des compilateurs C actuels pour DSP : le langage de description utilisé, leur cycle de développement et l'héritage RISC.

2.3.1. Le langage C

2.3.1.1. Les problèmes pour décrire une application de traitement du signal en C

Le choix du langage de haut niveau permettant de décrire l'application est un facteur important pour une compilation efficace. D'autres langages de spécification que le C ont été spécifiquement développés pour décrire les algorithmes s'exécutant sur un processeur DSP en point fixe. Citons par exemple Silage [Kal93] [Hil90] ou DFL (Data Flow Language) [Can94] utilisés respectivement dans Ptolemy et la DSP Station de Mentor Graphics. Le langage C s'est néanmoins imposé comme le langage de description des applications de traitement du signal. En effet, ce dernier est très généralement utilisé par les instituts de normalisation pour décrire les applications de traitement de signal. Par exemple, l'ITU (International Telecommunication Union) ou l'ETSI (European Telecommunication Standard Institute), deux organismes de normalisation, accompagnent le plus souvent la spécification textuelle d'une norme d'un code écrit en C. L'avantage de ce langage est sa popularité auprès des programmeurs, sa portabilité et sa maintenance.

Malheureusement le langage C n'a pas été conçu pour décrire des applications de traitement du signal. Par exemple, un des problèmes majeurs du C est son incapacité à représenter des données fractionnaires, ce qui rend impossible l'utilisation d'opérateurs classiques des DSP sur des données de ce type. La localisation des données dans les différents bancs mémoires d'un DSP ou la description d'un mode d'adressage modulo sont d'autres problèmes classiques rencontrés avec ce langage pour décrire une application. Or les constructions disponibles dans un langage de haut niveau constituent aussi un facteur important pour une compilation efficace. Des extensions orientées DSP ont été apportées par les constructeurs au C-ANSI afin d'améliorer l'efficacité des compilateurs. Il est à noter que ces extensions n'ont pas encore été normalisées par la communauté du C-ANSI, limitant l'utilisation par les industriels d'extensions communes pour les processeurs DSP.

2.3.1.2. Les extensions du C pour DSP

L'objectif de telles extensions est de spécifier au niveau du C des caractéristiques de l'implémentation sur DSP. De nombreuses tentatives ont été menées ces dernières années pour doter le C ANSI d'extensions permettant de générer un code assembleur efficace. Citons par exemple Intermetrics pour NEC [Kre94], Analog Devices [Hof93], Texas Instruments [Yos96], Philips Semicondutors [Hor98] ou DSP Group [Occ96].

Pour le OakDSPCore, le fait de déclarer des données dans deux bancs mémoires différents (XRAM et YRAM) permet de faciliter la génération d'une instruction MAC. De même l'utilisation du mot clé « bkrep » apporte une aide au compilateur dans la prise en compte des boucles matérielles.

On énumère ci-dessous des extensions significatives que l'on a pu trouver pour différents DSP et détaillées dans [Yos96][Hof93][Occ96][Kre94][Hor98]:

- plusieurs bancs de mémoires (XRAM et YRAM),
- placement des données en mémoire à une adresse précise,
- type de données point fixe,
- type de données accumulateur (36 ou 40 bits),
- buffers circulaires et mode d'adressage modulo,
- instructions de boucles matérielles,
- nombres complexes,
- extensions pour architectures parallèles ou vectorielles.

Toutes ces extensions donnent à l'utilisateur un meilleur contrôle sur le code écrit en C et dédié à être compilé sur DSP. Dans un article paru dans EDN en juin 1997 [Lev97], l'auteur compare, pour huit DSP différents, les performances (en nombre de cycles et en taille mémoire programme) d'un code assembleur généré à partir soit d'un code C « *out-of-the-box* », c'est-à-dire sans modification, soit d'un code C utilisant toutes les possibilités d'extension offertes par le compilateur. Les optimisations apportées au code C sont de deux sortes : soit on utilise des extensions comme la localisation de données dans différents bancs de mémoire, soit on effectue des modifications algorithmiques comme le dépliement de boucle. Les tests sont réalisés avec trois types d'architectures distinctes.

- Pour les DSP de type filtrage comme le OakDSPCore ou le Motorola 56002, le bénéfice de l'utilisation de ces extensions est indéniable puisque, en moyenne, le nombre de cycles est divisé par quatre et la taille du code est en baisse de 16,3%.
- Pour le TMS320C6X qui est un DSP 32 bits de type VLIW, le nombre de cycles est divisé par trois en moyenne sur l'ensemble des applications testées. Par contre la taille du code augmente de 45% en moyenne. Ceci est dû aux techniques utilisées pour améliorer les performances. Dans le cas d'une DCT utilisée par la norme JPEG par exemple, les boucles les plus imbriquées sont dépliées ce qui augmente significativement leur taille ainsi que celle des blocs précédents (préambules) et successeurs (postambules) de la boucle.
- Enfin les tests sont réalisés sur deux DSP de type virgule flottante, le TMS320C30 de Texas Instruments et le SHARC d'Analog Devices. Pour ces deux processeurs le nombre de cycles est divisé en moyenne par 1,58 alors que la taille du code augmente de 28,5% pour le TMS320C30 et de 1,5% pour le SHARC.

Ces résultats montrent que les extensions apportées au C permettent d'obtenir un code beaucoup plus efficace. L'auteur remarque cependant fort justement que plus le code C est modifié, moins le compilateur est bon. En effet, l'ajout des extensions nécessaires à la localisation des données dans les différents bancs mémoire par exemple, montre que le compilateur n'est pas en mesure d'effectuer ce travail seul. On pourrait cependant s'attendre à trouver ce type d'optimisation dans les compilateurs dédiés aux DSP. De plus, on peut noter un certain nombre d'inconvénients à une approche basée sur la modification du code source.

- La lisibilité du code diminue et la portabilité n'est plus forcément garantie.

- Le temps nécessaire à la localisation des parties de code critiques et à la modification du code source correspondant n'est pas négligeable.
- Une connaissance des mécanismes de génération de code du compilateur mais aussi de l'architecture cible est nécessaire.

Enfin, notons que l'utilisation des extensions du C permet pour des algorithmes simples de type filtre à réponse impulsionnelle finie (FIR) de générer un code optimal [Kre94]. Malheureusement, pour certains compilateurs les résultats sont très décevants sur des applications complètes[Wil96][Pau97]. Malgré tout, un code C est toujours plus facile à écrire et à comprendre qu'un code assembleur. Ceci devient d'autant plus vrai que les architectures DSP actuelles deviennent de plus en plus complexes.

2.3.2. Le cycle de développement des compilateurs C pour DSP

L'origine de l'inefficacité des compilateurs C pour DSP provient aussi en partie du cycle de développement de ces derniers [Ziv96b][Sag94], ceci pour les raisons industrielles invoquées dans le premier chapitre. En effet, l'approche courante consiste à développer tout d'abord entièrement l'architecture cible DSP, de manière à obtenir les meilleures performances pour le type d'applications visées. Une fois la conception du circuit terminée, les concepteurs du compilateur essaient d'exploiter de la meilleure façon possible l'architecture cible. Malheureusement, le résultat de cette approche séquentielle est le plus souvent une solution sous optimale.

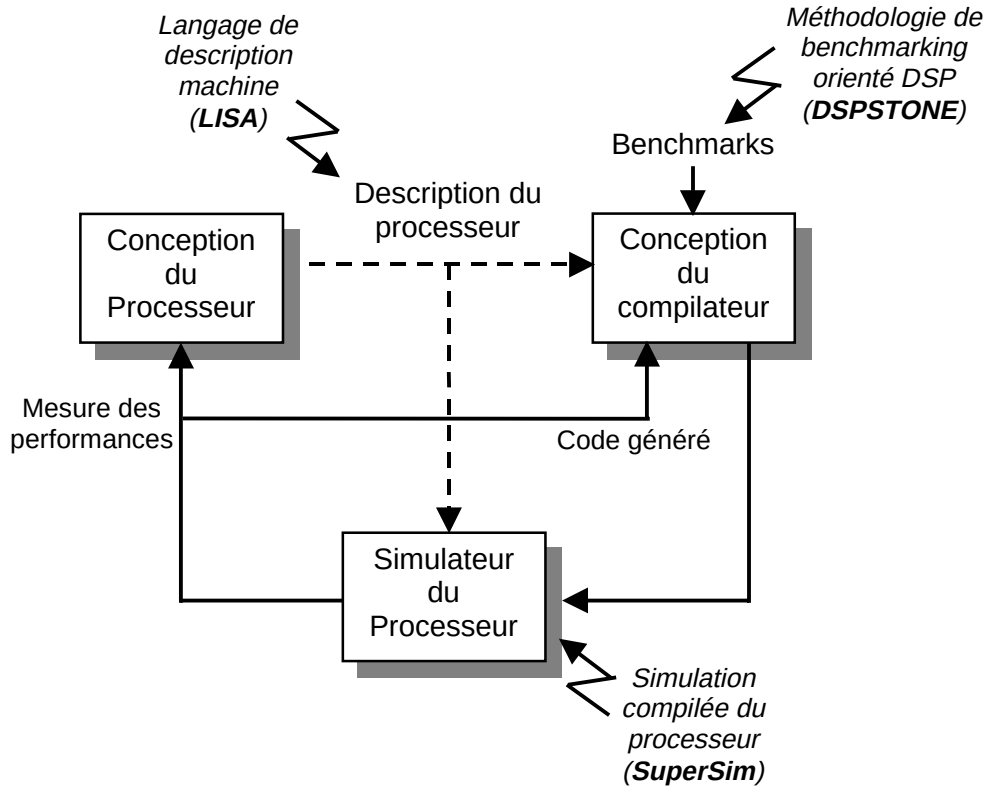


Figure 20 : Flot de conception simultané compilateur - processeur

Afin de tirer profit des caractéristiques architecturales spécifiques des DSP et d'obtenir un code assembleur optimisé, il est indispensable de concevoir en parallèle l'architecture et le compilateur. C'est l'approche utilisée depuis les années 80 pour la conception de processeurs à usage général et plus particulièrement RISC [Pat82]. C'est la principale stratégie pour obtenir les meilleurs performances processeur/compilateur. Pour un

processeur RISC il n'est pas rare en effet, que le code assembleur généré par le compilateur soit meilleur qu'un code écrit à la main. La conception simultanée du processeur et de son compilateur est généralement guidée par des statistiques de temps d'exécution du code généré pour des applications représentatives [Hen90], ce qui permet d'évaluer l'impact des choix architecturaux et des optimisations apportées au compilateur. Le processus est itératif afin d'affiner petit à petit la qualité du code généré. Des méthodes similaires pour des systèmes embarqués ont été menées récemment [Hor98]. Le principe du flot de conception simultané du processeur et du compilateur développé en [Ziv96b] est exposé sur la Figure 20.

Le développement simultané du processeur et du compilateur est dirigé par la mesure de performance délivrée par le simulateur du processeur. L'outil dispose pour cela d'un simulateur compilé, SuperSim[Ziv96a], permettant d'accroître de manière significative les temps de simulation souvent très longs des simulateurs interprétés. Grâce à des mesures quantitatives précises de l'efficacité du compilateur fournies par l'outil DSPSTONE[Ziv94], on détermine si le compilateur représente une alternative raisonnable par rapport au codage manuel d'applications. Notons que cet outil analyse également les fréquences d'exécution pour des classes spécifiques d'instructions. Il est ainsi montré que l'inefficacité du compilateur n'affecte pas de la même façon les différentes classes d'instructions. Si le processeur a été modifié, la nouvelle description du processeur est utilisée pour mettre à jour le compilateur. Pour cela, l'outil utilise un langage de description du processeur, LISA[Pee99], suffisamment générique pour couvrir l'ensemble des classes de processeurs visés.

Une approche similaire a fait l'objet de recherches à l'Université de Toronto [Sag94]. Cependant ces travaux se distinguent de ceux de Zivojnovic et al. [Ziv96a] par le choix d'un modèle d'architecture cible adapté aux contraintes d'une compilation efficace. Les auteurs ont ainsi conçu une architecture cible avec pour principal objectif de faciliter le travail du compilateur dans la génération d'un code efficace. Pour cela ils ont opté pour une architecture simple, fortement orthogonale, avec des opérations registres à registres et permettant au compilateur d'exploiter le parallélisme au niveau instruction. Afin de satisfaire ces objectifs, leur choix s'est porté sur une architecture de type VLIW disposant des principales fonctionnalités nécessaires aux DSP. Un simulateur de niveau instruction a été développé afin de fournir des mesures sur les temps d'exécution d'applications représentatives guidant le développement conjoint du compilateur et de l'architecture. Le compilateur basé sur le GNU-CC a été modifié afin de tirer profit des caractéristiques spécifiques du processeur et du parallélisme au niveau instruction. Un optimiseur de code dédié a également été développé en marge de l'utilisation d'algorithmes d'optimisation classiques [Aho86]. Les tests montrent une accélération moyenne du temps d'exécution de 4.86 pour des algorithmes de base de traitement du signal (FIR, FFT, etc.) et de 2.83 pour des applications complètes (ADPCM, …) par rapport aux performances obtenues sans l'utilisation de l'optimiseur de code dédié. Malheureusement, le modèle d'architecture VLIW a été choisi pour ces performances mais avec peu de considération des critères de coût et de consommation. De plus, les algorithmes d'optimisation du code développés pour ce type de processeur ne s'appliquent pas à une architecture DSP hétérogène.

2.3.3. L'héritage RISC

Les techniques de compilation classiques utilisées pour les processeurs à usage général RISC [Aho86] ont été développées dans les années 80. Ces techniques sont basées sur un modèle d'architecture homogène caractérisé par des registres à usage général et un modèle de transfert de données avec la mémoire basé sur ces registres (modèle « load-store »). Or

les DSP possèdent généralement des architectures différentes : jeu de registres restreint et hétérogène, modes d'adressage complexes, plusieurs bancs de mémoires, opérateurs dans les chemins de données. L'héritage RISC représente donc cette inadéquation entre les architectures DSP et les techniques de compilation développées pour des processeurs RISC. C'est le cas du compilateur OCC pour le OakDSPCore comme de nombreux autres[8] basés sur le compilateur GNU-C [Sta94] qui fut typiquement développé pour des processeurs à usage général. Lee [Lee88][Lee89] fut l'un des premiers à décrire les problèmes associés à la génération de code pour les processeurs DSP par rapport aux techniques développées pour les processeurs RISC.

Dans les paragraphes suivants nous présentons brièvement les techniques de compilation classiques ainsi que les optimisations développées spécifiquement pour un modèle d'architecture DSP. Pour cela nous décrivons dans un premier temps les contraintes imposées par ce type de processeur dans un système embarqué.

2.4. Techniques de compilation actuelles

2.4.1. Les challenges d'une compilation DSP

Les contraintes des outils de compilation pour les processeurs embarqués de type DSP sont différentes de celles des processeurs à usage général [Cra97][Ara95a]. Voici ces principales contraintes.

a) Générer un code performant, c'est-à-dire respectant les contraintes temps réel fortes. Si le compilateur n'est pas en mesure de générer un code assez efficace, alors un processeur plus rapide doit être utilisé de manière à respecter ces contraintes. Mais un processeur plus rapide est généralement plus cher et consomme plus d'énergie, ce qui peut être inacceptable dans le cas d'applications sans fil par exemple. Notons que la durée de compilation n'a que peu d'importance dans le cas des DSP, ce qui permet de reconsidérer des algorithmes de compilation rejetés jusqu'alors pour leur trop grande complexité.

b) Générer un code compact afin de respecter les limitations de taille mémoire programme des systèmes embarqués.

c) Offrir un support pour les algorithmes de traitement du signal : le programmeur doit être en mesure de spécifier ces algorithmes dans un langage de haut niveau supportant la définition de nombres en précision finie ou des opérateurs de saturation par exemple.

d) Le compilateur doit exploiter les caractéristiques architecturales essentielles des DSP [Lee88][Goo97].

- Un jeu de registres restreint et hétérogène.
- La prise en compte du parallélisme. Pour le OakDSPCore par exemple, le compilateur doit être en mesure de générer une instruction MAC. Ceci est encore plus vrai pour les architectures de type VLIW comme le TMS320C6X [Tur97] ou le R.E.A.L [Hor98]. La difficulté des compilateurs actuels à prendre en compte ce parallélisme est une autre source d'inefficacité.
- Support pour les instructions spécifiques. Les compilateurs doivent être en mesure d'exploiter les caractéristiques matérielles spécifiques des architectures DSP.

[8] Analog Device 2101 [AnDev], AT&T 1610 ou Motorola 56001 [Mot56k]

e) Un compilateur reciblable (*retargetable*). Beaucoup de compilateurs DSP sont aujourd'hui plus ou moins spécifiques à un processeur cible. Un compilateur reciblable a pour ambition de générer du code pour une famille d'architectures. Notons qu'en général plus un compilateur est reciblable pour une large famille d'architectures, plus il est difficile d'obtenir un code efficace pour un processeur spécifique. Il est important de rechercher un bon compromis. Une telle approche facilite le passage d'un processeur à un autre, ce qui est capital compte tenu de la durée de vie réduite des processeurs DSP. L'aspect reciblable peut-être vu de différentes manières.

- Dans certains cas un nouveau compilateur est obtenu en réécrivant une partie plus ou moins grande du logiciel. Ces compilateurs sont dits « portables » et peuvent demander plusieurs mois de travail.
- Un nouveau compilateur peut-être obtenu aussi par compilation où seule la description du processeur cible change. Dans ce cas le passage d'un compilateur à un autre revient à écrire une nouvelle description du processeur. On parle dans ce cas de « compilateur de compilateur ».
- Enfin un compilateur est dit « indépendant » si un nouveau compilateur peut-être obtenu sans même recompiler les sources.

Le compilateur GNU-CC est reciblable et appartient à la catégorie des « compilateurs de compilateur ». En principe seule la description du processeur change. Cependant le modèle utilisé pour décrire le processeur peut s'avérer insuffisant pour certaines architectures limitant ainsi la famille de DSP couverte par le compilateur. Il est aussi parfois nécessaire de modifier (en plus du fichier décrivant le processeur cible) ou d'ajouter certaines routines aux sources du compilateur. De nombreuses recherches ont été entreprises ces dernières années dans le domaine des compilateurs DSP reciblables [Mar95a] [Mar95b] [Bau97] [Lan95] [Lie95] [Leu98a] [Mes99b] donnant lieu à la mise au point de plusieurs environnements de compilation flexibles permettant de générer un code pour différents processeurs DSP. Un effort particulier a été effectué sur les langages de description d'un processeur ou d'une classe de processeur. Citons par exemple les langages LISA (Language Instruction Set Architecture) [Pee99] ou ISDL (Instruction Set Description Language) [Had97] développés dans le souci de décrire une classe d'architecture.

2.4.2. Les différentes phases d'un générateur de code

2.4.2.1. Définition

« Un compilateur (Figure 21) est un programme qui lit un programme écrit dans un langage (langage source) et le traduit dans un programme équivalent dans un autre langage (le langage cible) » [Aho86].

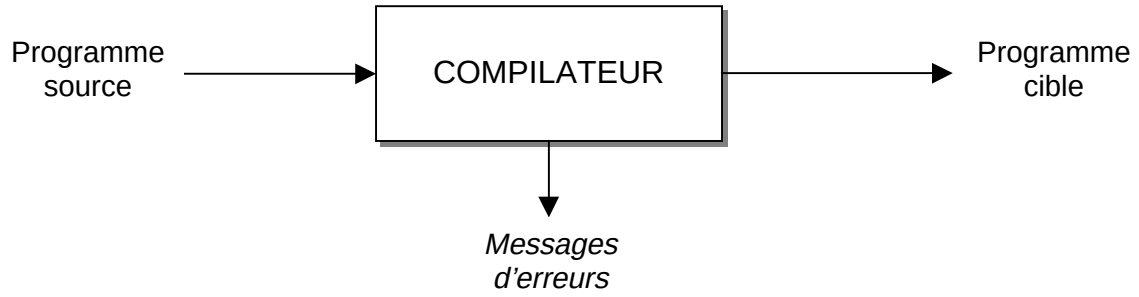


Figure 21 : Le compilateur

Notons qu'une partie importante de ce processus de traduction est de signaler à l'utilisateur d'éventuelles erreurs dans le programme source. Dans le cadre de cette étude nous nous intéressons aux compilateurs dont le langage source est le C et le langage cible est l'assembleur (dépendant du processeur cible).

2.4.2.2. Les différentes phases d'un compilateur

Un compilateur travaille en plusieurs phases, chacune d'elles transformant le programme source d'une représentation en une autre. Une décomposition typique d'un compilateur est montrée sur la Figure 22. En pratique certaines phases peuvent être regroupées, constituant la phase initiale (*front-end*) et la phase finale (*back-end*) du compilateur. Le ***front-end*** représente l'ensemble des phases dépendantes du langage source mais complètement indépendantes du processeur cible. Le *front-end* inclut ainsi les phases d'analyses lexicales et syntaxiques, la création de la table des symboles, l'analyse sémantique et la génération du code intermédiaire. Notons que bon nombre d'optimisations sont effectuées par le *front-end*. Le ***back-end*** inclut pour sa part les parties du compilateur qui dépendent du processeur cible. Généralement, ces phases dépendent du langage intermédiaire mais pas du langage source. Le *back-end* inclut les phases d'optimisation et de génération de code. La gestion des erreurs et la gestion de la table des symboles interviennent au niveau de toutes les phases du compilateur.

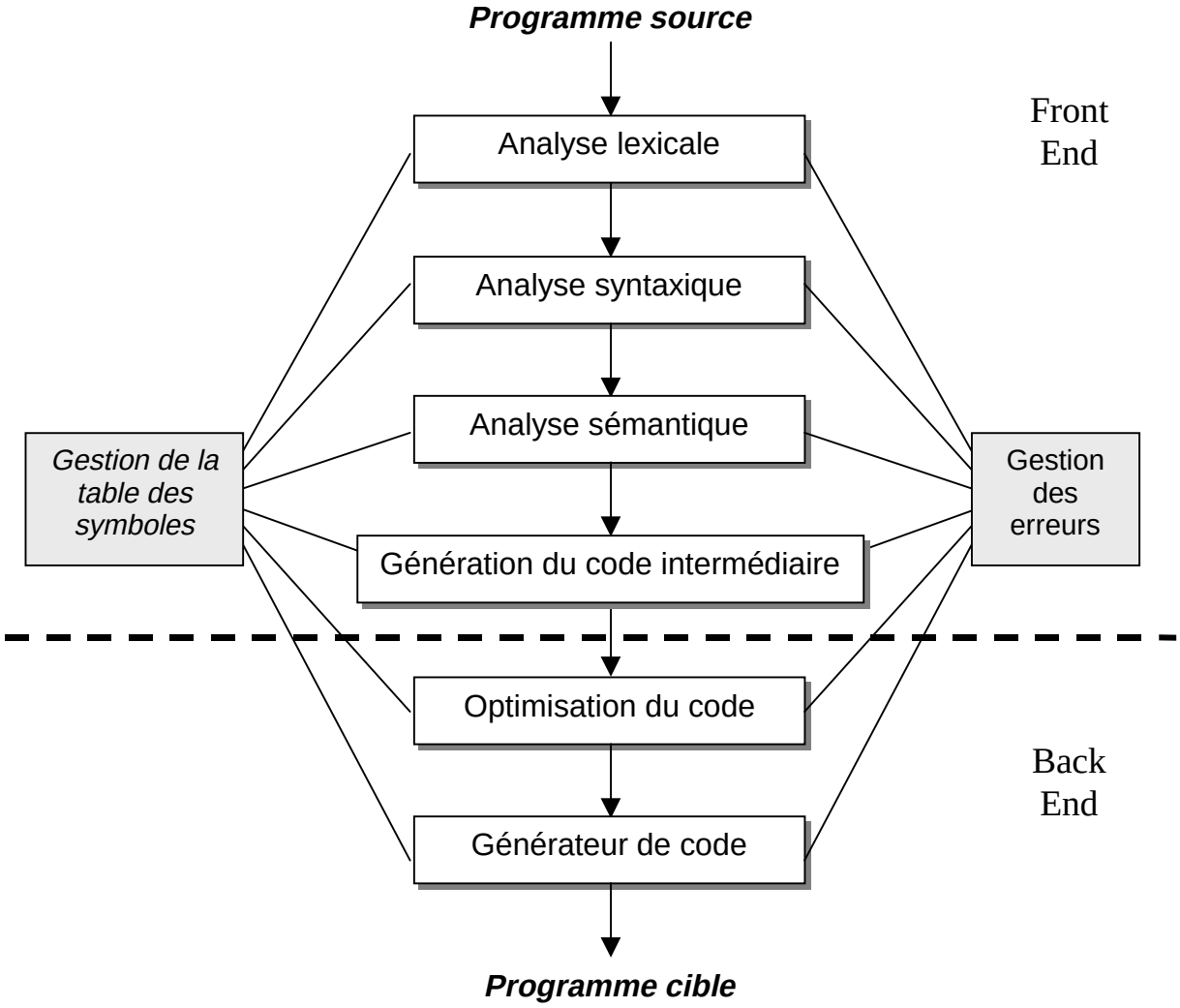


Figure 22 : Les différentes phases d'un compilateur

Pour la plupart des compilateurs actuels les phases d'optimisation et de génération de code sont une séquence de transformations fixe. Or, en général, une succession de transformations qui convient parfaitement à un processeur est inutilisable pour d'autres types d'architectures. En pratique il est néanmoins possible de modifier les transformations existantes. C'est le cas avec le GNU-CC [Sta94] pour lequel des optimisations spécifiques ont été développées pour le OakDSPCore et le PalmDSPCore par exemple. Dans la suite de ce rapport nous étudierons plus particulièrement les approches de ce type. Il existe cependant d'autres approches basées par exemple sur une bibliothèque de modules de production et d'optimisation de code. Ces méthodes sont particulièrement utilisées pour la conception de processeurs spécialisés (ASIP) [Yan98] ou l'exploration architecturale [Mes99a]. Citons l'outil SUIF [Wil94] qui intègre de multiples analyses du code source et la production d'une représentation intermédiaire ou la bibliothèque SPAM [Ara98] consacrée à l'optimisation de code pour des processeurs DSP.

L'indépendance du langage intermédiaire par rapport au processeur cible permet de construire différents compilateurs pour un même langage source. En d'autres termes les optimisations restent généralement identiques quelque soit la machine cible. Si le « *back-end* » a été conçu avec soin, seules quelques modifications et/ou rajouts sont nécessaires (par exemple réécrire une nouvelle description du processeur cible). De même, il est possible de compiler différents langages sources en une représentation intermédiaire commune, et d'utiliser un *back-end* unique pour obtenir différents compilateurs pour une seule machine. L'aspect multi-cible s'avère néanmoins difficilement réalisable dans le cas général compte tenu de certaines subtilités des langages sources et des langages cibles. Cependant une telle approche peut très bien être mise en œuvre pour un nombre réduit de machines et de langages sources.

Nous décrivons brièvement dans ce qui suit les différentes phases classiques de la Figure 22. Cette synthèse se base sur l'ouvrage de référence [Aho86].

2.4.2.3. L'analyse lexicale ou lexicographique

L'analyseur lexical vérifie la concordance lexicale entre le texte source et les contraintes imposées par la définition du langage. Le texte source est constitué d'unités lexicales de trois types:

- les mots clés du langage,
- les séparateurs (ponctuation, opérateurs arithmétiques/logiques, espace, caractères spéciaux, etc.),
- et des identificateurs ou lexèmes (identificateurs de variables, label).

L'analyseur lexical prépare aussi les analyses suivantes en générant un pseudo-code et en créant une table des symboles évolutive pour les lexèmes, mais fixe pour les mots-clés et les séparateurs.

2.4.2.4. L'analyse syntaxique

L'analyseur syntaxique vérifie la conformité syntaxique du texte source en s'appuyant sur un ensemble de règles de grammaire définies par l'utilisateur pour ce langage. La vérification s'applique sur des expressions régulières résultant de la concaténation de pseudo-codes issus de l'analyse lexicographique. L'analyseur syntaxique produit un arbre syntaxique conforme au texte source.

2.4.2.5. L'analyse sémantique

L’analyseur sémantique utilise l'arbre syntaxique issu de l’analyse syntaxique et vérifie la compatibilité sémantique résultant de paramètres complémentaires associés à la table des symboles. De plus, l’analyseur sémantique peut effectuer certaines corrections. Par exemple l'expression

x = 63;

est correcte syntaxiquement mais sémantiquement incorrecte si x est du type flottant. Dans ce cas l'analyseur sémantique peut éventuellement corriger l'expression, ce qui donne

x = 63.0;

Remarque : Une erreur sémantique est indépendante de la grammaire.

2.4.2.6. La génération de code intermédiaire

Cette phase consiste à générer une représentation intermédiaire explicite du programme source, généralement sous la forme de graphe flot de données (DFG). Le langage intermédiaire peut-être vu comme un programme pour une machine abstraite. La représentation intermédiaire doit être facilement produite et facile à traduire dans le langage du processeur cible. La plus grande partie du travail du compilateur est effectuée sur la représentation intermédiaire (phase d'optimisation et de génération de code). Elle peut avoir de nombreuses formes comme le code à trois adresses présenté dans [Aho86], ou les listes utilisées par le niveau RTL du GNU-CC [Sta94]. Notons que la représentation intermédiaire doit également prendre en compte le flot de contrôle du programme et les appels de fonctions.

Remarque : La génération d'un code intermédiaire n'est pas indispensable.

2.4.2.7. L'optimisation du code

La phase d'optimisation du code tente d'améliorer le code intermédiaire afin d'en faire un code plus rapide lors de son exécution. Certaines optimisations sont triviales et d'autres beaucoup plus complexes. Cette phase est constituée d'un ensemble d'optimisations qui transforme le programme afin d'améliorer le code cible sans prendre en considération les propriétés de la machine cible. En principe, les optimisations les plus fructueuses sont celles effectuées sur les parties de code les plus fréquemment exécutées. Malheureusement, un compilateur n'a pas ce type d'information et doit faire son possible pour localiser les parties critiques de l'application. En pratique le compilateur met l'accent sur les boucles imbriquées. Une technique pour déterminer quelles transformations sont efficaces est de collecter des informations statistiques sur le programme à l'aide d'échantillons représentatifs du programme source et ainsi d'évaluer l'intérêt des optimisations [Sol96]. **L'analyse du flot de contrôle** (control-flow analysis) du programme permet d'identifier les boucles. **L'analyse flot de données** (data-flow analysis) est le processus collectant des informations sur la manière dont les variables sont utilisées dans un programme. Ces informations sont également utilisées pour l'optimisation du programme.

Les transformations fournies par un compilateur optimisé doivent préserver le comportement du programme : l'optimisation ne doit pas changer les valeurs des sorties du programme ou causer une erreur pour un même jeu de données en entrée. Aussi une transformation doit permettre une accélération significative de l’exécution du programme même si occasionnellement une « optimisation » peut légèrement la ralentir.

La phase d'optimisation du code s'organise comme illustré sur la Figure 23. Les principales optimisations de cette phase sont présentées brièvement dans la suite. Notons qu'une transformation est dite « locale » si elle est effectuée sur des instructions appartenant a un même bloc de base.

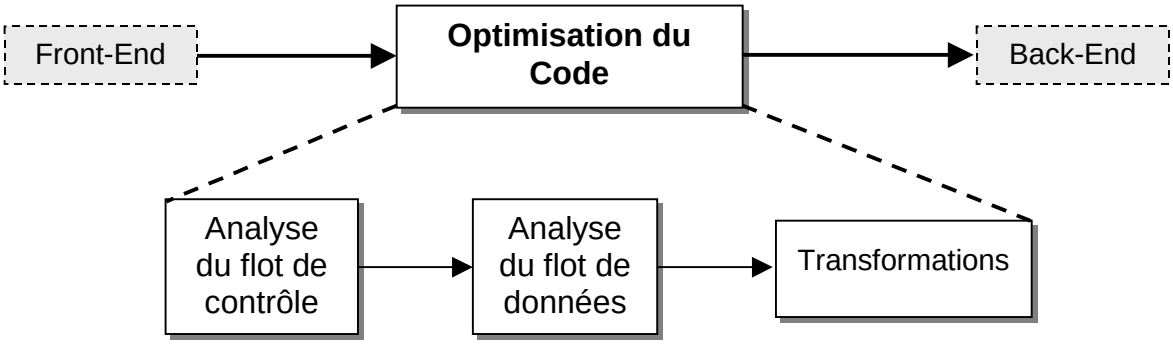


Figure 23 : Phase d'optimisation du code

Un bloc de base est un segment de programme dont le seul point d'entrée est la première instruction et le seul point de sortie est la dernière instruction [Aho86].

Dans le cas contraire la transformation est dite « globale ». Généralement les transformations locales sont effectuées en premier. Il est important de noter que ces transformations ne peuvent le plus souvent être effectuées directement au niveau du langage source car elles n'apparaissent qu'à la suite de transformations précédentes ou reposent sur un niveau de détail plus fin que celui du langage source. Voici les principales transformations qui ont lieu sur la représentation intermédiaire.

- Elimination de sous expressions communes : cette transformation évite de recalculer inutilement des expressions identiques à différents endroits du programme.
- Elimination des variables intermédiaires inutiles.
- Elimination de code mort : suppression des parties de code dont les résultats ne sont jamais utilisés ailleurs dans le programme.
- Propagation des constantes : remplacement d'une expression dont la valeur est une constante.
- Optimisation des boucles : l'optimisation des boucles et plus spécifiquement des boucles imbriquées est très importante puisqu'en général les programmes y passent la plupart de leur temps. Ainsi le temps d'exécution d'un programme peut-être amélioré si le nombre d'instructions dans la boucle est réduit, cela même si le nombre d'instructions à l'extérieur de la boucle augmente. Les trois principales techniques utilisées pour l'optimisation de boucles sont :
  - le déplacement de code à l'extérieur de la boucle,
  - l'élimination de variable d'induction : suppression d'une expression ou remplacement par une autre moins coûteuse en temps de calcul,
  - le remplacement d'une opération par une autre moins coûteuse (exemple: remplacer une multiplication par une addition).

2.4.2.8. La génération du code

Cette phase a pour objectif de générer à partir d'un code intermédiaire un code pour le langage cible. Le générateur de code prend en entrée une représentation intermédiaire du programme source issue du « *front-end* » ayant subi des optimisations indépendantes du

processeur cible. Durant cette phase, des optimisations dépendantes de l'architecture cible sont effectuées par le compilateur. Le générateur de code doit produire un code correct et de qualité, c'est-à-dire faisant un usage aussi efficace que possible des ressources du processeur cible. Mathématiquement, le problème de la génération de code est indécidable. En pratique on utilise donc des heuristiques et généralement une grande partie du temps de compilation total est passée dans cette phase. La génération de code consiste en un certain nombre de tâches qui sont principalement la sélection du code, l'allocation de registre, l'ordonnancement, la compaction du code et les optimisations « *peephole* ».

- **La sélection du code** est la tâche principale de tout compilateur. Elle consiste à rechercher dans un graphe flot de données représentant le programme source (Figure 24), des instructions dites partielles supportées par le jeu d'instructions du processeur. Pour cela chaque instruction machine est représentée par un sous-graphe (ou motif) décrivant son comportement. On parle de techniques de reconnaissance de motifs (*pattern matching*) et de couverture de graphe [Aho86][Goo97][Mar95c]. La Figure 25 décrit cinq motifs d'instructions différents. On suppose pour ces motifs que les nœuds + ou * ont pour arguments des registres et pour destinations des registres en l'absence de nœuds prédécesseurs « ref » ou « # ».

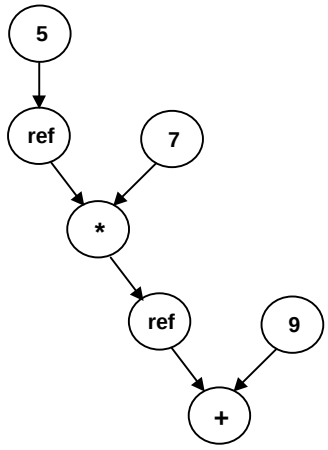


Figure 24 : Graphe flot de données

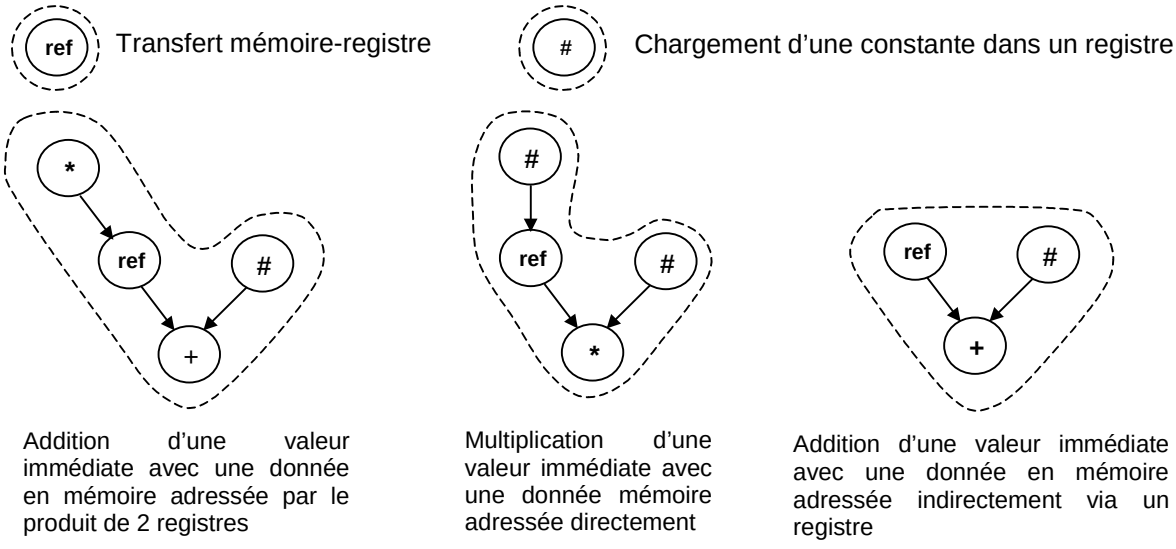


Figure 25 : Motifs d'instructions

La Figure 26 montre clairement qu'il existe deux alternatives pour couvrir le graphe de la Figure 24 avec les sous graphes d'instructions de la Figure 25. Plus le jeu d'instructions est riche (en mode d'adressage plus particulièrement) plus il existe d'alternatives pour sélectionner une instruction machine. Généralement, le critère de choix est le nombre minimum de motifs permettant de couvrir le graphe. En fait, le problème de la couverture optimale d'un graphe est un problème NP-complet. Les approches sont donc généralement basées sur des heuristiques décomposant le graphe en arbres [Got96a][Got96b], ou se servant de caractéristiques architecturales spécifiques permettant une décomposition optimale du graphe. Le modèle utilisé pour décrire le processeur est par conséquent d'une importance capitale. La nature du jeu d'instructions de la machine cible détermine le niveau de difficulté de la tâche de sélection du code. Un jeu d'instructions complet et uniforme facilite le travail du compilateur.

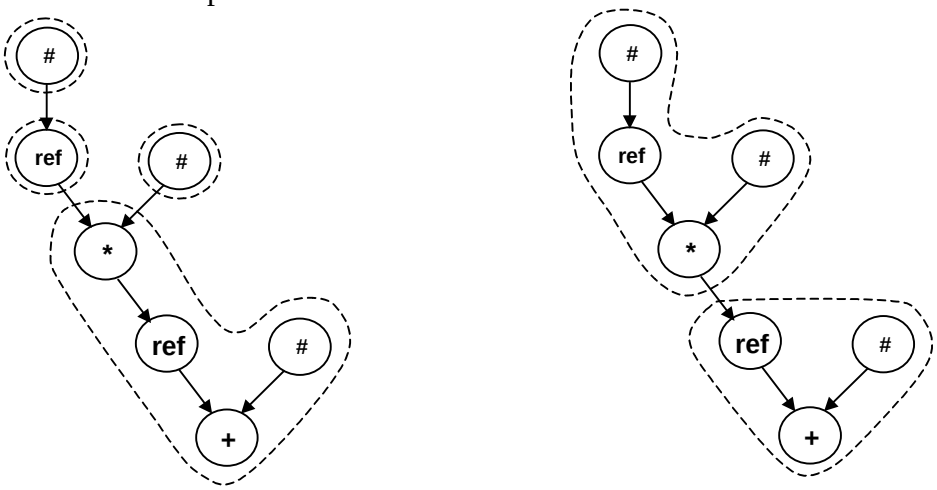


Figure 26 : Deux couvertures possibles du DFG par les motifs d'instructions

- **L'allocation de registres** est la tâche qui fait correspondre aux variables du programme (ou résultats temporaires) des registres de la machine cible. Comme les instructions impliquant des opérandes en registre ont un code plus compact et plus rapide que celles invoquant directement des opérandes en mémoire, une utilisation efficace des registres est très importante pour obtenir un bon code. L'allocation de registres détermine si une variable doit résider dans un registre. Si tel est le cas, l'assignation de registres designe dans quel registre la variable doit être rangée. Trouver une assignation optimale des variables dans des registres est un problème NP-complet, qu'il est encore plus difficile d'aborder lorsque la machine possède des conventions particulières sur l'utilisation de certains registres (hétérogénéité). L'allocation *locale* de registres alloue des registres aux variables utilisées uniquement à l'intérieur d'un bloc de base (i.e. dont la durée de vie n'excède pas la durée de vie du bloc de base). Or, à la fin de chaque bloc de base les variables vivantes sont sauvegardées en mémoire. Afin d'éviter des sauvegardes en mémoire inutiles, l'allocation *globale* de registres assigne aux variables les plus utilisées des registres fixes à travers plusieurs blocs de base. Cette technique est particulièrement payante pour les variables utilisées dans les boucles.

- **L'ordonnancement** est la tâche qui établit un ordre partiel parmi les opérations machines sélectionnées lors de la sélection du code, tout en maintenant la sémantique du programme. L'ordre dans lequel les calculs sont effectués peut influer sur l'efficacité du code généré et dans le cas d'architecture parallèle, l'ordonnanceur doit permettre autant de parallélisme que possible. Cependant, choisir le meilleur ordre est aussi un problème NP-complet. Il est important de noter que l'ordonnanceur doit avoir accès à une

description détaillée du jeu d'instructions du processeur, en particulier des conflits résultant des contraintes d'encodage ou structurelles, c'est-à-dire les ressources partagées par plusieurs instructions.

- **La compaction** du code est la tâche qui assigne aux opérations machines partielles des instructions machines. Pour beaucoup de machines cette tâche n'est pas nécessaire ce qui fait que la distinction entre les tâches d'ordonnancement et de compaction n'est pas faite. L'instruction MAC par exemple, commune à tout DSP, regroupe plusieurs opérations machines partielles.

- **L'optimisation *peephole*** est une méthode qui essaie d'améliorer les performances du programme cible en examinant une séquence d'instructions cibles (appelée *peephole*) et en les remplaçant par une séquence plus courte ou plus rapide chaque fois que possible. Notons que ces techniques peuvent s'utiliser directement sur la représentation intermédiaire afin d'améliorer cette dernière. Les optimisations *peephole* requièrent le plus souvent plusieurs passes successives sur le code cible pour obtenir les meilleurs résultats. Les transformations *peephole* typiques sont :
  - l'élimination d'instruction redondante,
  - la suppression d'instruction inaccessible,
  - l'optimisation du flot de contrôle,
  - les simplifications algébriques (e.g. x = x + 0),
  - l'utilisation d'instructions spécifiques du processeur.

2.4.3. La description du processeur cible

Les différentes phases de la génération de code ont besoin d'une description du processeur cible. Le modèle utilisé doit contenir toutes les informations nécessaires au compilateur afin de générer le code le plus compact et le plus rapide possible. Nous verrons les différentes approches pour modéliser un processeur dans le paragraphe 6.2.1.2.

2.5. Les problèmes spécifiques de génération de code pour processeurs DSP

Les principes et techniques de compilation sont étudiés depuis les années 50. Ces travaux ciblaient cependant les microprocesseurs à usage général de type CISC ou RISC [Aho86] ou encore des architectures VLIW [Ell85] et superscalaires. Pour ces processeurs, il existe des compilateurs très efficaces, c'est-à-dire produisant un code assembleur rapide et compact (parfois meilleur qu'un code écrit a la main). Il est en effet montré qu'il est possible pour des processeurs possédant une structure de registre homogène d'obtenir un code vertical (non parallélisé) optimal [Aho76] en un temps polynomial. Dans beaucoup de cas les compilateurs DSP commerciaux ont été développés à partir du GCC (GNU C Compiler) [Sta94], un compilateur reciblable distribué gratuitement par la Free Software Foundation. Les exemples de DSP pour lesquels il existe un compilateur C basé sur le GCC ne manquent pas : Analog Devices 2101, At&T 1610, Motorola 56001, ST Microelectronics D950 et bien sûr PineDSPCore et OakDSPCore de DSP Group. Mais les techniques de compilation incluent dans le GCC ont été développées pour des processeurs à usage général de type RISC. Les DSP nécessitent d'adapter ou de développer de nouvelles techniques de compilation spécifiques à leur modèle d'architecture [Mar95b][Leu98b][Goo97]. De plus, l'hétérogénéité caractéristique du jeu de registres de ces processeurs provoque une dépendance mutuelle des phases de génération de code appelée « ***phase coupling*** »

[Ara95b]. Une coopération intelligente des différentes tâches de génération de code est indispensable au succès d'un compilateur pour DSP. C'est au concepteur, en fonction de l'architecture DSP considérée, de décider du couplage[9] des différentes phases.

Remarque : Les problèmes de compilation pour les processeurs ASIP sont à notre sens les mêmes que ceux des DSP programmables. En effet, dans les deux cas l'architecture cible doit être en mesure d'exécuter efficacement les algorithmes de l'application cible.

2.5.1. Techniques de sélection de code et d'allocation de registres pour DSP

Dans le cas des RISC, les deux phases sont indépendantes et donc effectuées séparément. Pour un DSP, ces deux phases sont indissociables compte tenu de l'hétérogénéité du jeu de registres [Lee88]. Les techniques présentées dans ce paragraphe se restreignent à la phase de sélection du code avec une allocation locale de registres.

Des compilateurs récents pour DSP ont appliqué des méthodes de programmation dynamique pour résoudre les problèmes liés à la sélection de code et à l'allocation de registres, dans le cas des processeurs avec un jeu de registres hétérogène [Goo97]. Citons les outils *TWIG* [Aho89], *IBURG* [Fra92], *CODESYN* [Pau95] ou *OLIVE* [Ara95a] utilisant cette approche pour la sélection de code automatique. Malheureusement, ces techniques souffrent d'un faible couplage avec la phase d'allocation de registres. Il est parfois supposé durant cette phase que chaque classe de registres possède un nombre infini de registres (pour des raisons de simplification), ce qui repousse le problème du nombre restreint de registres au niveau de la phase d'allocation de registres. De même, ces techniques ne considèrent pas des optimisations comme la sauvegarde temporaire de données en mémoire (*spilling*). Dans [Wes95] une méthode combinant l'allocation de registres et la sélection du code pour une structure de registres hétérogène est présentée. Le *spilling* ainsi que le nombre restreint de registres est pris en compte. Cependant la méthode ne garantit pas un résultat optimal.

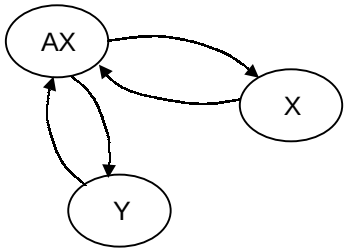


Figure 27 : Présence de cycle dans le RTG du OakDSPCore

Dans le cadre du projet SPAM, Malik et Araujo proposent un algorithme optimal utilisant la programmation dynamique pour effectuer la sélection du code, l'allocation de registres et l'ordonnancement des instructions en un temps polynomial pour une classe de DSP [Ara98]. Une première phase effectue la sélection du code et l'allocation de registres simultanément en utilisant une variante de l'algorithme de Aho et Johnson [Aho76]. La seconde phase réalise l'ordonnancement des instructions de telle sorte qu'aucun *spilling* n'est effectué. L'algorithme proposé est basé sur un graphe de transfert de registre (RTG) qui est une représentation structurelle du chemin de données du processeur annoté par des informations issues du modèle de l'architecture au niveau instruction (ISA). Chaque nœud

[9] Le terme « couplage » est également utilisé dans la littérature pour décrire le travail de coopération intelligente des différentes tâches de génération de code.

du graphe représente un emplacement du chemin de données de l'architecture ou une donnée peut y être stockée. Chaque arc du RTG reliant un nœud r_i à un nœud r_j est annoté par la ou les instructions de l'ISA qui prend un opérande de l'emplacement r_i et range le résultat dans l'emplacement r_j. Les nœuds du graphe peuvent représenter une file de registres ou un simple registre. La classe de DSP pour lequel un code optimal est garanti doit avoir un RTG acyclique. Malheureusement les DSP actuels ont généralement un RTG cyclique, i.e. comportant des cycles dans le chemin de données.

La Figure 27 montre la présence de cycles dans le RTG du OakDSPCore puisqu'il est possible de charger les registres X ou Y depuis les accumulateurs A0 ou A1 (AX sur la figure), et inversement. Le OakDSPCore offre ce mode de transfert afin d'accélérer les traitements. Dans le cas du TMS320C25 cette opération ne peut s'effectuer sans passer au préalable par la mémoire. Quand un RTG est cyclique, comme dans le cas du OakDSPCore ou du Motorola 56001, on ne peut pas garantir que toutes les opérations pourront s'ordonnancer sans faire de sauvegardes inutiles en mémoire (*spilling*). Par contre, si une suite d'opérations n'utilise pas de cycle du RTG il est toujours possible de garantir un ordonnancement optimal pour cette suite particulière. Montrer que dans un programme beaucoup d'expressions n'utilisent pas de cycle du RTG reste un problème ouvert.

La sélection du code peut également être vue comme un problème de *pattern matching* sur un graphe acyclique orienté (DAG) et non plus sur un arbre [Lia96]. Une alternative à la sélection de code basée sur une énumération exhaustive de tous les motifs est l'utilisation d'algorithme d'empaquetage (*Bundling*) qui construisent les motifs nécessaires « à la volée ». Ces algorithmes ont été récemment utilisés par des compilateurs pour des processeurs embarqués comme par exemple *Chess* [Lan95]. Un des avantages de cette approche est que ces algorithmes supportent des représentations intermédiaires et des instructions partielles représentées sous la forme de graphe plutôt que d'arbre ce qui est utile dans le cas des DSP. Un inconvénient est l'augmentation de la complexité algorithmique de cette approche par rapport aux méthodes de programmation dynamique.

Citons enfin les méthodes basées sur un ensemble de règles qui guident les transformations à effectuer sur la représentation du programme source à chaque phase de la compilation (*Rule-Driven Code Selection*). Utilisée dans le compilateur FlexCC [Lie95], cette approche suppose la définition d'une machine virtuelle et d'un ensemble de règles qui font correspondre aux motifs des opérations, les instructions de la machine virtuelle. Notons que cette dernière ne supporte aucun parallélisme au niveau instruction, ce qui repousse le problème du parallélisme sur la phase de compaction du code et diminue son intérêt pour les nouvelles cibles DSP. Si cette méthode est rapide, la qualité du code est de plus directement dépendante de l'habilité du concepteur dans l'écriture des règles de transformations et nécessite une réécriture préalable du code source à un niveau proche du jeu d'instructions du processeur.

2.5.2. L'allocation globale de registres

Dans les méthodes vues précédemment, la phase de sélection du code effectue aussi une allocation locale de registres. Nous décrivons à présent des techniques réalisant une allocation globale de registres. Ces approches sont également capables de prendre des décisions sur la sélection du code ce qui illustre bien le couplage entre les différentes phases de génération de code. Une technique fréquemment utilisée pour formuler le problème de l'allocation de registres est le coloriage d'un graphe modélisant la durée de vie des variables [Cha82]. L'allocation de registres se ramène alors à trouver au plus N couleurs (dans le cas

ou le processeur possède un nombre restreint N de registres) pour colorier les sommets du graphe. Si ce n'est pas possible cela signifie que le nombre de registres du processeur est insuffisant et qu'une sauvegarde temporaire de données en mémoire est nécessaire (*spilling*). Malheureusement, ces techniques se basent sur un ensemble d'hypothèses (jeu de registres homogène, phase de sélection de code séparée ou encore que toutes les durées de vie des variables sont connues à l'avance) incompatibles pour des processeurs DSP.

Afin de prendre en compte la structure hétérogène du jeu de registres des DSP, des techniques spécifiques de routage de données ont été développées. Le routage de données représente le transfert de données entre des unités fonctionnelles par le biais de registres intermédiaires. Or sélectionner le chemin le plus approprié est un problème non trivial. Comme il peut s'avérer nécessaire d'insérer des chemins indirects et donc des instructions supplémentaires, il existe aussi un couplage des phases d'allocation de registres et d'ordonnancement. Dans le compilateur CHESS [Lan95] par exemple, le couplage de ces deux phases se fait à l'aide d'estimateurs probabilistes d'ordonnancement pendant le processus d'allocation de registres. Ce compilateur utilise une technique de routage des données globale contrairement aux techniques locales (gloutonnes) utilisées dans le compilateur Bulldog [Ell85] par exemple qui cherche ainsi à limiter d'éventuels problèmes d'explosion combinatoire.

2.5.3. L'allocation mémoire et la génération des adresses

L'allocation mémoire consiste à déterminer les emplacements mémoires (les adresses) pour les données sélectionnées lors d'un transfert avec la mémoire durant la phase d'allocation de registres. Les DSP possèdent à cet effet des unités matérielles dédiées appelées **générateurs d'adresses (AGU)** autorisant en particulier la modification des adresses en parallèle avec des opérations du processeur (post/pré modification). Bien que ces unités existent depuis longtemps, les techniques de compilation ont ignoré le problème lié à l'utilisation efficace de ces unités, contribuant à la faible efficacité du code généré. Afin de tirer le meilleur profit de ce type d'adressage, il est le plus souvent avantageux d'allouer les données en mémoire à des adresses consécutives ou à des adresses suffisamment proches les unes des autres pour utiliser les mécanismes spécialisés de modifications d'adresses. Cette optimisation typique des processeurs DSP a été appliquée dans un premier temps sur le TMS320C2X [Bar92] supportant un mode d'adressage post/pré incrémenté ou décrémenté. L'auteur propose de trouver une solution avec un nombre maximum de post-modifications par la recherche d'un chemin hamiltonien dans un graphe. Les résultats pour un codeur de parole à base de LPC[10] sur le TMS320C30 montrent en effet jusqu'à 12% d'accélération pour certaines fonctions de l'application. Dans [Leu96] les auteurs ont généralisé la méthode précédente à plusieurs registres d'adresse. Par rapport à une assignation naïve des adresses c'est-à-dire non optimisée DSP, leur méthode permet de réduire de 80 à 90% le nombre d'instructions dédiées à la génération des adresses. Or, si l'on se réfère à [Lia96] ces instructions peuvent constituer 20% de la taille totale du code. Il est donc évident que de telles méthodes sont très importantes pour les systèmes embarqués.

Des travaux complémentaires ont été effectués par [Lie95] pour une génération efficace des adresses dans le cas d'accès à des tableaux. L'objectif de cette méthode est de transformer les références aux éléments d'un tableau par index (tab[i]) en des références par pointeurs (*ptr) avec des post-modifications appropriées et optimisées pour la spécification de l'unité de génération d'adresse fournie. En effet, l'utilisation de pointeurs permet

[10] Linear Predictive Coding

généralement de tirer un meilleur profit de l'unité de génération d'adresse. Grâce à des traces obtenues lors de l'exécution du code C sur le processeur hôte, il est possible de déterminer l'évolution des accès au tableau et d'analyser si une référence à une position du tableau est visitée linéairement à l'intérieur d'une boucle (référence stable). Si tel est le cas, un pointeur est utilisé avec éventuellement des post/pré modifications. Cette méthode a été testée avec un compilateur dédié pour un processeur VLIW pour diverses fonctions DSP. En moyenne les résultats montrent que la taille du code est réduite de 23% et le temps d'exécution de 39% en comparaison du code d'origine utilisant des accès indicés aux tableaux.

Si les techniques que nous venons d'évoquer permettent une accélération significative des performances, ces optimisations sont toutefois restreintes au mode simple de post modification (linéaire). Or, les DSP possèdent souvent des modes d'adressage spécifiques beaucoup plus complexes comme l'adressage modulo ou le *bit reversal*. De plus, il existe généralement des contraintes sur l'utilisation de ces modes d'adressage. Par exemple, l'adressage modulo du OakDSPCore [Oak96] possède des restrictions sur la taille maximale ou l'adresse de base du tableau. Dans le cas de la FFT, la gestion des adresses est particulièrement délicate puisque lors du calcul d'un papillon, l'incrément sur les adresses des données (les échantillons d'entrées comme les coefficients) dépend de l'étage considéré mais aussi du nombre de points de la transformée qui peut-être inconnu au moment de la compilation. De plus, afin d'optimiser le nombre de transferts avec la mémoire, il est dans certains cas indispensable d'allouer des données dans différents bancs mémoires. Or, il n'est pas forcément trivial pour un compilateur d'arranger efficacement les données dans différents bancs. En conclusion, la gestion des adresses est un facteur prépondérant de l'inefficacité des compilateurs DSP. Si des techniques adaptées aux AGU des DSP permettent d'améliorer l'utilisation de ces unités et donc les performances, elles sont encore dans beaucoup de cas incapables de générer un code optimisé. Par conséquent, il est généralement indispensable que l'utilisateur apporte des modifications au code source (utilisation de pointeurs, localisation des données dans les différents bancs mémoires, réorganisation des données en mémoire à des adresses consécutives, etc.). Malgré cela, nos expérimentations montrent que dans le cas d'algorithmes complexes comportant de nombreux accès mémoires, il est parfois très difficile pour le compilateur de tirer au maximum profit des modes d'adressage et donc d'éviter l'écriture du code en assembleur.

2.5.4. L'ordonnancement

Les architectures DSP récentes se caractérisent par plus de parallélisme au niveau instruction et de plus grande profondeur de pipeline, ce qui augmente l'importance de la phase d'ordonnancement. On distingue des algorithmes d'ordonnancements locaux et globaux. Un ordonnancement local comme l'algorithme d'ordonnancement par liste [Lan80], opère au niveau des instructions appartenant à un même bloc de base. Quand l'architecture cible a peu de parallélisme, un ordonnancement local produit généralement de bons résultats. Cependant, il se peut que le parallélisme décrit au niveau d'un bloc de base soit en deçà du parallélisme offert par l'architecture. Pour utiliser plus efficacement les ressources du processeur, un ordonnancement global modifie alors la description algorithmique de l'application sans en changer la sémantique. Ces transformations algorithmiques sont particulièrement efficaces en présence de branchement conditionnel ou dans le cas de formes itératives. Dans ce dernier cas, les techniques fréquemment utilisées sont le dépliement de boucle et le pipeline logiciel. Ces deux transformations permettent d'exprimer plus de parallélisme dans les boucles, ce qui les rendent indispensables pour les

DSP compte tenu du nombre important de boucles dans les applications de traitement du signal.

Afin d'illustrer ces deux techniques, nous prenons le cas de l'implémentation d'un filtre à réponse impulsionnelle finie (FIR) sur les cœurs de DSP OakDSPCore puis PalmDSPCore. La description algorithmique en langage C de ce filtre peut se résumer à la forme itérative décrite sur la Figure 28.a:

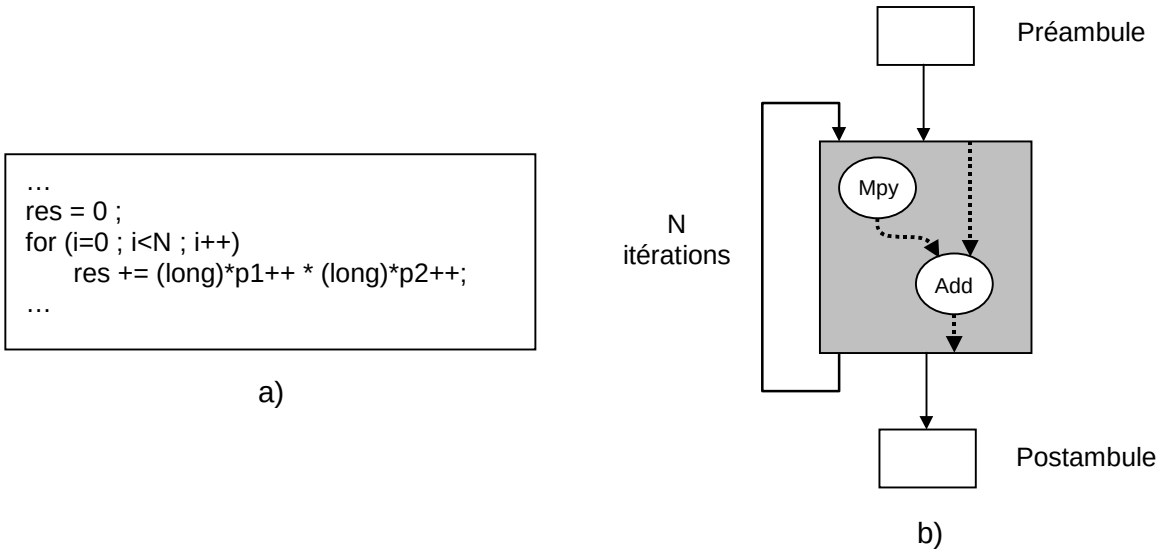


Figure 28 : Implémentation d'un FIR

Dans le cas du OakDSPCore, si aucune précaution n'est prise, on pourrait penser que la somme des produits s'effectue à l'intérieur d'un même bloc de base comme le montre la Figure 28.b. Or ceci n'est pas une implémentation optimale puisque le OakDSPCore possède un « faux » MAC (on montre que N+1 itérations sont nécessaires dans ce cas). Pour tirer au maximum profit de l'architecture, une transformation algorithmique de type pipeline logiciel est nécessaire afin de sortir la première multiplication (amorçage de la boucle) ainsi que la dernière addition de la boucle.

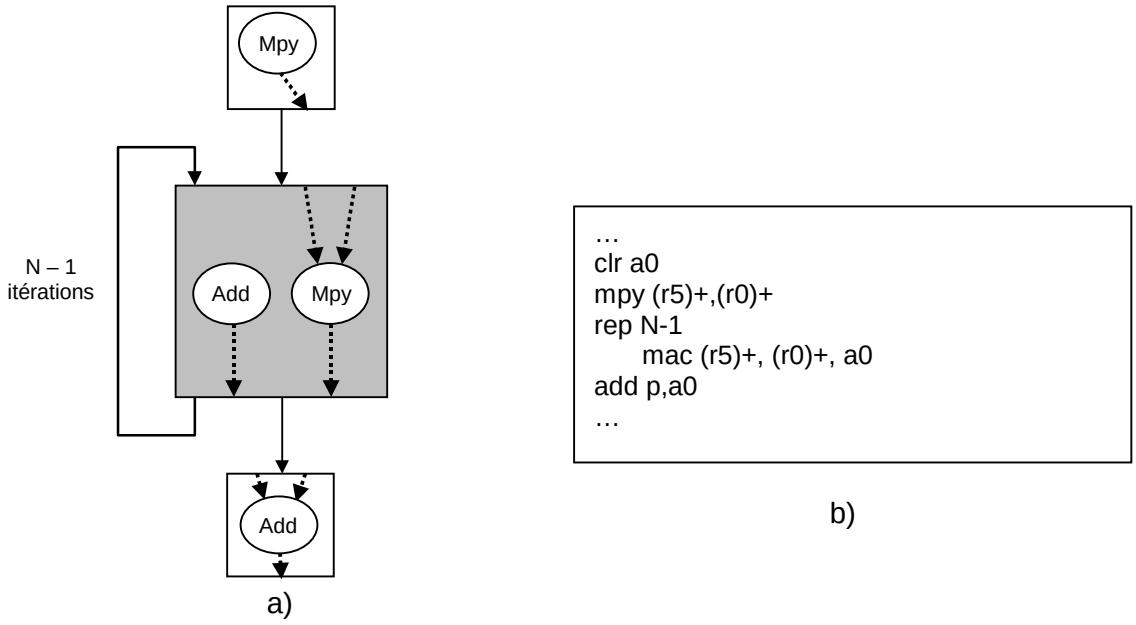


Figure 29 : Pipeline logiciel

C'est ce qu'illustre la Figure 29.a. **Le pipeline logiciel** tente donc de réorganiser le code (globalement si besoin) de manière à optimiser l'implémentation de l'algorithme sur l'architecture cible. La Figure 29.b représente le code assembleur généré par le compilateur OCC tenant compte de cette modification (i.e. le code est optimisé).

Dans le cas du PalmDSPCore une transformation algorithmique de type pipeline logiciel est également nécessaire. On constate de plus que la représentation de la Figure 28.a comporte moins de parallélisme que l'architecture cible le permet, puisque ce DSP est capable de faire deux MAC en parallèle à chaque cycle. La technique utilisée dans ce cas est le **dépliement de boucle** qui consiste à réduire le nombre d'itérations tout en augmentant le nombre d'opérations à l'intérieur du bloc de base de la boucle. La Figure 30.a illustre ces transformations alors que la Figure 30.b décrit le code assembleur optimisé pour le PalmDSPCore dans ce cas. On remarque bien que le nombre d'itérations est passé de N à (N/2).

Le nombre restreint de registres et l'hétérogénéité des DSP limitent cependant très fortement l'utilisation du pipeline logiciel et du dépliement de boucle. Dans le cas du OakDSPCore par exemple, si le dépliement de boucle est effectué par le OCC le résultat est catastrophique. En effet, le compilateur déplie aussi les adresses et alloue à chacune d'elles un registre, ce qui provoque des sauvegardes temporaires de données en mémoire (*spilling*) à l'intérieur de la boucle. Il est cependant possible d'utiliser cette technique efficacement lors d'un codage à la main.

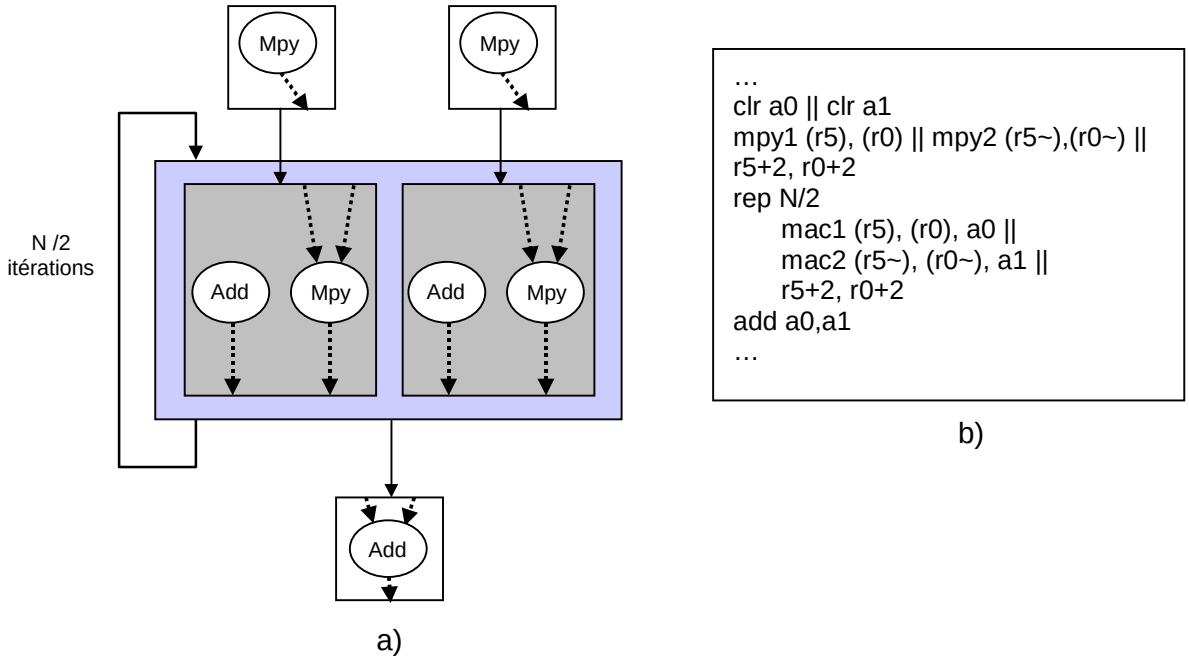


Figure 30 : Dépliement de boucle

Les techniques de pipeline logiciel peuvent être utilisées afin d'optimiser les formes itératives (*loop folding*) [Goo92] mais aussi les structures de branchements conditionnels (*modulo scheduling*) [Lam88]. Citons la technique du *trace scheduling*, utilisée dans le compilateur *Bulldog* pour processeur VLIW [Ell85], basée sur des probabilités d'exécution de branchements conditionnels. Grâce à ces informations, des séquences de blocs de base, ou *traces*, sont identifiées et ordonnancées comme s'il s'agissait d'un unique bloc de base. Ainsi, les opérations peuvent être déplacées à l'extérieur des blocs de base d'origine. Si la trace n'a qu'un seul point d'entrée et qu'un seul point de sortie, la méthode se simplifie

considérablement (on parle alors de *superblock scheduling*). D'autres approches consistent à déplacer des instructions entre blocs de base en s'appuyant sur un jeu de transformations sémantiquement invariantes basées par exemple sur des estimateurs probabilistes [Hor98]. Si ces techniques sont efficaces pour améliorer les performances en nombre de cycles, il est nécessaire d'ajouter un nombre parfois important d'instructions pour garder un code sémantiquement équivalent. Ceci fini par augmenter considérablement la taille du code, ce qui est inacceptable dans le cas d'applications embarquées.

2.6. Conclusion

De nombreuses recherches ont été entreprises ces dernières années pour optimiser la qualité du code généré par les compilateurs DSP. Certaines techniques donnent d'ailleurs d'excellents résultats [Ara98][Lie96][Mar95a][Lin94]. On peut cependant reprocher à ces tests d'être souvent peu représentatifs de la qualité réelle du compilateur. En effet, les tests sont généralement effectués sur des algorithmes de base de traitement du signal (FIR par exemple) et pas toujours sur des applications complètes. Or, nos expérimentations ont mis en évidence que pour des applications complètes les performances peuvent parfois être nettement moins bonnes. En effet, l'inefficacité du compilateur se diffuse à l'ensemble du programme et plus particulièrement dans les préambules et postambules des boucles, ce qui finit par détériorer les performances globales du programme.

De plus, on observe que l'amélioration des performances passe généralement par une réécriture du code C d'origine dans une forme se rapprochant d'un code assembleur. Dans le cas d'un code C « *out of the box* », le code généré a souvent un taux d'expansion très élevé (supérieur à cinq) par rapport à un code assembleur optimisé. Néanmoins, nous savons qu'il est possible d'améliorer significativement les performances en optimisant le code C pour le DSP (i.e. compilateur) considéré [Peg98a] [Peg99c] [Lie95] [Lev97]. Partant de ce constat, une approche de haut niveau est envisageable pour la classe de DSP qui nous concerne. Cependant, l'optimisation du code C sous-entend au préalable une étude fonction par fonction des performances du code assembleur généré (*profiling*). En effet, pour les applications de traitement du signal il est généralement admis que 80% du temps d'exécution représente moins de 20% de la taille totale du code. Le travail d'optimisation doit en conséquence être focalisé sur ces parties du programme. Sans l'aide d'outils adaptés, la localisation de ces parties critiques s'effectue en exécutant le code assembleur généré à l'aide d'un simulateur de niveau instruction fourni avec le DSP. Or, les temps de simulation correspondants sont très longs [Ziv96a]. De plus, dans le cas du OakDSPCore par exemple, il est nécessaire d'indiquer au *profiler* le début et la fin de chaque fonction si l'on souhaite connaître les performances pour chacune d'elles. On imagine aisément le travail que représente cette tâche dans le cas par exemple de la norme V22bis [Ste91], qui possède 280 fonctions. Afin de concentrer l'effort sur l'optimisation du code source, nous proposons d'aider le programmeur en lui fournissant des estimations logicielles sans aucune simulation au niveau assembleur. Ces estimations vont permettre de localiser avec un gain de temps considérable, les fonctions critiques de l'application mais aussi de guider le programmeur dans son travail d'optimisation.

Les méthodes d'estimation logicielle

La détermination du temps d'exécution d'une application sur un processeur peut s'effectuer simplement en exécutant le code sur ce processeur ou en utilisant un simulateur de niveau instruction. Malheureusement, dans le premier cas s'il s'agit d'un système embarqué il n'est pas aisé de mesurer les performances surtout lorsque le comportement du processeur dépend de son environnement. Dans ce cas on peut utiliser un simulateur mais ces derniers sont très lents et peuvent demander plusieurs jours de simulation. De plus, les résultats dépendent des données d'entrée (i.e. séquences de test). Les méthodes d'estimation logicielle ont pour principal objectif de réduire le temps nécessaire à l'élaboration de mesures de performance. Il s'agit pour cela d'éviter l'utilisation de simulateurs de niveau instruction et de déterminer les temps d'exécution à partir d'une description de haut niveau de l'application.

3.1. Les différents types de mesure de performance pour les systèmes embarqués

3.1.1. Performance dans le pire cas

Dans le cas des systèmes réagissant à des événements externes, les tâches associées doivent parfois s'effectuer à des instants précis. La validité de tels systèmes temps-réel dépend non seulement de l'exactitude des calculs, mais aussi des instants où sont fournis les résultats. Certains de ces systèmes ne peuvent tolérer de dépassement de temps limites quelles que soient les conditions (e.g. l'unité de contrôle ABS d'une voiture). Ainsi, pour ce type de système la mesure de performance appropriée est *une mesure du temps d'exécution dans le pire cas*[11].

3.1.2. Performance statistique

Dans certains systèmes, les contraintes de temps sont moins fortes. Il est en effet toléré de dépasser occasionnellement certaines limites de temps. C'est le cas par exemple des téléphones cellulaires. Pour ce type d'applications, *une mesure de performance statistique* qui garantit une haute probabilité de respecter les contraintes de temps suffit.

3.1.3. Performance moyenne

L'utilisation de *mesures de performances moyennes* est généralement nécessaire dans le cas ou le système ne possède aucune contrainte temps-réel. Un exemple typique d'un tel système est une imprimante pour laquelle la vitesse d'impression moyenne est souvent exprimée en page imprimée par minute.

3.1.4. Modèle de calcul du temps d'exécution

Le modèle couramment utilisé pour calculer le temps d'exécution d'une application est le suivant :

$$T_{exe} = \sum_{i=0}^{N} x_i c_i \qquad (1)$$

Trouver le temps d'exécution dans le pire cas (respectivement dans le meilleur cas) revient à maximiser (respectivement à minimiser) cette expression, ou N est le nombre de blocs de base du programme.

[11] La *mesure du meilleur cas* peut aussi être importante pour s'assurer que le système ne répond pas plus rapidement que prévu.

Les x_i représentent le nombre d'exécutions de chaque bloc de base B_i, alors que les c_i dénotent le temps d'exécution associé à chaque bloc de base B_i. Notons que seuls les c_i dépendent de l'architecture cible. Le modèle défini en (1) suppose cependant que l'exécution d'un bloc de base B_i prend toujours le même nombre de cycles. Dans le paragraphe 3.3.2 nous verrons que ce temps peut varier sensiblement suivant les propriétés architecturales du processeur cible (pipeline, mémoire cache, etc.).

Dans ce qui suit nous décrivons différentes approches permettant de déterminer le nombre d'exécutions (x_i) et le nombre de cycles (c_i) associés à chaque bloc de base. Le choix d'une méthode dépend du type d'estimation souhaité ainsi que du type d'application cible.

3.2. Le calcul des x_i

Le nombre d'exécutions x_i de chaque bloc de base du programme dépend du flot de contrôle du programme ainsi que des valeurs possibles des variables du programme. Déterminer les x_i revient donc à analyser les différents chemins possibles du programme lors de son exécution. Les techniques d'analyse de chemins se basent sur une décomposition du programme en blocs de base. Tout programme peut être partitionné en blocs de base disjoints. La structure d'un programme est représentée sous la forme d'un graphe de flots de contrôle (CFG[12]) ou chaque nœud est un bloc de base. Classiquement, on considère deux types de structure de contrôle dans un programme : les branchements et les boucles.

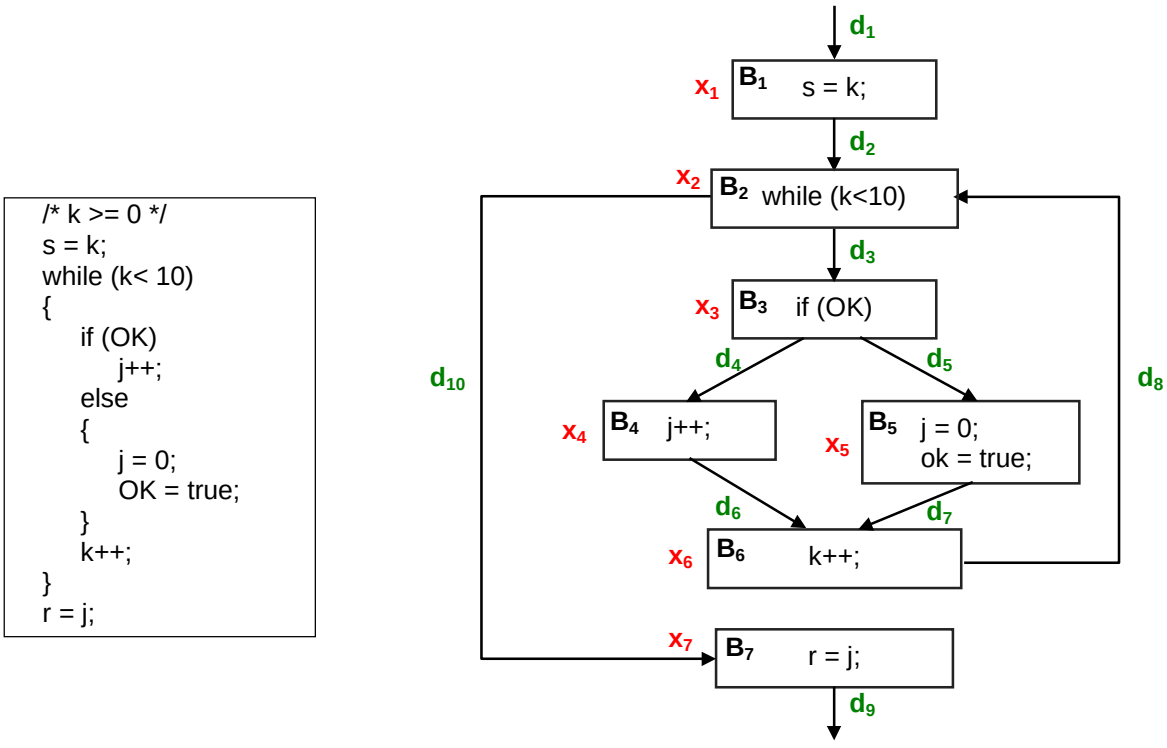


Figure 31 : Exemple d'un graphe de flots de contrôle

La Figure 31 montre un exemple de représentation sous forme de CFG d'un programme C comportant une instruction de branchement à l'intérieur d'une boucle. Notons que

[12] Control Flow Graph

certaines approches présentées dans la suite annotent chaque arc du graphe par une variable d_i représentant le nombre de fois que le programme passe par cet arc lors de son exécution.

On distingue trois types de méthodes permettant de déterminer la valeur des x_i d'un programme.

- **Les approches statiques** font références aux techniques d'analyse des chemins du programme indépendamment des valeurs des données d'entrée. Ces techniques, appelées aussi approches formelles, s'effectuent avant ou pendant la phase de compilation du programme.
- **Les approches dynamiques** font, quant à elles, références aux méthodes récupérant des informations par l'exécution du programme sur un ensemble de données d'entrée (séquences de test).
- **Les approches mixtes** combinant des méthodes formelles et dynamiques.

3.2.1. Les techniques d'analyse statique

La plupart des travaux d'analyse statique cherchent à déterminer les valeurs des x_i dans le pire ou le meilleur des cas[13]. Or, trouver le pire (ou le meilleur) des cas est un problème indécidable équivalent au problème de la terminaison d'un programme. Les structures de contrôle sont responsables de l'augmentation de la complexité du problème. Le nombre de chemins possibles d'un programme augmente en effet de manière exponentielle chaque fois qu'il existe une instruction de branchement à l'intérieur d'une boucle [Ern95]. Pour rendre ce problème décidable, le programme doit respecter certaines restrictions [Pus89]: toutes les boucles doivent avoir un nombre fini d'itérations, ne comporter aucune structure de données dynamiques (comme les pointeurs) ni d'appel récursif. Dans la suite nous nous intéressons au temps d'exécution dans le pire cas. Les performances dans le meilleur cas s'obtiennent généralement de façon analogue.

3.2.1.1. Sélection des cas extrêmes

Une approche directe dans l'analyse du pire cas, utilisée par Shaw [Sha89], consiste à choisir toujours le chemin le plus long lors d'un branchement ou d'une boucle. Par exemple, pour une instruction « *if* (condition) *then* C1 *else* C2 », il compare les temps d'exécution de C1 et C2 et prend toujours le plus grand pour le calcul du pire cas. Cette méthode conduit inévitablement à un pessimisme excessif comme le montre l'exemple de la Figure 32.

```
        if (ok)
S1          i = i*i + 1;    /* i != 0 */
        else
S2          i = 0;
        .
        .
        .
        if (i)
S3          j++;
        else
S4          j = j*j;
```

Figure 32 : Parties de code en exclusion mutuelle

[13] Correspondant respectivement aux chemins le plus long et le plus court du programme.

Les instructions S_1 et S_3 ainsi que S_2 et S_4 sont toujours exécutées ensemble. En utilisant la méthode décrite précédemment, S_1 et S_4 seront les instructions sélectionnées pour le calcul du pire cas. Or, en pratique ces deux instructions ne sont jamais exécutées ensemble (S_1 et S_4 sont en exclusion mutuelle). Cet exemple met en évidence la présence de chemins impossibles dans un programme.

Un chemin impossible est un chemin dans le CFG qui ne peut être exécuté quelles que soient les conditions sur les données en entrée.

Ce type de relation est fréquent dans les programmes. Il est donc important de détecter les chemins impossibles et d'en tenir compte si l'on souhaite obtenir des estimations précises de performance.

3.2.1.2. Extensions du langage source

Puschner et Koza [Pus89] ont étendu l'approche de Shaw pour améliorer le calcul dans le pire cas en laissant au programmeur le soin d'ajouter des informations sur les contraintes du CFG du programme. Cette approche part du constat que pour certaines parties du programme, il n'est pas possible de déterminer le nombre maximum d'exécutions sans l'aide du programmeur. Ce dernier dispose alors d'extensions du langage source lui permettant d'utiliser ses connaissances sur l'exécution de ses algorithmes (ce qu'il n'aurait pu faire avec les constructions classiques du langage source). Ceci permet de faire localement des choix non pessimistes.

Dans [Pus89] trois types d'extensions sont disponibles.

- Un **scope** définit une partie du programme.
- Un **marqueur** défini à l'intérieur d'un scope, permet de spécifier le nombre maximum de fois qu'une partie du programme ainsi « marquée » est susceptible de s'exécuter entre l'entrée et la sortie du scope. Les marqueurs sont utilisés principalement pour spécifier que le nombre d'exécutions d'un ou plusieurs chemins à l'intérieur d'une boucle est borné. La Figure 33 montre l'utilisation d'un scope et d'un marqueur pour le programme C légèrement modifié de la Figure 31. Si le programmeur sait que le nombre d'itérations est toujours inférieur à la valeur maximum (VAL_MAX), on comprend aisément le gain de précision apporté par ce type d'annotation. Ici le programmeur indique que le nombre d'itérations de la boucle « *while* » ne dépasse pas VAL_MAX/2.

```
/* k >= 0 */
s=k;
while (k< VAL_MAX) SCOPE MAX_COUNT(VAL_MAX/2)
{
    if (OK)
        j++;
    else
    {
        j=0;
        OK = true;
    }
    k++;
}
r = j;
```

Figure 33 : Exemple d'annotation d'un programme C

- Une **séquence de boucle** entoure un ensemble de boucles pour lequel le nombre d'itérations n'excède pas une certaine valeur lors de l'exécution. Cette extension est très utile dans le cas de boucles imbriquées et/ou mutuellement dépendantes. En effet, dans le cas de boucles imbriquées, le nombre d'itérations des boucles les plus internes n'est pas forcément égal au produit du nombre maximum d'itérations des boucles les plus externes. Un cas typique est l'algorithme de calcul de la FFT dont le nombre d'itérations pour les deux boucles les plus imbriquées est inversement proportionnel mais dont le produit est toujours égal à N/2, N étant le nombre de points de la transformée.

En utilisant ces nouvelles constructions, les auteurs montrent sur un exemple de traitement d'image une réduction supérieure à 10 du calcul du pire cas. Mais cette méthode ne garantit pas que toutes les relations entre différentes parties du programme sont exploitées et suffisantes. Ce travail peut d'ailleurs s'avérer extrêmement difficile dans le cas de structures de contrôle complexes et/ou mutuellement dépendantes.

3.2.1.3. Enumération explicite des chemins

Pour prendre en compte les relations entre différentes parties du programme, une approche consiste à énumérer l'ensemble des chemins du programme. Cette énumération ne peut être que partielle puisque le nombre de chemins d'un programme augmente exponentiellement avec la taille de l'application. Cependant, dans la recherche du temps d'exécution dans le cas pire, cette énumération doit être pessimiste, i.e. elle peut inclure des chemins même si ces derniers ne sont jamais parcourus. Les premières recherches dans cette direction ont été effectuées par Park et Shaw [Par89]. Ces derniers montrent que l'ensemble des chemins possibles du programme[14] peut être exprimé sous la forme d'expressions régulières. Par exemple, les équations suivantes décrivent les expressions régulières pour une structure « if-then-else » et une boucle « while » dont le nombre d'itérations est borné et égal à n.

if (*Cond)* **then** S_1 **else** S_2 : $Cond \,.\, (S_1 + S_2)$
while (*Cond*) **do** *S* : $Cond \,.\, (S \,.\, Cond)^n$

L'ensemble de ces expressions régulières appartiennent à un ensemble A_p. L'utilisateur peut également (comme dans la méthode précédente) fournir des informations sur les chemins du programme en utilisant un langage appelé IDL (**I**nstruction **D**escription **L**anguage). Il est ainsi possible de décrire le type de relations suivantes :

- deux instructions (ou groupe d'instructions) sont toujours exécutées ensemble,
- deux instructions (ou groupe d'instructions) ne sont jamais exécutées ensemble, i.e. elles sont en exclusion mutuelle,
- une instruction (ou groupe d'instructions) est exécutée un certain nombre de fois.

Par la suite, ces relations sont elles aussi traduites sous la forme d'expressions régulières dénommées I_p. L'intersection de ces expressions régulières ($A_p \cap I_p$) représente tous les chemins possibles du programme. En examinant l'ensemble de ces solutions il est alors possible de déterminer les chemins (i.e. les temps d'exécution) dans le pire et le meilleur des cas.

[14] D'un point de vue statique

Le principal inconvénient de cette approche est que l'intersection de A_p et I_p est une opération complexe et coûteuse. Il peut en effet s'avérer nécessaire d'examiner *explicitement* un nombre exponentiel de chemins potentiels. Ainsi, dans de nombreux cas, des simplifications sont indispensables et limitent la précision des résultats.

3.2.1.4. Une méthode basée sur les probabilités de branchement

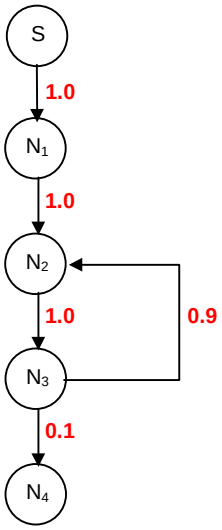
Dans [Gon93], la détermination des fréquences d'exécution (i.e. le nombre d'exécution) de chaque nœud du CFG du programme est effectué à partir de probabilités de branchements associées aux arcs du CFG. Ces probabilités peuvent être déterminées de différentes manières.

- **Probabilités égales** : dans le cas de branchement une probabilité égale est assignée à chaque arc successeur du nœud. Ainsi, si un nœud possède *n* arcs successeurs, on assigne à chacun d'eux une probabilité égale à *1/n*.
- **Probabilités pour les boucles** : quand le nombre d'itérations d'une boucle est connu, égal à n, l'arc sortant a une probabilité de *1/n* tandis que l'arc de retour a une probabilité de *(n-1)/n*.
- **Probabilités définies par l'utilisateur** : l'utilisateur peut spécifier des probabilités de branchement en annotant directement la spécification du programme.

Les deux premières probabilités peuvent être déduites automatiquement à partir du CFG. Une fois que les probabilités de branchement de tous les arcs du graphe sont déterminées en utilisant une des méthodes citées précédemment, la fréquence d'exécution $F(N_j)$ d'un nœud N_j du graphe de flots de contrôle dépend des fréquences d'exécution pondérées de tous ses nœuds prédécesseurs. La fréquence d'exécution de chaque nœud prédécesseur N_i est multipliée par la probabilité de branchement $P(e_{ij})$ de l'arc reliant les nœuds N_i et N_j. Pour chaque nœud N_j du graphe et pour chacun de ses prédécesseurs N_i, on a :

$$x_j = F(N_j) = \sum F(N_i) \times P(e_{ij})$$

La méthode est illustrée sur l'exemple de la Figure 34.



F(S) = 1.0

$F(N_1) = 1.0 \times F(S)$

$F(N_2) = 1.0 \times F(N_1) + 0.9 \times F(N_3)$

$F(N_3) = 1.0 \times F(N_2)$

$F(N_4) = 0.1 \times F(N_3)$

Figure 34 : Obtention d'un ensemble d'équations linéaires

Ce programme comporte une boucle dont le nombre d'itérations est connu et égal à 10. En utilisant la règle décrite précédemment, l'arc de retour du nœud N_3 a une probabilité de

0,9 ((10–1)/10), alors que l'arc sortant a une probabilité de 0,1 (1/10). Notons que le nœud S est un nœud factice ajouté au début de chaque CFG et dont la fréquence d'exécution est 1. La résolution de ce système d'équations[15] donne les fréquences d'exécution suivantes :

$F(N_1) = 1$

$F(N_2) = 10$

$F(N_3) = 10$

$F(N_4) = 1$

Les auteurs calculent ensuite le temps d'exécution en moyenne, P(G), d'un graphe de flot de contrôle G de la manière suivante :

$$P(G) = \sum_{i=1}^{N} x_i c_i + \sum_{j=1}^{M} F(e)W(e)$$

où

- N est le nombre de blocs de base du CFG,
- x_i est la fréquence d'exécution de chaque bloc de base ($F(N_i)$),
- c_i est le coût associé à chaque bloc de base (i.e. le temps d'exécution),
- M est le nombre d'arcs du CFG,
- F(e) est la fréquence d'exécution de chaque arc du graphe[16] (i.e. les d_i de la Figure 31),
- W(e) est le coût associé à chacun des arcs (souvent nul) ; ce coût représente par exemple les instructions de branchements relatives au CFG.

En spécifiant les probabilités de branchement des arcs du CFG, cette approche permet également d'identifier des chemins impossibles. Malheureusement, s'il est possible de déterminer automatiquement à partir du CFG des probabilités de branchement, la méthode perd de l'intérêt dans le cas de structures de contrôle plus complexes (e.g. boucle dont le nombre d'itérations est variable ou n'est pas connu à l'avance). Dans ce cas, il est indispensable que l'utilisateur annote le code source d'informations sur les probabilités de branchement. Les auteurs suggèrent également de déterminer les probabilités de branchement à partir de traces collectées lors de simulation.

3.2.1.5. Enumération implicite des chemins

Dans le projet Cinderella [Mal95][Mal97], la méthode utilisée ne cherche pas à énumérer explicitement l'ensemble des chemins du programme, mais les considère plutôt comme faisant *implicitement* partie de la solution. A partir du nombre d'exécutions x_i d'un ensemble de blocs de base, le problème de la détermination du temps d'exécution dans le pire cas se ramène à résoudre un ensemble de problèmes de programmation linéaire en nombres entiers (ILP[17]). En d'autres termes il s'agit de trouver les valeurs des x_i (à c_i constant) permettant de maximiser l'équation (1). Les x_i dépendent des contraintes structurelles et fonctionnelles de l'application. S'il est possible d'exprimer ces contraintes

[15] Par une méthode d'élimination Gaussienne
[16] La fréquence d'exécution d'un nœud est égal à la somme des fréquences de ses arcs prédécesseurs.
[17] Integer Linear Programming

sous une forme linéaire, il est alors possible d'utiliser une méthode de résolution par ILP pour déterminer la valeur maximale de l'expression ainsi obtenue.

➢ Modélisation des contraintes structurelles

Les contraintes structurelles sont automatiquement extraites du graphe de flots de contrôle du programme [Aho86][Mal95]. Elles peuvent être déduites de la manière suivante: pour chaque nœud B_i du graphe (i.e. bloc de base), son nombre d'exécutions est égal à la fois à la somme des flots entrants et à la somme des flots sortants. En reprenant le CFG de la Figure 31, voici l'ensemble des contraintes structurelles que l'on peut déduire de ce graphe:

$x_1 = d_1 = d_2$
$x_2 = d_2 + d_8 = d_3 + d_{10}$
$x_3 = d_3 = d_4 + d_5$
$x_4 = d_4 = d_6$
$x_5 = d_5 = d_7$
$x_6 = d_6 + d_7 = d_8$
$x_7 = d_{10} = d_9$

Les appels de fonction sont représentés par des arcs dans le CFG. Chaque arc est annoté par une variable f_i similaire à la variable d_i, excepté le fait que cet arc pointe sur le CFG de la fonction appelée. La Figure 35 montre un exemple d'appel de fonction.

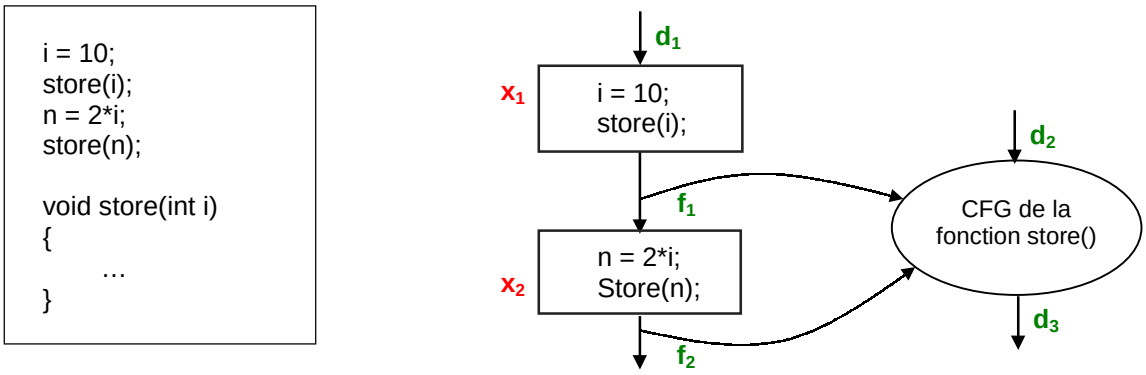


Figure 35 : Modélisation des appels de fonction

La construction des contraintes structurelles reste identique. Le nombre de fois que la fonction est exécutée est représenté par le ou les arcs pointant sur elle. Ainsi sur notre exemple, cette information est représentée par l'équation suivante:

$d_2 = f_1 + f_2$

Les autres contraintes structurelles sont:

$x_1 = d_1 = f_1$
$x_2 = f_1 = f_2$
$d_1 = 1$

Les contraintes structurelles ne représentent qu'un sous ensemble des contraintes du CFG, en particulier elles ne transportent aucune information sur les compteurs de boucles.

Ces derniers dépendent en effet de valeurs de variables qu'il n'est pas toujours possible de déterminer automatiquement à partir du CFG.

- **Modélisation des contraintes fonctionnelles**

Après avoir construit l'ensemble des contraintes structurelles, il est demandé à l'utilisateur de fournir des informations sur les bornes des boucles ainsi que d'autres informations qui dépendent des fonctionnalités du programme. Prenons par exemple le programme de la Figure 36 [Mal95]. La fonction *check_data()* vérifie les valeurs d'un tableau (*data*) de taille DATASIZE. Si l'une des valeurs du tableau est négative la fonction retourne la valeur zéro, sinon la valeur un est retournée. Supposons que la taille du tableau est préalablement définie à la valeur 10, il est alors possible d'en déduire les contraintes spécifiant la borne de la boucle :

$$1.x_1 \leq x_2$$

$$x_2 \leq 10.x_1$$

Ici, x_1 est le compteur du bloc de base avant l'entrée dans la boucle et x_2, celui du premier bloc de base à l'intérieur de la boucle. Notons qu'il est possible de déterminer automatiquement ces deux contraintes à condition que l'utilisateur fournisse les valeurs 1 et 10.

```
1:                  check_data()
                    {
2:                      int i, morecheck, wrongone;

3:      x1              morecheck=1; i=0; wrongone=-1;
4:                      while (morecheck) {
5:      x2                if (data[i] < 0) {
6:      x3                  wrongone = i; morecheck = 0;
7:                        }
8:                        else
9:      x4                  if (++i ≥ DATASIZE)
10:     x5                    morecheck = 0;
11:     x6              }

12:     x7              if (wrongone >=0)
13:     x8                return 0;
14:                     else
15:     x9                return 1;
16:                 }
```

Figure 36 : Exemple de contraintes fonctionnelles disjonctives

Les autres contraintes sont destinées à affiner éventuellement la borne recherchée. Par exemple, les lignes **6** et **10** sont mutuellement exclusives, i.e. les blocs B_3 et B_5 ne sont jamais exécutés ensemble. Ceci se traduit par la contrainte fonctionnelle disjonctive suivante :

$$(x_3 = 0 \ \& \ x_5 = 1) \mid (x_3 = 1 \ \& \ x_5 = 0)$$

Nous ne sommes plus en présence de contraintes linéaires. Il s'agit néanmoins d'une disjonction de deux ensembles de contraintes conjonctives linéaires. De plus, les lignes **6** et **13** sont exécutées un même nombre de fois, ce qui se traduit par la contrainte fonctionnelle suivante :

$$x_3 = x_8.$$

- **Méthode de résolution**

L'ensemble des contraintes structurelles de l'application est un ensemble de contraintes conjonctives, i.e. qui doivent être satisfaites simultanément. Les contraintes *fonctionnelles* peuvent présenter, comme nous l'avons vu, des contraintes disjonctives, formant des ensembles mutuellement exclusifs de contraintes conjonctives. Parmi ces ensembles se trouve une solution satisfaisant toutes les contraintes de l'application. Afin de caractériser la borne maximale, chaque ensemble de contraintes fonctionnelles est combiné avec les contraintes structurelles. Chacun d'eux est traité par un solveur ILP dans le but de maximiser l'équation (1). Le solveur ILP retourne la valeur maximale de l'expression (1), ainsi que la valeur de tous les compteurs x_i de chaque bloc de base B_i. Le maximum obtenu parmi tous les ensembles de contraintes détermine la borne maximale.

Le temps nécessaire à la résolution de l'équation (1) par un solveur ILP dépend à la fois du nombre d'ensembles de contraintes fonctionnelles et du temps requis pour résoudre un ensemble : la taille des ensembles de contraintes double à chaque présence d'une contrainte disjonctive. Dans l'exemple de la Figure 36 nous avons en effet deux ensembles de contraintes :

1er ensemble	*2ème ensemble*
$x_1 - x_2 \leq 0$	$x_1 - x_2 \leq 0$
$10x_1 - x_2 \geq 0$	$10x_1 - x_2 \geq 0$
$x_3 = 0$	$x_3 = 1$
$x_3 - x_8 = 0$	$x_3 - x_8 = 0$
$x_5 = 1$	$x_5 = 0$

En pratique, les auteurs remarquent que l'ajout de contraintes conduit souvent à définir des ensembles de contraintes qui s'annulent (par exemple $x_i > 1$ avec $x_i = 0$). Il s'agit alors de détecter puis d'éliminer ces ensembles nuls avant leur analyse par le solveur ILP. La complexité pour résoudre chaque problème ILP est en général un problème NP-complet. Pour résoudre le problème, il est nécessaire de restreindre le type de contraintes fonctionnelles utilisées, ce qui permet de se ramener à un problème de « *network flow* » qui peut être résolu en un temps polynomial [Mal95].

En théorie, l'information minimum que doit fournir l'utilisateur pour effectuer l'analyse de performance est le nombre d'itérations de toutes les boucles. En pratique, il est souvent indispensable de fournir d'autres contraintes fonctionnelles afin d'améliorer les estimations de performance. C'est ce que nous montrons sur l'exemple de la FFT traité dans [Kwi99]. Nous insistons également sur le fait qu'il n'est pas toujours trivial de déterminer les « bonnes » contraintes fonctionnelles, c'est-à-dire celles qui permettent une estimation réaliste des performances dans le pire cas.

3.2.1.6. Détermination des contraintes fonctionnelles pour l'algorithme de FFT

Cette méthode intéressante présente des inconvénients vis à vis de son utilisation pratique sur des applications industrielles. Pour illustrer ce point nous décrivons son application à un algorithme de FFT 1024 points (Figure 37) [Kwi99]. Le CFG correspondant est décrit sur la Figure 38 .

```
/* FFT 1024 points, décimation en temps */

Initialisation des variables                                   Bloc D

for (i=0; i<nb_pass; i++) {
    for (j=0; j<nb_group; j++) {
        for (k=0; k<nb_bfly; k++) {

            Calcul du papillon

        }                                                      Bloc A

    Mises à jour de variables entre chaque groupe
    }                                                          Bloc B

Mises à jour de variables entre chaque passe                   Bloc C

}
```

Figure 37 : Programme C de la FFT

L'ensemble des contraintes structurelles déduites du CFG du programme sont les suivantes :

x1 = d1 = d2
x2 = d2 + d3 = d4 + d5
x3 = d5 = d6
x4 = d6 + d7 = d8 + d9
x5 = d8 = d3
x6 = d9 = d10
x7 = d10 + d11 = d12 + d13
x8 = d12 = d7
x9 = d13 = d11

Pour calculer le temps d'exécution dans le pire cas il est nécessaire sur cet exemple que l'utilisateur fournisse des contraintes fonctionnelles décrivant l'évolution du programme. Nous détaillons dans ce qui suit l'étude menée et l'évolution des estimations en fonction des contraintes fonctionnelles fournies au solveur ILP[18].

1) L'utilisateur ne fournit que les bornes des trois boucles imbriquées, à savoir :

x1 = 1
x2 = (nb_pass + 1) x1
x4 = (nb_group +1) x3
x7 = (nb_bfly + 1) x6

[18] Le solveur ILP est celui utilisé dans le Projet Cinderella [Mal95].

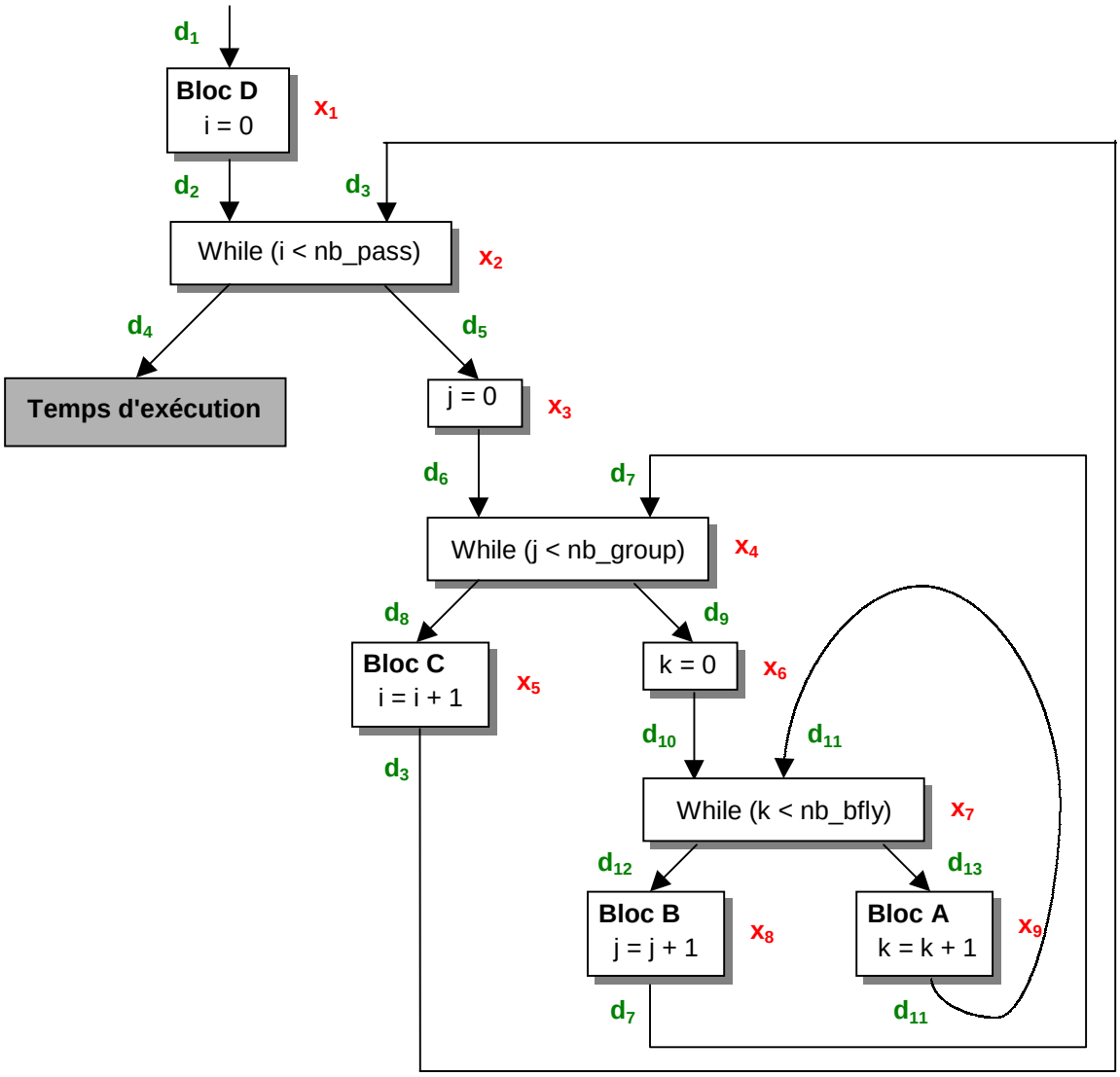


Figure 38 : Graphe de flots de contrôle de la FFT

Comme on considère une FFT de 1024 points, « *nb_pass* » est une constante égale à 10. Par contre, les variables « *nb_group* » et « *nb_bfly* » évoluent au cours du temps en fonction de l'étage de FFT considéré. Leurs valeurs sont inversement proportionnelles en puissance de deux de telle sorte que :

$$nb\_group * nb\_bfly = N/2 = 512 \quad (2)$$ [19]

Si l'utilisateur ne prend pas en compte cette relation et affecte les bornes maximum aux trois boucles prises séparément, soit *nb_pass* = 10, *nb_group* = 512 et *nb_bfly* = 512, alors un pessimisme excessif est introduit. En effet, la **borne estimée** (b_e) dans le pire cas est estimée à *68 249 894* cycles. La **borne mesurée** (b_m) sur le OakDSPCore, obtenue en utilisant le simulateur de niveau instruction fourni avec le DSP, est égale à *132 336* cycles. Bien entendu les bornes estimée et mesurée se basent sur des valeurs de c_i identiques. En utilisant la définition du pessimisme introduite dans [Mal95] et définie par :

$$\text{Pessimisme} = \frac{b_e - b_m}{b_m}$$

[19] Car N = 1024.

le pessimisme est de 514 73%, ce que l'on peut qualifier d'excessif à juste titre.

2) Afin de prendre en compte la relation définie en (2), une solution consiste à déplier le CFG. Pour chacune des passes nous avons ainsi les valeurs montrées sur la Figure 39 :

i	0	1	2	3	4	5	6	7	8	9
nb_group	512	256	128	64	32	16	8	4	2	1
nb_bfly	1	2	4	8	16	32	64	128	256	512

Figure 39 : Nombre de groupes et de papillons par étage de FFT

En sommant pour chaque passe (i.e. chaque *i*) le nombre de cycles obtenu avec le solveur ILP, la borne estimée dans le pire cas est de *151 828* cycles, soit un pessimisme de seulement 14,73%. Si cette méthode permet de limiter le pessimisme, elle parait difficilement exploitable dans le cas d'applications comportant un grand nombre d'itérations.

3) En étudiant le CFG de la Figure 38 on remarque que le nombre d'itérations x_9 est lié au compteur x_3 par le produit de « *nb_group* » par « *nb_bfly* », soit N/2. De plus, x_4 est forcément supérieur ou égal à deux fois x_3 et inférieur ou égal à (N/2 + 1). Pour une FFT de 1024 points les contraintes fonctionnelles sont donc :

$$x_9 = 512\, x_3 \quad (3)$$
$$x_4 \leq (512 + 1)\, x_3 \quad (4)$$
$$x_4 \geq 2\, x_3 \quad (5)$$

Ces contraintes laissent au solveur ILP le choix des valeurs pour « *nb_group* » et « *nb_bfly* » pour que le produit soit toujours égal à N/2. La borne estimée dans le pire cas à l'aide de ces contraintes est de *225 574* cycles, soit un pessimisme de 70%. Ces contraintes ne sont donc pas satisfaisantes et une nouvelle contrainte fonctionnelle est introduite permettant finalement de solutionner le problème. En étudiant à nouveau le CFG, la relation suivante entre les blocs de base B_6 et B_1 est déterminée :

$$x_6 = 1023\, x_1 \quad (6)$$

Cette contrainte provient du fait que le bloc de base B_6 est exécuté 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 = 1023 fois. Cette contrainte fonctionnelle rend inutile les contraintes (4) et (5) définies précédemment. En utilisant les relations (3) et (6), la borne estimée dans le pire cas est alors de *151 828* cycles, c'est-à-dire exactement la même estimation que celle obtenue lors du dépliement des boucles sur les groupes et les papillons.

4) Conclusions

L'exemple de la FFT montre qu'il n'est pas trivial de déterminer les contraintes fonctionnelles permettant d'obtenir une estimation du temps d'exécution dans le pire cas sans introduire de pessimisme excessif. De plus, si l'utilisateur ne fournit que le nombre d'itérations maximum de toutes les boucles du programme, le résultat est catastrophique. En ajoutant des contraintes fonctionnelles, on s'approche de la performance mesurée. On constate cependant que sans l'équation (6) le résultat n'est pas satisfaisant. Ceci est dû à l'imprécision des informations (i.e. équations (3), (4) et (5)) sur les chemins possibles du

programme. En d'autres termes, une estimation pessimiste considère des chemins infaisables comme possible. La solution consiste comme nous l'avons vu à fournir de nouvelles contraintes fonctionnelles. Le problème est alors de déterminer si les contraintes fonctionnelles ajoutées sont celles nécessaires et suffisantes pour une estimation précise des performances.

3.2.2. Les approches dynamiques

Les approches dynamiques [Gon93][Par93a][Lee96] font référence aux méthodes récupérant des informations par l'exécution du programme sur un ensemble de données d'entrée, ou séquences de test. Dans [Gon93], les auteurs suggèrent l'utilisation de méthodes de simulation pour déterminer les probabilités de branchement. Dans [Lee96] seules les instructions de contrôle sont simulées, accélérant considérablement les temps de simulation. Si les séquences de test ont un impact significatif sur les performances, c'est un domaine qui a fait l'objet de peu de recherche [Mal97].

Pour les parties de code dont le temps d'exécution ne dépend pas des données, les x_i sont uniques et peuvent être obtenus par un *profiling* du code source en utilisant un PC ou une station. Citons par exemple l'outil « tcov » de Sun Microsystems disponible sur toute station UNIX. Le *profiling* évite d'effectuer une simulation très coûteuse en temps du code assembleur. En réalité le problème est plus complexe puisqu'il faut prendre garde à la correspondance des blocs de base entre le code source et le code assembleur généré. En effet, certaines optimisations du compilateur peuvent modifier la structure du programme, donc le CFG. Un exemple typique est la génération d'instruction conditionnelle qui fait disparaître généralement un bloc de base.

Les valeurs de la séquence de test doivent correspondre au temps d'exécution dans le pire cas ce qu'il est impossible de déterminer dans le cas général. En effet, le comportement dépend non seulement des données mais aussi de l'état dans lequel se trouve le programme à un instant donné de l'exécution. Les approches dynamiques sont donc utilisées de préférence pour l'analyse de performances statistiques. Dans ce cas, il est nécessaire de spécifier la distribution des données d'entrée. Il n'existe malheureusement pas de mécanisme permettant de spécifier facilement ces données et il est courant d'utiliser un large ensemble d'échantillons d'entrée. La situation la plus acceptable est certainement pour l'analyse de performance en moyenne où les résultats sont obtenus à partir d'un jeu réduit de données typiques. Cette approche marche bien tant que les séquences de test sont représentatives des données d'entrée, ce qui n'est pas une tâche triviale dans le cas d'applications complexes.

3.2.3. Une approche mixte

Les méthodes statiques sont imprécises, contraignantes pour l'utilisateur et donnent souvent lieu à un pessimisme excessif car elles prennent en compte des chemins impossibles. Ernst et al. [Ern97] ont eu l'idée d'utiliser une approche mixte (**SYMTA**[20]) combinant les techniques de simulation et d'estimation formelles. L'objectif est d'améliorer la précision des résultats tout en réduisant les temps d'analyse. Le principe consiste à exploiter les propriétés du programme et de l'architecture pour simplifier l'analyse des chemins.

[20] Symbolic hybrid timing analysis

Le processus d'estimation commence tout d'abord par la construction d'un CFG hiérarchique du programme. Les auteurs cherchent alors à déterminer les parties du programme dont le contrôle (i.e. l'exécution) ne dépend pas des valeurs des données d'entrée (SFP[21]). Les MFP[22] sont les parties du programme pour lesquelles il existe différents chemins possibles suivant les données d'entrée. L'algorithme de FFT de la Figure 37 est un programme complètement SFP : le nombre d'itérations des trois boucles est indépendant des données d'entrée. Le programme de la Figure 36 contient une partie (if.. then.. else..) dont l'exécution dépend des valeurs du tableau « *data* ».

Bien que la plupart des programmes contiennent des parties MFP, les auteurs observent que pour les algorithmes de traitement du signal des systèmes embarqués, il existe de longues parties de programme ayant des propriétés SFP. Ces parties, une fois regroupées, peuvent être traitées plus simplement que les parties MFP. Pour cela il est nécessaire d'isoler les parties SFP des parties MFP, ce qui revient à partitionner le CFG. La détermination des parties SFP s'effectue tout d'abord par une analyse globale des flots de données [Aho86]. Comme cette dernière ne peut couvrir tous les cas de dépendances, une simulation symbolique des blocs de base [Ern95] complète l'analyse globale des flots de données. Les inconvénients des méthodes formelles ne concernent plus que les parties MFP du programme pour lesquelles une approche ILP [Mal95] est utilisée. Les SFP sont quant à elles simulées (sur un SPARC) avec une séquence de test fournie par l'utilisateur puisque pour ces parties, les x_i sont uniques.

Cette approche nous semble très attrayante par rapport aux méthodes entièrement statiques pour deux raisons principales. Tout d'abord il parait difficile de demander à un programmeur, même expérimenté, de fournir les bornes de toutes les boucles ainsi que des informations sur les chemins impossibles (que ce soit via le langage IDL ou des contraintes fonctionnelles). La deuxième raison est la rapidité de l'analyse par rapport aux méthodes formelles. En effet, seules les parties MFP nécessitent une étude plus détaillée du programme.

Nous avons présenté quelques approches pour estimer le nombre x_i d'exécutions de chaque bloc de base d'un programme. L'estimation du temps d'exécution global nécessite de déterminer également le temps d'exécution c_i de chaque bloc de base.

[21] Single Feasible Part
[22] Multiple Feasible Part

3.3. La détermination des c_i

3.3.1. Les techniques d'analyses du temps d'exécution

Il existe généralement deux types de techniques pour déterminer les temps d'exécution d'un bloc de base.

- La méthode la plus fréquemment utilisée est l'**addition des temps d'exécution** des instructions appartenant à un même bloc de base à partir d'un modèle de l'architecture cible. Les estimations dépendent directement de la précision du modèle du processeur. Dans beaucoup de cas, les temps d'exécution sont déterminés à partir de tables établies après étude du manuel décrivant le jeu d'instructions du processeur [Pus89][Par90][Gon93][Mal95][Lee96][Ott97]. C'est une méthode très rapide qui permet de considérer des temps d'exécution minimum et maximum. Le principal inconvénient de cette approche vient des instructions dont le temps d'exécution dépend des données.

- Une autre solution consiste à effectuer **une simulation** des instructions assembleur de chaque bloc de base en utilisant un simulateur de niveau instruction [Ern95][Par93a][Par93b]. Cette approche a pour principal inconvénient des temps de simulation très longs. C'est un facteur important pour les méthodes de développement d'un code temps réel qui nécessitent un nombre important d'itérations pour sa mise au point (respect des contraintes).

Dans [Ern97] les auteurs utilisent un simulateur de niveau instruction pour le calcul des c_i. Pour les parties de code non couvertes par la séquence de test, ils utilisent alors la méthode d'addition des temps d'exécution.

3.3.2. Les critères de variation du temps d'exécution d'un bloc de base

Le temps d'exécution d'un bloc de base peut varier sensiblement suivant les caractéristiques architecturales du processeur cible [Ern93] :

- instruction (i.e. bloc de base) dont le nombre de cycles dépend des données,
- architecture superscalaire,
- architecture pipeline avec exécution entrelacée de blocs de base,
- architecture avec des mémoires caches.

La Table 1 montre que le choix d'une technique appropriée dépend non seulement des caractéristiques architecturales du processeur mais aussi du type d'estimation souhaité.

	Addition des temps d'exécution	Simulation
Instructions dont le nombre de cycles dépend des données	Les résultats dans le pire cas sont obtenus en prenant les temps d'exécution maximum de chaque instructions (pessimisme excessif). Lors d'une mesure de performance statistique, une étude préalable de la dynamique des données peut permettre de déterminer un nombre de cycles par instruction plus représentatif.	Résultats dont la précision dépend des séquences de test.
Architectures superscalaires	Méthode inappropriée.	
Pipeline	Méthode imprécise (pas de prise en compte des gains dus au pipeline).	
Mémoire cache	Utilisation d'une valeur de probabilité de défaut de cache (toujours égale à 1 par exemple pour le calcul du pire cas [Mal95]) ou prise en compte dans le modèle du processeur.	Performance précises pour les parties de code qui ne dépendent pas des données d'entrée sinon dépend des séquences de test.

Table 1 : Critère de choix d'une technique appropriée

3.3.3. Importance de la précision du modèle de processeur

Dans [Mal95], un modèle simplifié du processeur est utilisé pour déterminer les coûts associés à chaque bloc de base. Pour chaque instruction assembleur d'un bloc de base, les auteurs analysent les instructions adjacentes à l'intérieur de ce bloc de base et déterminent le temps d'exécution à partir du manuel de l'architecture du processeur cible. Tous les temps de calcul sont alors additionnés afin d'obtenir le résultat pour le bloc de base. Si ce modèle prend raisonnablement en compte le pipeline, les auteurs reconnaissent que leur modèle de processeur est trop simple pour prendre en compte les problèmes liés aux mémoires caches. Pour le calcul du temps d'exécution dans le pire cas par exemple, ils considèrent que l'exécution donne toujours lieu à des défauts de cache (et inversement pour le calcul dans le meilleur cas), ce qui conduit inévitablement à un pessimisme excessif.

Les auteurs ont étendu cette approche pour les architectures pipeline[23] ou possédant une mémoire cache [Mal96]. Ces caractéristiques architecturales sont modélisées par des contraintes linéaires (qui s'ajoutent aux contraintes structurelles et fonctionnelles) et le calcul du pire cas est toujours obtenu en maximisant l'expression (1) par une méthode ILP. Dans [Ott97] le CFG du programme est annoté avec des variables représentant l'historique de l'exécution (et non plus seulement des valeurs cumulatives) afin de déterminer l'état du cache et du pipeline avant l'exécution de chaque bloc de base. Pour estimer précisément le gain de temps d'une mémoire cache, l'état de cette dernière est propagé de bloc de base en bloc de base.

Il est intéressant de noter que pour déterminer l'utilisation du pipeline ou les défauts de cache il est nécessaire de connaître le temps d'exécution dans le pire cas (i.e. le chemin le plus long dans le CFG). D'un autre côté, comme le pipeline ou les mémoires caches ont un impact sur les performances, il est indispensable de connaître les défauts de cache et les aléas de pipeline pour déterminer le temps d'exécution dans le pire cas. Cette dépendance

[23] Les gains de performance apportés par le pipeline sont généralement plus facile à modéliser que le comportement des mémoire caches.

mutuelle rend plus complexe la modélisation du problème de l'analyse des performances pour des architectures possédant de telles caractéristiques.

3.4. Conclusion

Les approches statiques présentées [Pus89][Par90][Mal95] demandent à l'utilisateur de fournir les bornes de toutes les boucles du programme. Dans certains cas [Pus89][Sha89], il est même indispensable que le nombre d'itérations de chaque boucle du programme soit borné et connu avant l'exécution du programme afin de rendre possible le calcul du temps d'exécution maximum.

D'un point de vue industriel, ces approches nous paraissent très contraignantes et donc difficilement envisageables. En effet, dans le cas d'applications de taille importante le travail demandé à l'utilisateur est très important. Dans [Kwi99], l'auteur montre que ces informations ne sont pas toujours une condition suffisante pour des analyses précises de performance. D'autre part, déterminer une borne correcte est un tâche très complexe dans le cas de boucles imbriquées interdépendantes ou dont le nombre d'itérations dépend des valeurs des données en entrée.

Un autre point commun à l'ensemble de ces méthodes est que l'amélioration de la précision des estimations nécessite l'ajout par l'utilisateur d'informations sur le comportement du programme. Il s'agit de renseigner l'outil sur des relations entre différentes parties du programme qu'il ne peut « voir » ou l'aider dans la recherche des chemins impossibles. Ces informations se retrouvent sous différentes formes suivant l'approche utilisée : les contraintes fonctionnelles [Mal95], les extensions du langage utilisées dans [Pus89] et [Gon93] ou le langage IDL [Par89]. Or, nous avons montré avec l'exemple de la FFT qu'il n'est pas toujours trivial de trouver les bonnes contraintes fonctionnelles dans le cas d'applications complexes (i.e. structures de contrôle complexes et/ou mutuellement dépendantes). De plus, ces méthodes ne garantissent pas que toutes les relations entre différentes parties du programme sont exploitées. L'ajout d'informations par l'utilisateur nous parait une approche contraignante, longue et ne garantissant pas une estimation dans le pire cas sans pessimisme ou optimisme excessif dans le cas de contraintes fonctionnelles erronées.

Dans le cadre de cette étude, nous recherchons une méthode pour le calcul des c_i et des x_i qui soit adaptée aux contraintes industrielles imposées par VLSI Technology. Bien que le calcul des performances dans le pire cas soit une mesure intéressante pour VLSI Technology, les méthodes que nous avons présenté sont trop contraignantes pour l'utilisateur et donc inadaptées aux besoins industriels. L'une des principales contraintes est en effet la rapidité d'obtention des estimations. En conséquence, pour le calcul des x_i notre choix s'est porté sur une approche dynamique (statistique) basée sur l'exécution de séquences de test représentatives[24]. Le point critique d'une telle méthode porte sur la qualité de la séquence de test. Il est en effet indispensable de fournir des données d'entrée représentatives de l'application considérée (e.g. des voix de femme ou d'homme dans différentes langues pour un algorithme de compression de parole) et dont le taux de couverture est le plus élevé possible. Notons qu'il existe à cet effet des outils tel que « tcov » de Sun Microsystems qui permettent de déterminer le pourcentage de code non pris

[24] Des séquences de test représentatives accompagnent généralement le code C d'une application fournie par l'ITU ou l'ETSI.

en compte lors de l'exécution du programme avec la séquence de test. C'est un des avantages des approches formelles par rapport aux approches dynamiques : il n'y a pas besoin de séquences de test.

Pour le calcul des c_i le choix s'est porté sur une méthode basée sur l'addition des temps d'exécution des instructions générées par le compilateur appartenant à un même bloc de base. Cette approche est appropriée à la classe de processeurs DSP concernée par cette étude puisque ces derniers ne sont pas des processeurs superscalaires et ne possèdent pas de mémoire cache. Dans le cas du OakDSPCore, le pipeline est complètement transparent. Il existe par contre des instructions dont le nombre de cycles dépend des données. Ce problème se limite cependant aux instructions de branchement conditionnels dont le nombre de cycles dépend du test sur la condition. Pour le PalmDSPCore il existe également des instructions dont le nombre de cycles dépend des données mais aussi des optimisations effectuées par le compilateur (ou éventuellement par le programmeur lors d'un codage manuel). Ces problèmes concernent à nouveau les instructions conditionnelles (branchement retardé) dont le nombre de cycles peut varier entre 2 et 5 cycles. L'approche que nous avons choisi pour traiter ce type de problème consiste à déterminer un nombre de cycles moyen par instruction déterminé à partir de mesures statistiques sur des données représentatives.

Notre approche est donc basée sur l'exécution du code avec une séquence de test pour déterminer les x_i et l'addition des instructions pour le calcul des c_i. Une approche de ce type a été développée spécifiquement pour les DSP par [Lee96]. Les auteurs sont en effet arrivés à la même conclusion : il est essentiel d'effectuer des exécutions réalistes pour obtenir des estimations précises sans trop d'effort pour l'utilisateur. Leur technique consiste à retirer du programme les parties effectuant des calculs sur des données et d'extraire les instructions de contrôle. Seules ces instructions sont simulées, accélérant considérablement la phase d'analyse du nombre d'exécutions des blocs de base (le nombre d'itérations des boucles plus particulièrement). Pour le calcul des c_i, ils additionnent les instructions appartenant à un même bloc de base.

Dans [Lee96] les auteurs soulèvent le problème du niveau de description de l'application pour effectuer des estimations. Ils considèrent qu'une approche basée sur l'analyse du code assembleur est plus appropriée pour l'analyse de performance des DSP. En effet, il est nécessaire de prendre en compte les optimisations effectuées par le compilateur. Mais, l'estimation de performance à partir d'un code assembleur généré par un compilateur C représente le plus souvent une mesure de l'inefficacité du compilateur C (par rapport à un code optimisé manuellement). En effet, les performances obtenues ne sont pas représentatives de ce que pourrait faire un programmeur expérimenté en écrivant le code assembleur manuellement. Gong et al. [Gon93] utilisent un rapport d'optimisation pour estimer un code optimisé à partir du code assembleur généré par le compilateur. Cette approche nous semble imprécise puisque le facteur d'optimisation peut varier sensiblement d'une application à une autre suivant le type de traitement effectué. Un autre problème est qu'un code assembleur optimisé manuellement n'est pas toujours disponible et il n'est plus envisageable d'effectuer ce travail dans le futur. C'est pour cette raison que nous proposons de fournir une estimation d'un code optimisé à partir du code C de l'application. Dans la suite nous présentons la méthode d'estimation des performances d'un code assembleur optimisé, c'est-à-dire comme s'il avait été écrit par un programmeur expérimenté.

Présentation générale de la méthode d'estimation

Les compilateurs DSP actuellement disponibles sont le plus souvent incapables de générer un code assembleur respectant les fortes contraintes de performance et de coût (i.e. taille du code) des systèmes temps-réel embarqués. Cette inefficacité oblige le programmeur à coder directement en assembleur tout ou partie de l'application. C'est la méthode « classique » de développement d'un code assembleur optimisé sur DSP. Mais le codage et l'optimisation d'un code assembleur est un travail long[25] et fastidieux. Cette situation est de plus en plus inacceptable compte tenu de l'augmentation de la complexité des applications, des architectures DSP[26] et de la pression du *time-to-market*. Pour ces raisons et pour répondre aux besoins de la société VLSI Technology, nous avons imaginé une méthode de développement de code assembleur optimisé qui cherche à maximiser l'utilisation du compilateur C. Cette méthode est basée sur la comparaison de performance du code assembleur généré par le compilateur et de l'estimation de la performance d'un code optimisé.

4.1. Réduction du temps de développement d'une application sur DSP

4.1.1. Une approche de plus haut niveau

La réduction du temps de développement d'un code assembleur optimisé pour DSP passe par une approche de plus haut niveau. Il s'agit en effet d'utiliser plus systématiquement le compilateur afin d'écrire le moins possible de code assembleur. Or, si les compilateurs pour DSP sont globalement inefficaces, nous savons également qu'ils peuvent être localement efficaces et que les performances du code assembleur généré dépendent fortement du code C lui-même. Cette sensibilité est d'autant plus forte que l'architecture du DSP est hétérogène. Il est possible d'améliorer sensiblement la qualité du code assembleur en modifiant le code C d'origine pour le compilateur cible (i.e. l'architecture cible). L'objectif est de tenir compte du schéma d'exécution des DSP dans l'écriture du code C. Une étude intéressante à ce sujet a été menée en 1997 sur différents DSP du commerce [Lev97]. Nous présentons dans ce qui suit les principales règles de réécriture du code C pour le OakDSPCore. Une présentation plus détaillée de ces règles est fournie en Annexe 1.

4.1.2. Les règles de réécriture indépendantes des DSP

Le processus d'optimisation commence généralement par la modification du code C en respectant un ensemble de règles d'écriture indépendantes de l'architecture du DSP cible.

- **L'utilisation de pointeurs** [Lie96] au lieu de tableaux indicés permet généralement d'utiliser plus efficacement les modes d'adressage complexes des DSP (modulo par exemple).

- Comme les DSP ont un nombre de registres très restreint, l'écriture du code C qui favorise l**a réduction de la durée de vie des variables** peut aider le compilateur lors de la phase d'allocation de registres.

[25] Souvent plusieurs mois de travail.
[26] Ecrire un code assembleur optimisé pour un DSP possédant une architecture VLIW est une tâche très complexe.

- **L'utilisation de variables locales** au lieu de variables globales permet généralement d'éviter des sauvegardes temporaires (*spilling*) de données en mémoire.

- Minimiser l'utilisation des variables dont **le type a une largeur supérieure à la largeur naturelle des données** du processeur (e.g. utiliser un type « long » (32 bits) pour un DSP 16 bits), afin d'éviter l'appel à des routines définies en librairie très coûteuses en nombre de cycles.

- **Limiter le nombre de variables et de pointeurs** facilite le travail du compilateur lors de la phase d'allocation de registres. Cette règle est d'une utilisation délicate puisqu'elle augmente la durée de vie des variables.

- **La factorisation du code**[27] permet de localiser plus finement les parties critiques d'une application.

4.1.3. Les règles de réécriture dépendantes d'une classe de DSP

Nous décrivons ici les extensions que nous utilisons pour optimiser un code C pour le OakDSPCore. Ces extensions permettent de renseigner le compilateur sur des aspects qu'il ne peut pas prendre en compte seul. Nous indiquons également d'autres extensions que nous avons pu trouver dans la littérature.

- **L'utilisation de type de donnée orienté DSP**. Les accumulateurs du OakDSPCore ont une largeur de 36 bits. Comme en C le type « long » est codé sur 32 bits, un nouveau type a été ajouté au langage: *acc_t* [Occ96]. Ce type de donnée est particulièrement recommandé pour les opérations de type « MAC ». Dans [Kre94] les auteurs définissent non seulement un type accumulateur (de 40 bits) mais aussi de nouveaux types permettant de représenter des données en point fixe. Dans [Hof93], un type « complexe » (utile pour le calcul de la FFT par exemple) est défini ainsi qu'un ensemble de fonctions mathématiques utilisant ce type de données.

- Pour le OakDSPCore, un nouveau mot clé (« __bkrep__ ») a été défini. Placé devant une structure itérative, il facilite la tâche du compilateur pour générer une boucle gérée matériellement. Cette extension est également présente dans les travaux présentés en [Hof93][Kre94] et [Yos96] puisque les DSP correspondants contiennent également les mécanismes de gestion matérielle des boucles.

- **La localisation des données dans les différents bancs de mémoire** permet d'optimiser la génération des opérations « MAC » opérant sur deux données en mémoire en un seul cycle (cas où le DSP possède deux bancs de mémoire). Ceci s'effectue avec les extensions (nouveaux mots clés) « __xram__ » et « __yram__ » que l'on insére dans le code source au moment de la déclaration des variables. On retrouve cette possibilité dans [Kre94] et [Yos96].

4.1.4. Exemple de l'utilisation de ces extensions pour le OakDSPCore

La Figure 40 montre un exemple typique de l'utilisation de règles d'écriture et des extensions apportées au langage C pour un filtre à réponse impulsionnelle finie (FIR). Dans cet exemple, les règles utilisées sont commentées à l'intérieur du programme modifié. Il est important de noter que grâce à l'utilisation du préprocesseur, le programme ainsi modifié est

[27] Regrouper des traitements identiques dans une fonction commune.

portable, c'est-à-dire qu'il est toujours possible de le compiler sur le processeur hôte. Ce dernier remplace par exemple le type « acc_t » par le type « long » avant la phase de compilation sur le processeur hôte.

Programme d'origine

```
short a[10];
short b[10];
int i;
long acc;

acc = 0;
for (i=0; i<10; i++)
      acc += (long)a[i] * (long)b[i];
```

Programme modifié

```
__xram__ short a[10];    /* déclaration des données dans        */
__yram__ short b[10];    /* différents bancs de mémoire.        */
__xram__ short *p1;
__yram__ short *p2;
int i;
acc_t acc;   /* Type accumulateur */
p1 = a;
p2 = b;

acc = 0;
__bkrep__    /* indication d'une boucle matérielle */
for (i=0; i<10; i++)
      /* pointeurs au lieu de tableaux indicés */
      acc += (long)*p1++ * (long)*p2++;
```

Figure 40 : Exemple de réécriture d'un code C pour le OakDSPCore

4.1.5. Conclusion

La modification du code C d'origine et plus particulièrement l'utilisation d'extensions doit rester limitée pour éviter un travail trop important de la part du programmeur. Comme pour les applications de traitement du signal les parties critiques en temps d'exécution représentent généralement une faible partie de la taille totale du code (typiquement les instructions appartenant à des boucles imbriquées), le travail d'optimisation doit évidemment s'effectuer prioritairement sur ces parties critiques afin d'éviter de perdre inutilement du temps sur des parties de code ayant peu d'influence sur les performances globales de l'application.

En l'absence d'outil adapté, la localisation de ces parties critiques s'effectue par une simulation du code assembleur généré par le compilateur. Malheureusement, le principal inconvénient de cette approche est le temps de simulation très long. Il faut en effet compter plusieurs heures voire plusieurs jours de simulation pour une application complète avec une séquence de test représentative (plusieurs secondes de parole dans le cas d'un algorithme de compression comme G.728 [Rec94]). De plus, pour obtenir des résultats fonction par fonction, il est nécessaire d'indiquer à l'outil le début et la fin de chacune d'elles. Ces problèmes ont motivé l'élaboration de nouveaux outils permettant d'accélérer ce processus.

4.2. La méthode d'estimation logicielle

Nous proposons une méthode d'estimation logicielle [Peg99a] qui, à partir d'une description en C de l'application permet de :

- localiser les parties critiques de l'application,
- fournir une mesure de qualité du code assembleur généré par le compilateur.

Cette mesure de qualité est basée sur une estimation d'un code assembleur optimisé, c'est-à-dire comme s'il avait été écrit à la main par un programmeur expérimenté. En comparant les performances du code assembleur généré par le compilateur avec cette borne, il est alors possible de déterminer les parties de l'application qui ont été compilées efficacement ou non. La Figure 41 compare les cycles de développement d'une application avec la méthode proposée et une approche entièrement manuelle.

Soit une application de traitement du signal que l'on souhaite implémenter sur un DSP en respectant une contrainte temps-réel. L'approche classique consiste à écrire directement le code en assembleur sur le DSP en partant généralement du code C de l'application (que l'on suppose disponible et valide[28]). Soit t_0, la date de début du travail d'implémentation. La durée $t_2 - t_0$ représente le temps nécessaire pour obtenir un code assembleur valide. A t_2, le code assembleur requiert un nombre P_2 de cycles. Afin de respecter la contrainte de temps imposée par le système, il est nécessaire d'optimiser le code assembleur. Pour cela, l'utilisateur effectue une simulation du code assembleur afin de déterminer les parties critiques de l'application. Le travail d'optimisation s'effectue alors sur les fonctions les plus coûteuses en temps et à t_4, le code assembleur ainsi optimisé permet de respecter la contrainte de temps.

Soit à présent un second programmeur dont l'objectif est d'utiliser au maximum le compilateur C. A t_0, on obtient par compilation un code assembleur dont la performance P_3 est généralement très supérieure à la performance d'un code écrit manuellement. En utilisant les règles d'écriture du code C décrites précédemment, le gain de performance est souvent rapide et assez spectaculaire. A t_1, après la phase d'optimisation du code C on obtient une performance P_1, certes meilleure que P_3, mais qui ne permet toujours pas de respecter la contrainte de temps. Il est alors généralement nécessaire de coder certaines parties de l'application (en pratique environ 5 à 10% du code selon des expérimentations effectuées à VLSI Technology) directement en assembleur. En suivant cette approche mixte, la Figure 41 montre qu'il est possible théoriquement de réduire le temps de développement d'une durée égale à $t_4 - t_3$. Notons qu'il existe a priori une différence Δ_1 entre un code entièrement optimisé manuellement et un code optimisé en utilisant une approche mixte. Si pour les DSP à parallélisme limité (e.g. OakDSPCore, TMS320C54) il semble difficile pour un compilateur de faire mieux qu'un programmeur expérimenté, cette tendance a des chances de s'inverser pour les nouveaux DSP comportant plus de parallélisme ou de niveaux de pipeline (e.g. architecture VLIW).

[28] Ce qui signifie généralement que le code C exécute correctement un ensemble de séquences de test accompagnant la spécification de l'application.

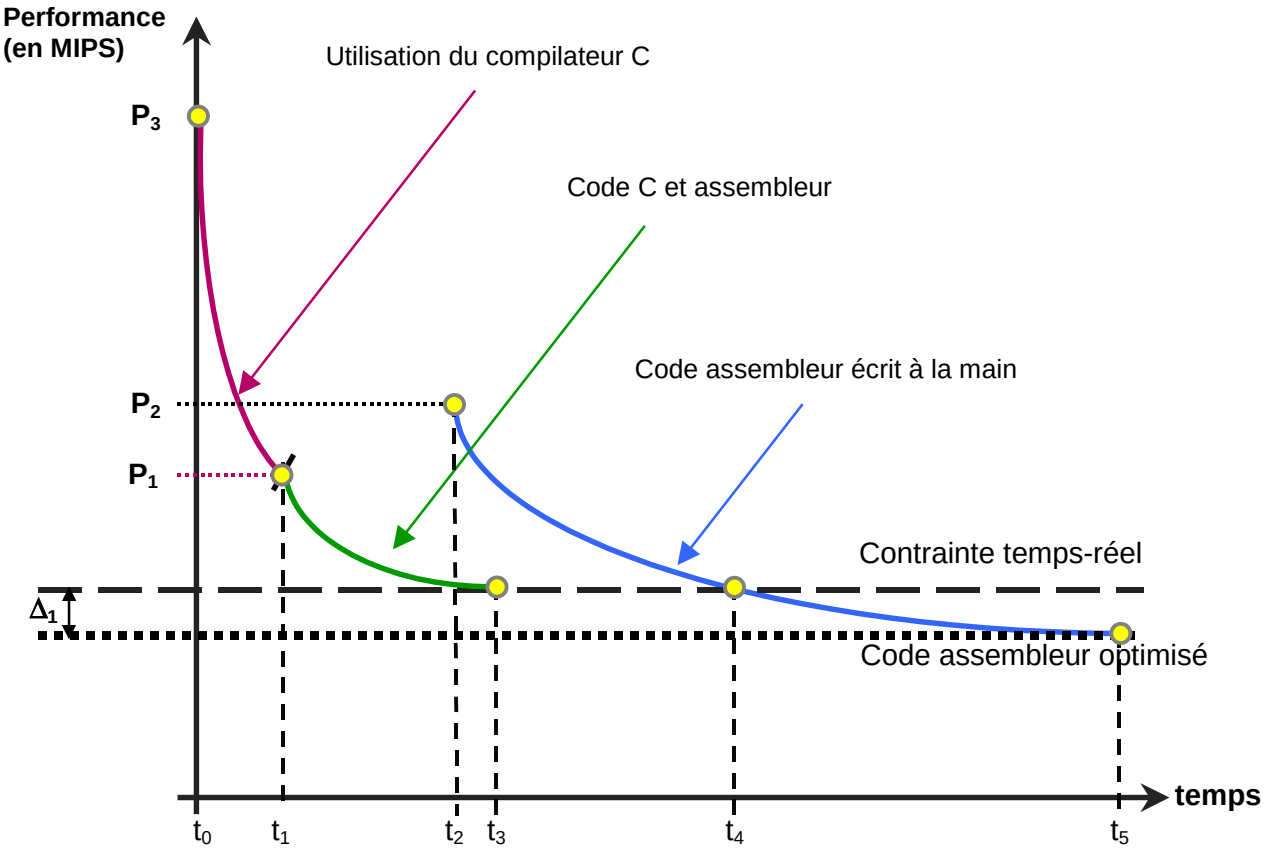


Figure 41 : Comparaison de deux méthodes de développement

L'utilisation d'un compilateur permet de réduire très significativement le *time-to-market* puisque seule une faible partie de l'application est codée manuellement.

Une méthode basée sur l'utilisation du compilateur C possède néanmoins des problèmes similaires à une méthode classique, à savoir la localisation des parties critiques. Sans outil spécifique, il est nécessaire de simuler le code assembleur généré par le compilateur et d'indiquer les adresses de début et de fin de chaque fonction dans le programme. Notre objectif est double :

- accélérer le processus d'estimation par une localisation rapide des parties critiques,
- guider le programmeur dans l'optimisation du code C et dans le choix des parties qu'il est nécessaire d'implémenter directement en assembleur.

C'est ce que réalise la méthode proposée et mise en œuvre dans l'outil d'estimation VESTIM développé au cours de cette étude pour les besoins de VLSI Technology.

4.3. Flot général de la méthode d'estimation logicielle

Le flot général de la méthode développée dans VESTIM est décrit sur la Figure 42.

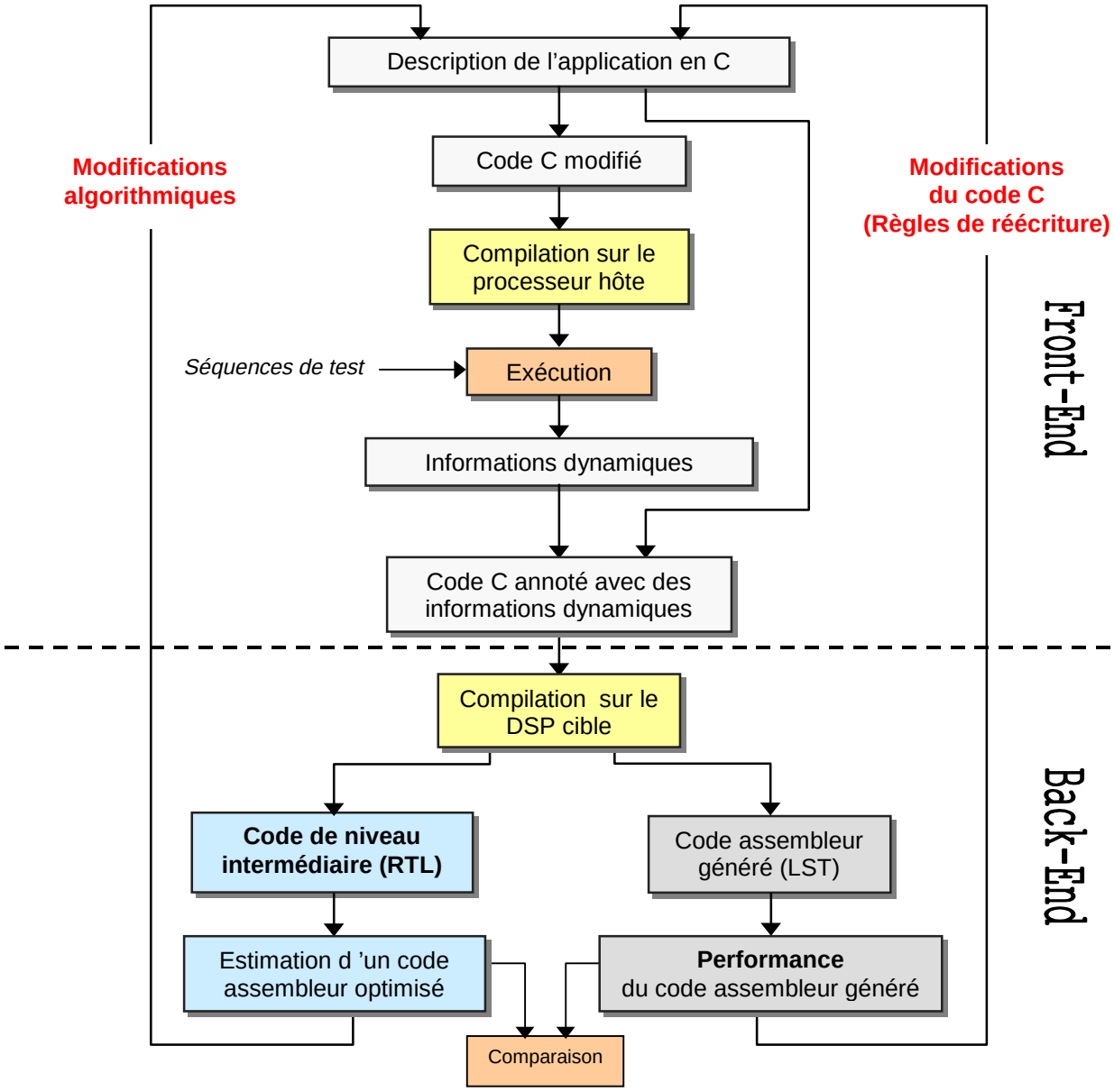


Figure 42 : Flot global de l'outil d'estimation

4.3.1. Méthode d'optimisation

L'utilisation de l'outil VESTIM s'effectue à partir d'une description en C de l'application. Lors d'une première itération (i.e. avec le code C d'origine), les résultats fournis permettent de localiser les parties critiques de l'application mais aussi de mesurer la différence entre les performances du code généré et l'estimation d'un code optimisé. Plus la différence est importante plus le code généré est de mauvaise qualité. Au contraire, si la différence est faible cela signifie que le compilateur a effectué un bon travail. En fonction de ces résultats et des contraintes temps-réel, l'utilisateur modifie son code C d'origine en concentrant son travail d'optimisation sur les parties critiques de l'application. L'estimation

d'un code assembleur optimisé permet également de guider l'utilisateur dans le choix des parties du code qu'il est indispensable de coder manuellement. Ce sont généralement les parties pour lesquelles, malgré de nombreuses tentatives d'optimisation du code C, la différence entre les performances du code généré et l'estimation d'un code optimisé reste importante. Notons que l'estimation permet de déterminer le gain en nombre de cycle d'un codage manuel (i.e. la différence entre les deux mesures). Le processus est itératif ce qui justifie le choix d'une méthode rapide évitant l'utilisation de simulateur de niveau instruction.

L'outil est composé de deux parties indépendantes : le « *front-end* » permet de déterminer les valeurs des x_i alors que le « *back-end* » fournit deux mesures pour les valeurs des c_i.

4.3.2. Le calcul des x_i

L'objectif de la phase initiale, ou « *front-end* », est de déterminer le nombre d'exécutions de chaque instruction C (en fait groupe d'instructions C) durant l'exécution de toute l'application. Ces informations, appelées **informations dynamiques** (les x_i), sont obtenues en utilisant une méthode statistique : le code C, une fois compilé, est exécuté avec une séquence de test choisie avec précaution[29]. Notons que le temps d'obtention des résultats (du code généré ou l'estimation d'un code optimisé) ne dépend quasiment pas de la durée de la séquence de test. Les séquences de test permettent également de valider les modifications apportées au code C. Les opérations de la phase initiale sont effectuées sur le processeur hôte (e.g. Pentium) ce qui permet un gain de temps considérable. En effet, la compilation et l'exécution d'applications complètes telles que G.728 [Rec94], G.721 [Rec84] ou AC-3 [Ac3] sur une station SPARC ou un Pentium ne prend que quelques secondes en opposition à une simulation de plusieurs heures voire plusieurs jours. Chaque instruction du code C est alors annotée par une information dynamique.

4.3.3. Le calcul de deux c_i différents

La phase finale, ou « *back-end* », débute par la compilation du code C (annoté d'informations dynamiques) avec le compilateur du DSP cible.

Le calcul des performances du code assembleur généré est effectué en utilisant directement les résultats de compilation. Nous additionnons le nombre de cycles des instructions en assembleur correspondant à chaque instruction C pour le calcul des c_i de chaque bloc de base. Une table construite à partir du manuel utilisateur décrivant le jeu d'instructions du DSP cible permet de déterminer un nombre de cycles pour chaque instruction assembleur. Les DSP possèdent une architecture pipeline qui reste assez simple et déterministe du fait qu'ils ne contiennent pas de cache et que les aléas dus aux dépendances ne provoquent que très peu de perturbation sur les performances : les opérations de recherche des opérandes, d'exécution et de rangement des résultats se font dans le même cycle. Ceci justifie l'utilisation d'une approche par table pour calculer les c_i.

L'estimation des performances d'un code assembleur optimisé s'effectue à partir d'un niveau de description intermédiaire (RTL) entre le C et l'assembleur. Le niveau RTL est le langage intermédiaire utilisé par le compilateur GNU-C. La suite de ce rapport est consacrée à la description détaillée de ce processus d'estimation.

[29] Généralement la séquence de test représente un nombre d'échantillons correspondant à un nombre entier de secondes ce qui permet d'obtenir aisément des résultats en MIPS.

Construction d'une représentation intermédiaire pour l'estimation

Pour faire de l'estimation de performance nous allons traduire le code C en une représentation intermédiaire adaptée à une classe d'architectures DSP cibles. A partir de cette représentation intermédiaire, le travail d'estimation consiste à évaluer au mieux les c_i associés à chaque instruction ou groupe d'instructions C. L'estimation de performance nécessite aussi de connaître les valeurs des informations dynamiques (x_i) qu'il s'agit de déterminer avant de traiter la représentation intermédiaire.

5.1. La récupération d'informations dynamiques

L'objectif est de récupérer le nombre d'exécutions de chaque instruction sans effectuer de simulation de niveau assembleur. Plusieurs outils sont disponibles sous Unix comme sous DOS pour récupérer des informations dynamiques lors de l'exécution du programme sur le processeur hôte. Nous présentons deux outils, « gprof » et « tcov », que nous avons expérimenté lors de la recherche d'une méthode permettant de récupérer des informations dynamiques.

5.1.1. L'outil « gprof »

Dans l'environnement GNU, un outil de *profiling* « gprof » donne, dans un ordre décroissant, le temps total d'exécution et le nombre d'appels de chaque fonction du programme lorsque ce dernier est exécuté sur le processeur hôte (e.g. SPARC 20).

Nom de la fonction	PineDSPCore	OakDSPCore	gprof
Decode	52,1	51,3	50,3
Encode	47,9	48,7	47,6
Fmult	39,3	36	10,4
Upd	6,4	8,6	3,4
Trans	3,8	4,4	0,5
Upa2	3,7	3,8	2,2
Floata	3,2	2,2	2,5
Floatb	3,2	2,2	1,5
Antilog	2,8	3,6	0,3
Log	2,8	2,2	1,2
Subtc	2	2,2	0,6
Quan	1,8	2	1,9
Upa1	1,6	1,7	1,2

Table 2 : Etude de performance à partir de l'outil *gprof*

La Table 2 montre les résultats d'une étude comparative de trois types de *profiling* pour l'application G.721 [Rec84]. Pour les principales fonctions de l'application, la seconde et troisième colonne représentent les pourcentages en temps d'exécution par rapport au temps global d'exécution obtenus en simulant un code assembleur optimisé manuellement, respectivement sur le PineDSPCore et le OakDSPCore. L'analyse des résultats montre que « gprof » permet d'extraire les fonctions critiques de l'application. Cependant, nous observons que pour la fonction « fmult » par exemple, il existe un rapport trois entre les temps d'exécution sur les DSP et sur la station UNIX utilisée. Ces mesures montrent qu'avec « gprof » il peut y avoir des écarts importants de performance par rapport au temps d'exécution d'une implémentation sur DSP (ce qui était prévisible). De plus, cet outil ne fournit des résultats qu'au niveau des fonctions ce qui ne correspond pas au niveau de granularité souhaité. En fait « gprof » fournit directement la somme des $x_i c_i$ relativement à

chaque fonction. Un autre outil mieux adapté, « tcov », permet de connaître le nombre d'exécutions de chaque instruction C du programme indépendamment du processeur cible.

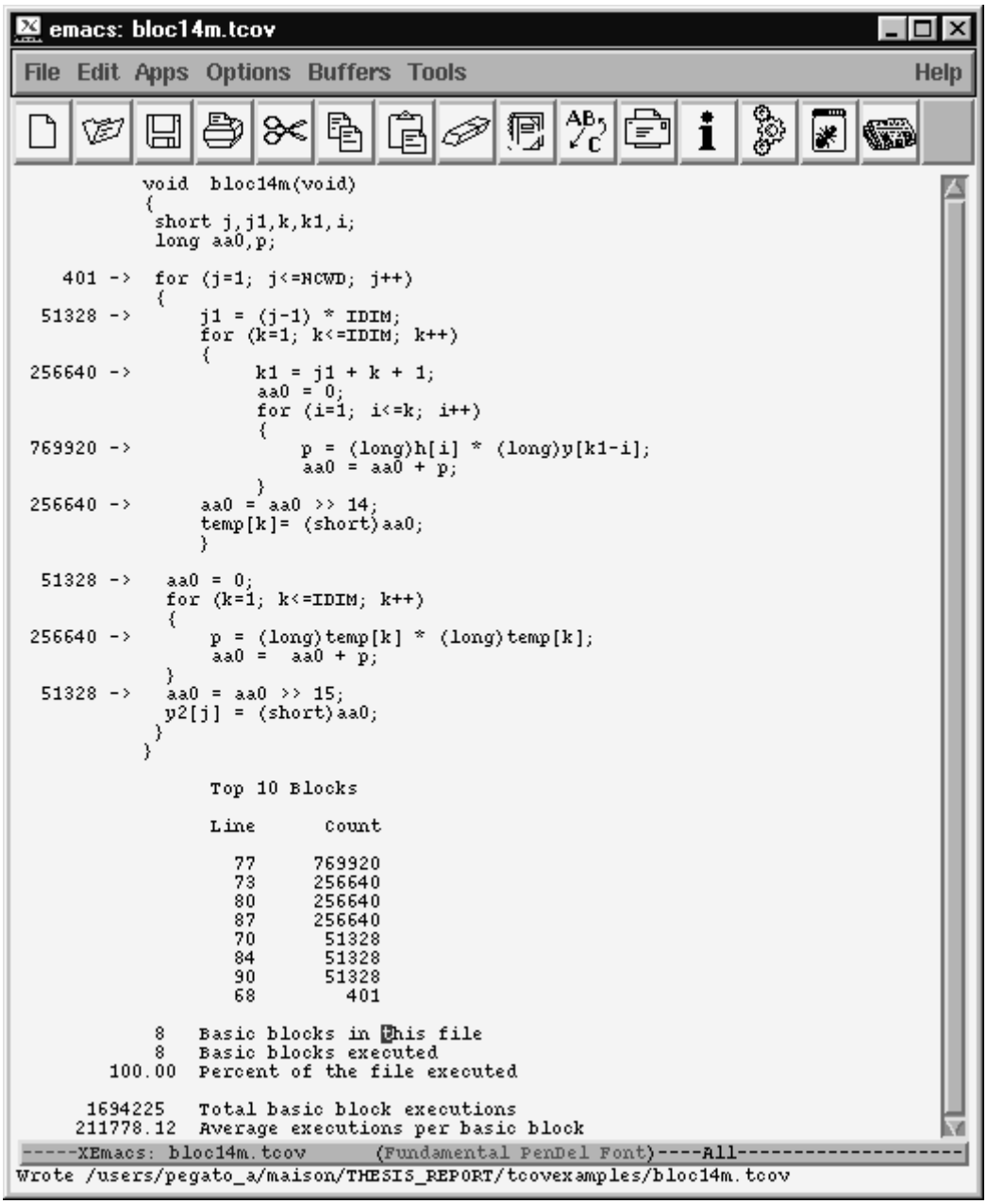


Figure 43 : Informations fournies par l'outil « tcov »

5.1.2. L'outil « tcov »

L'outil « tcov » permet de déterminer, à partir de l'exécution d'un code C sur le processeur hôte, le nombre d'exécutions de chaque instruction. La Figure 43 montre avec l'exemple du Bloc14 de la norme G.728 [Rec94], les informations fournies par cet outil pour une séquence de test représentant une seconde de temps-réel (signal de parole). L'outil calcule le nombre d'exécutions de chaque bloc de base[30] du programme pour la séquence de test exécutée et annote le fichier C avec ces informations. Sur cet exemple, pour une seconde de signal de parole le premier bloc de base est exécuté 401 fois, le second 51 328 fois, le troisième 256 640 fois, etc. On voit clairement que les instructions à l'intérieur de la troisième boucle imbriquée sont les plus fréquentes puisqu'elles sont exécutées 769 920 fois par seconde. Une autre information intéressante fournie par « tcov » est le taux de couverture du fichier exécuté (*percent of the file executed*). Ceci permet d'avoir une idée de la qualité de la séquence de test choisie. Sur cet exemple, toutes les instructions sont exécutées au moins une fois (taux de 100%), ce qui n'est cependant pas forcément représentatif du pire cas.

L'utilisation de cet outil pose un problème purement industriel : c'est un outil propriétaire (Sun Microsystems) et uniquement disponible sur une plate-forme UNIX. Or, les contraintes imposées par VLSI Technology sont d'avoir un outil d'estimation portable sur différentes plateformes. C'est pour cette raison que nous nous sommes orientés vers l'utilisation de logiciels issus du projet GNU.

5.1.3. La récupération d'informations dynamiques à l'aide des outils GNU

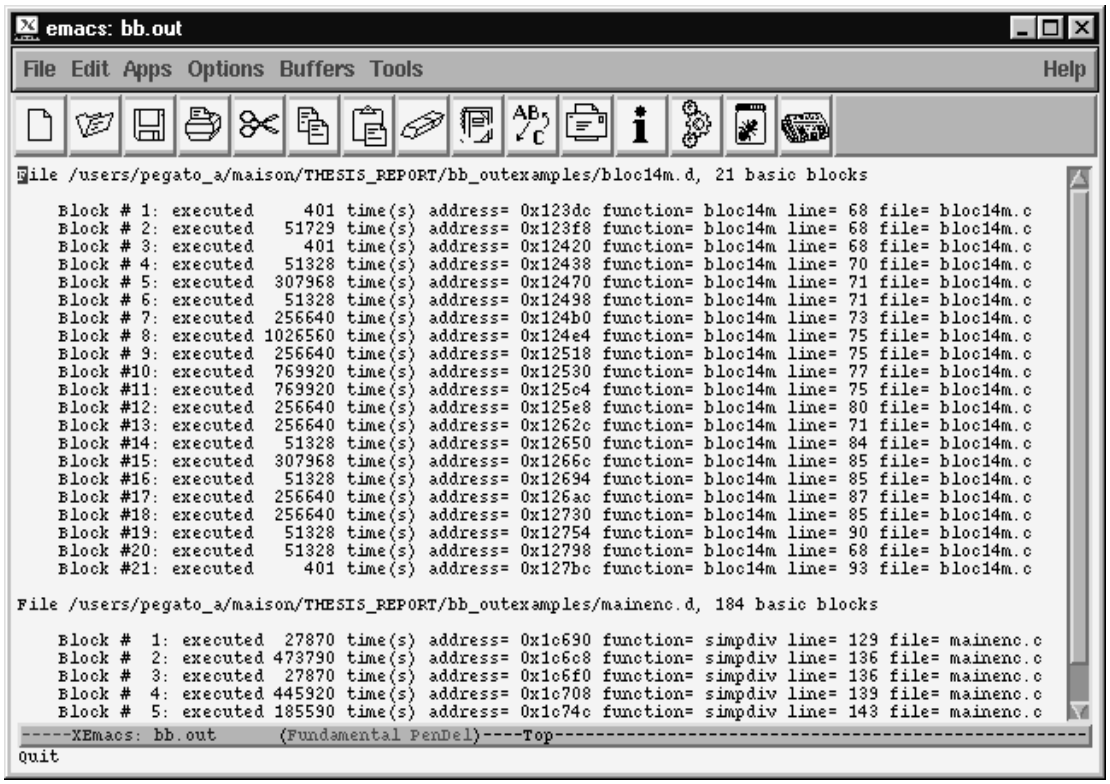


Figure 44 : Exemple de fichiers contenant des informations dynamiques

[30] Ou chaque instruction C

Grâce à des options de compilation spécifiques du compilateur GNU-C, il est possible après exécution d'un programme C de récupérer dans un fichier des informations dynamiques identiques à celles fournies par « tcov ». La Figure 44 montre le fichier « bb.out » ainsi obtenu. Pour chaque fonction de l'application, le fichier « bb.out » contient le nombre d'exécutions de chaque instruction C. On remarque que ce fichier contient plus d'informations par rapport aux résultats fournis par « tcov ». Nous ne détaillons pas ces différences mais il faut noter que ces informations supplémentaires sont parfois utiles pour effectuer des estimations plus précises. Notons enfin qu'il est possible de calculer le taux de couverture du programme exécuté pour la séquence de test choisie.

Figure 45 : Programme C annoté d'informations dynamiques

Grâce aux informations de numéro de ligne que contient le fichier « bb.out », il est aisé d'annoter le code C des informations dynamiques. On utilise pour cela une syntaxe

particulière du langage C, le « #pragma », ce qui permet de retrouver ces annotations au niveau des langages intermédiaire (RTL) et assembleur, caractéristique indispensable pour faire des estimations. La Figure 45 montre le programme C annoté équivalent à celui obtenu lors de l'utilisation de l'outil « tcov » (Figure 43).

5.2. L'utilisation d'un niveau de description intermédiaire

5.2.1. Raison de ce choix

Le choix du niveau de description du programme pour effectuer des estimations de performance est une question abordée dans [Mal95]. Dans [Par89] les auteurs affectent à chaque type d'instruction C (condition, boucle, affectation, etc.) un coût représentant le nombre de cycles (c_i) nécessaire à son exécution sur le processeur cible. Cette approche a pour principal inconvénient de ne pas prendre en compte les optimisations possibles du compilateur. De plus il semble difficile à partir du langage C d'obtenir des performances tirant parti de toutes les caractéristiques architecturales du processeur. Pour éviter ces inconvénients, dans [Mal95][Ern95][Lee96] ou [Ott97], les auteurs effectuent des estimations à partir du code assembleur généré par le compilateur. Malheureusement, dans le cas des processeurs DSP, ces mesures représentent plus une mesure de l'inefficacité du compilateur.

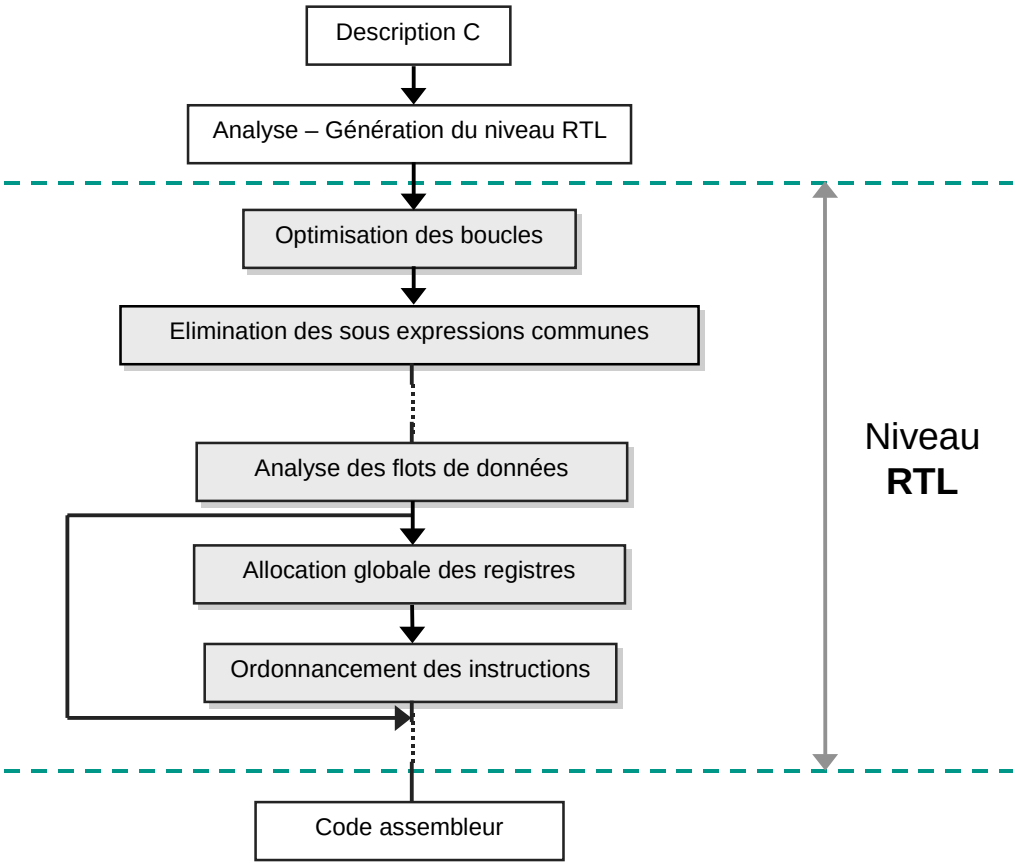


Figure 46 : Représentation simplifiée du niveau RTL

Comme notre objectif est d'estimer les performances d'un code assembleur optimisé, notre choix s'est porté sur un niveau de description intermédiaire afin de se rapprocher du processeur cible tout en décrivant l'application à partir d'un langage de haut niveau, en l'occurrence le C. Il nous paraît en effet difficile d'identifier dans le code assembleur généré

par le compilateur, les parties de code responsables du taux d'expansion par rapport à un code écrit manuellement. Ceci nous semble d'autant plus délicat pour des architectures avec beaucoup de parallélisme (e.g. le PalmDSPCore).

5.2.2. Description du niveau RTL

Les compilateurs C du OakDSPCore et du PalmDSPCore sont basés sur un compilateur du domaine public, le GNU-CC. Ce dernier génère entre le C et l'assembleur un langage intermédiaire, appelé RTL, pour niveau transfert de registre, qui constitue en quelque sorte un langage assembleur virtuel. La Figure 46 montre (de manière simplifiée) que les traitements effectués au niveau RTL sont constitués d'un ensemble de passes (une vingtaine en réalité). Ces passes constituent les phases classiques d'optimisation en compilation [Aho86] comme l'élimination de sous-expressions communes, l'optimisation des boucles ou l'ordonnancement des instructions. Des modifications spécifiques aux DSP cibles ont été effectuées sur certaines passes par la société DSP Group qui diffuse ce compilateur afin d'optimiser les traitements. Ces extensions restent néanmoins valables pour une classe de DSP (de type Harvard modifié).

5.2.3. Choix d'une passe pour effectuer des estimations

De nombreuses expérimentations nous ont permis de déterminer les principales causes d'inefficacité du compilateur C pour le OakDSPCore. Ces tests ont été effectués sur des algorithmes de base de traitement du signal ou des fonctions extraites dans diverses applications :

- FFT
- Produit de vecteurs
- Addition de vecteurs
- Machine d'état fini
- Filtre à réponse impulsionnelle finie (FIR)
- Multiplication de deux matrices
- Différents blocs de la norme G.721 [Rec84]
- Différents blocs de la norme G.728 [Rec94]

Les résultats montrent que la sauvegarde temporaire de données en mémoire (*spilling*) est la principale cause du taux d'expansion entre le code généré par le compilateur et un code écrit manuellement (30% du taux d'expansion en moyenne). Ce sont les phases d'allocation de registres qui ne sont pas capables de prendre en compte efficacement le jeu de registres restreint et hétérogène du OakDSPCore. Aussi, avant les phases d'allocation de registres, les passes de niveau RTL opèrent avec un ensemble de registres virtuels en nombre très important. Pour ces passes, il n'existe donc pas de problème de *spilling*. Or, en questionnant plusieurs programmeurs expérimentés du OakDSPCore, il s'avère que les opérations de *spilling* sont extrêmement rares lors d'un codage manuel, y compris dans les parties critiques de l'application. Sur la base de cette information, nous faisons l'hypothèse qu'un code assembleur optimisé ne comporte pas d'opérations de *spilling*. Nous admettons que c'est une hypothèse forte, mais nous verrons avec les résultats qu'elle est tout à fait justifiée. De plus, si des opérations de *spilling* apparaissent en trop grand nombre dans un code optimisé, on peut en déduire que l'architecture du DSP n'est pas adaptée et qu'il est préférable d'en choisir une autre.

Partant de cette hypothèse, pour effectuer des estimations le choix de la passe de niveau RTL s'est donc porté sur une des passes précédant les phases d'allocation de registres. L'analyse des différentes passes du niveau RTL a montré néanmoins l'intérêt de profiter des optimisations classiques de compilation [Aho86], comme des optimisations ciblées DSP apportées par DSP Group. Nous avons donc choisi de faire des estimations à partir d'une description RTL se situant avant les phases d'allocation de registres et après la passe d'analyse des flots de données[31].

La passe « flow » choisie comporte de nombreux intérêts car un certain nombre de spécificités de l'architecture sont prises en compte :

- mise à jour des informations sur les instructions RTL en particulier pour les registres utilisés,
- mise à jour des dépendances entre instructions appartenant à un même bloc de base,
- notion de mode d'adressage,
- suppression de nombreuses instructions inutiles,
- utilisation d'instructions spécifiques au DSP comme les instructions de boucles matérielles (*Bkrep*),

Enfin, il est aussi important de noter que le niveau RTL permet de représenter le parallélisme inter-instruction potentiel de l'application (composantes connexes au sein d'un bloc de base). Ceci est essentiel pour étendre la méthode à des DSP disposant d'instructions parallèles.

[31] On l'appelle passe "flow"

5.2.4. Représentation graphique du programme

5.2.4.1. La construction du CFG

La passe de niveau RTL « flow » contient toutes les informations nécessaires à la construction du CFG de chaque fonction de l'application. La Figure 47 montre le CFG correspondant au calcul d'un papillon de FFT construit à partir de la description RTL. Le graphe contient sept blocs de base et trois niveaux de boucles imbriquées. A chaque bloc de base on associe l'information dynamique x_i collectée lors de l'exécution du code C avec une séquence de test. Le bloc de base B_4 est par exemple exécuté 80 fois.

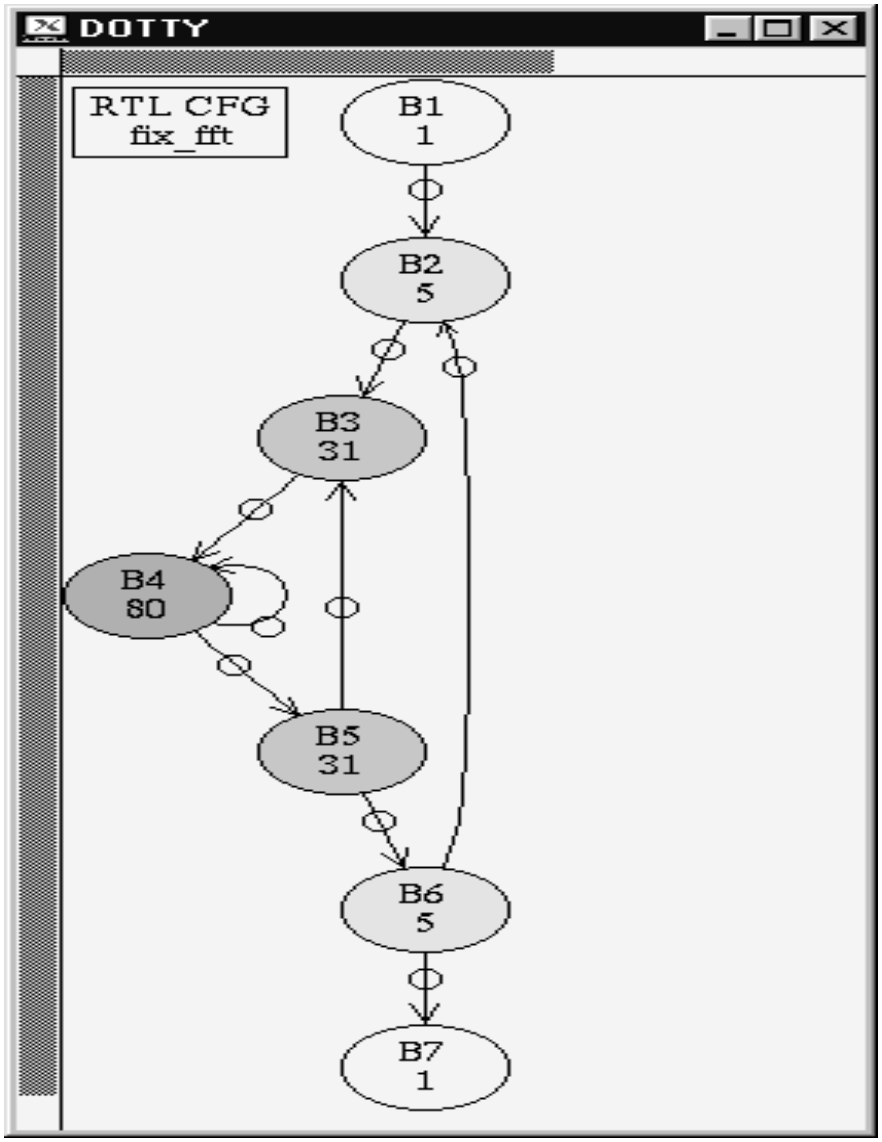


Figure 47 : CFG pour la transformée de Fourier rapide

5.2.4.2. La construction des DAG

Pour estimer le nombre de cycles c_i associé à chaque bloc de base, nous représentons ces derniers sous la forme de graphes orientés flots de données acycliques (DAG) à partir de la description de niveau RTL. La Figure 48 montre un exemple de code de niveau RTL correspondant à l'instruction C grisée (instruction « __bkrep__ for »), ce qui correspond au

bloc de base B_2 du CFG de la Figure 47. Le DAG correspondant est composé de six nœuds et deux composantes connexes comme le montre la Figure 49. Les nœuds « Set_Reg » désignent une affectation dans un registre alors que le nœud « Bkrep » décrit une instruction de boucle matérielle. Nous ne détaillons pas dans ce rapport la syntaxe du niveau RTL[32]. Notons cependant que la description de niveau RTL est très verbeuse et que beaucoup d'informations inutiles ne sont pas prises en compte lors de la construction du DAG.

```
(note 91 90 92 ("#pragma 5 , 36 , 5") NOTE_ILIST)

(note 92 91 94 ("__bkrep__ for (j=0; j<nb_group; j++) ") NOTE_ILIST)

(insn 407 94 408 (set (reg:QI 256)
        (reg/v:QI 216)) 29 {match_movqi} (nil)
    (nil))

(insn 408 407 96 (set (reg:QI 256)
        (minus:QI (reg:QI 256)
            (const_int 1))) 107 {subqi3} (insn_list 407 (nil))
    (nil))

(insn 409 96 97 (unspec[
            (use (reg:QI 256))
            (label_ref 412)
            (const_int 2)
        ]  1) 297 {bkrep_desc} (insn_list 408 (nil))
    (expr_list:REG_DEAD (reg:QI 256)
        (nil)))

(code_label 105 98 106 108 "")
```

Figure 48 : Exemple de code de niveau RTL

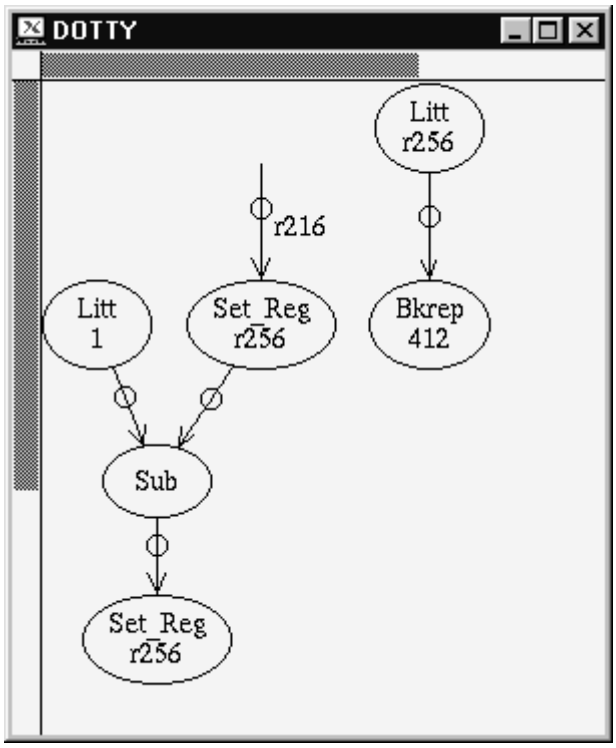


Figure 49 : Exemple de DAG

[32] Une description détaillée se trouve dans [Sta94].

5.2.4.3. Quelques définitions de syntaxe

Lors de la construction du DAG, nous avons, pour certaines instructions RTL, utilisé une syntaxe particulière.

- ***Litt*** : désigne un nœud manipulant une constante entière ou flottante (éventuellement rangée dans un registre).
- ***Addr*** : désigne la lecture à une adresse.
- ***VAL*** : désigne un nœud effectuant une opération de lecture en mémoire.
- ***ASSIGN*** : désigne un nœud effectuant une opération d'écriture en mémoire.

Le niveau RTL contient des informations qui permettent de déterminer les modes d'adressage du OakDSPCore ou du PalmDSPCore. Les nœuds VAL et ASSIGN peuvent ainsi prendre les formes suivantes :

- ***VALi*** ou ***ASSIGNi*** : adressage indirect
- ***VALix*** ou ***ASSIGNix*** : adressage indexé
- ***VALsd*** ou ***ASSIGNsd*** : adressage direct court
- ***VALld*** ou ***ASSIGNld*** : adressage direct long

Pour les opérations de lecture en mémoire (nœuds VAL), nous ajoutons une information sur le type d'accès réalisé en fonction du nœud successeur du nœud « VAL ».

- ***Direct*** : la donnée est lue sans mémorisation dans un registre (le nœud suivant peut lire la donnée directement en mémoire sans passer par un registre).
- ***Reg*** : la donnée lue en mémoire est obligatoirement rangée dans un registre.

Les opérations d'écriture du OakDSPCore sont toujours de type « Reg », ce qui n'est pas toujours le cas pour le PalmDSPCore. Nous indiquons enfin (dans le cas où l'information est présente) si l'accès mémoire est effectué en mémoire XRAM ou YRAM. Voici quelques exemples :

- ***VALi_Direct_Y*** : indique la lecture directe d'une donnée en YRAM via un mode d'adressage indirect.
- ***VALsd_Reg*** : désigne une lecture en mémoire XRAM ou YRAM (on choisira l'une ou l'autre lors de l'ordonnancement) par un mode d'adressage direct court. La donnée lue est forcément stockée dans un registre.
- ***ASSIGNix_X*** : indique une opération d'écriture en mémoire XRAM via un mode d'adressage indexé.

5.3. Un niveau de description intermédiaire orienté DSP

5.3.1. Expérimentations

L'estimation de performance d'un code assembleur optimisé ne peut être effectuée directement à partir du niveau RTL. En effet, si en utilisant la passe « flow » nous éliminons le problème du *spilling*, une partie de l'inefficacité du compilateur DSP se retrouve aussi au niveau RTL, comme illustrée par la Table 3 pour différents algorithmes de base de traitement du signal et quelques fonctions représentatives de l'application G.728.

	Code assembleur optimisé (**ASM**)	Code généré par le compilateur (**LST**)	Estimation de niveau **RTL**
FIR (ordre N)[33]	N+3	N+3	N+3
Somme de vecteurs[33]	2+3N	2+3N	2+3N
Produit de vecteurs[33]	2+3N	2+3N	2+3N
Machine d'état fini[33]	293	340 (+16%)	353 (+20%)
Papillon de FFT[33]	16	43 (268%)	32 (200%)
Bloc17 de G.728[34]	7,05	17,86 (253%)	11,96 (+56%)
Bloc14 de G.728[34]	3,14	6,27 (+99%)	5,75 (+83%)
Hwmcore de G.728[34]	2,56	3,75 (+46%)	3,8 (+48%)
Bloc50 de G.728[34]	4,71	33,86 (719%)	23,6 (501%)

Table 3 : Tableau comparatif de performance pour divers algorithmes

La première colonne représente pour chacun des algorithmes les performances d'un code assembleur optimisé (code ASM) écrit par un programmeur expérimenté. La seconde colonne indique les performances obtenues en utilisant le compilateur C du OakDSPCore (code LST), et la dernière colonne montre les résultats estimés à partir d'un code de niveau RTL (code RTL).

Les résultats de niveau RTL ont été obtenus à partir d'une analyse entièrement manuelle. Pour chaque fonction, on construit à partir du RTL obtenu après la passe « flow » le CFG ainsi que l'ensemble des DAG pour chacun des blocs de base. L'estimation du nombre de cycles pour chaque bloc de base s'effectue alors à partir du DAG correspondant en utilisant le manuel utilisateur décrivant le jeu d'instructions du processeur. Cette étape a nécessité un investissement en temps important mais a permis de déterminer les principales causes d'inefficacité du code RTL par rapport à un code assembleur optimisé. Nous détaillons dans le paragraphe suivant les résultats obtenus pour les différents algorithmes.

5.3.2. Analyse des résultats

La Table 3 montre que pour des algorithmes simples (FIR, somme ou produit de vecteurs) le compilateur C du OakDSPCore est capable de générer un code LST ayant les mêmes performances qu'un code assembleur optimisé. On remarque, sur ces trois algorithmes, que le code RTL est lui aussi optimisé.

[33] En nombre de cycles
[34] En nombre de MIPS

Le programme « machine d'états finis » est un algorithme intéressant pour tester l'efficacité du compilateur sur les structures de contrôle. L'augmentation de 16% des performances du code LST par rapport au code ASM est dû principalement à l'absence d'utilisation des instructions conditionnelles. Le code RTL possède les mêmes défauts que le code LST avec une instruction conditionnelle de plus non générée. Cette fonction montre néanmoins que le OakDSPCore est assez efficace pour ce type de traitement.

5.3.2.1. Le papillon de FFT

Analyse de la différence entre le code LST et le code ASM

La différence entre le code LST et le code ASM est de 27 cycles. Une étude détaillée du code généré par le compilateur et du code assembleur optimisé a permis de retrouver cette différence.

- +2 cycles : Lecture en mémoire non optimisée.

A deux reprises, le compilateur génère une instruction supplémentaire pour la lecture d'une donnée en mémoire qui doit être placée dans la partie haute d'un accumulateur.

Code généré	Code optimisé
mov (r1),b0 mov b0h,a1h	mov (r1),a1h

- +5 cycles : Gestion du registre PS à l'intérieur d'une boucle.

Selon la valeur du registre PS, il est possible de décaler automatiquement le résultat de la multiplication de un bit vers la droite ou un ou deux bits vers la gauche avant son utilisation par l'ALU. Lors d'un codage manuel, ce registre est toujours initialisé à l'extérieur d'une boucle.

- +3 cycles : Accès mémoire non optimisé lors d'une opération de multiplication.

Le compilateur n'est pas en mesure de produire un code qui permet de lire deux données simultanément en mémoire XRAM et YRAM.

Code généré	Code optimisé
mov (r5),y mac y,(r0)-,a1	mac (r5),(r0)-,a1

- +2 cycles : Instruction MAC non détectée (décomposée).

Le code assembleur optimisé comporte quatre instructions MAC alors que le code généré n'en comporte que deux. Le compilateur n'a pas sorti de la boucle la première multiplication, ce qui est nécessaire pour amorcer de manière optimale le pipeline à l'intérieur de la boucle.

- +6 cycles : Gestion du mode d'adressage modulo.

Le compilateur n'est pas en mesure (à partir de la description en C fournie) de mettre à jour de manière optimale les registres d'adresse pour la gestion d'un tableau circulaire.

- +9 cycles : Sauvegarde temporaire de données en mémoire (*spilling*).

Le code assembleur optimisé du papillon de FFT ne comporte aucune opération de *spilling*. La complexité de l'algorithme (le nombre de calculs effectués et les nombreux accès à la mémoire) fait que le compilateur génère neuf instructions de ce type.

Analyse de la différence entre le code LST et le code RTL

La différence de 11 cycles entre les deux codes est due en grande partie au problème de *spilling* que ne connaît pas le niveau RTL (-9 cycles). Le code RTL est également moins pénalisé lors de la mise à jour des registres pour l'adressage du tableau circulaire (-2 cycles) et par les problèmes de pipeline (-1 cycle). Par contre, le code RTL comporte une instruction de plus que le code LST pour la gestion du registre PS (+1 cycle).

Analyse de la différence entre le code ASM et le code RTL

La différence de 16 cycles entre ces deux codes est décomposée de la manière suivante :

- +6 (37,5%) : Gestion du registre PS
- +3 (19%) : Lecture en mémoire non optimisée
- +1 (6%) : Instruction MAC non détectée
- +6 (37,5%) : Gestion d'un tableau circulaire (mise à jour des registres d'adresse par un mode d'adressage modulo)

Les pourcentages indiqués entre parenthèses désignent le rapport entre la cause d'inefficacité considérée et le nombre total de cycles supplémentaires.

Conclusion

Le papillon de FFT est un algorithme intéressant puisqu'il révèle un certain nombre de problèmes très fréquemment rencontrés lors d'un codage sur DSP : nombreux calculs de type MAC, mode d'adressage modulo et nombreux accès mémoires. Dans la suite, nous nous intéresserons uniquement à la différence de performance entre un code assembleur optimisé (ASM) et le code RTL. En effet, le code LST comporte généralement les mêmes instructions supplémentaires que le niveau RTL avec les problèmes de *spilling* en plus.

5.3.2.2. Le Bloc17 de G.728

L'analyse des performances des fonctions *Bloc17*, *Bloc14*, *Hwmcore* et *Bloc50* de l'application G.728 est plus complexe que celle de la FFT. En effet, ces fonctions sont constituées de plusieurs blocs de base. Il est donc nécessaire de faire une étude des DAG bloc de base par bloc de base. La Table 4 montre le résultat de cette analyse pour le *Bloc17* composé de 70 lignes de code C environ. Cet exemple est un cas intéressant puisque la structure (i.e. le CFG) du programme diffère entre le code de niveau RTL et le code assembleur écrit manuellement du fait des optimisations du compilateur C. Il est alors parfois impossible de trouver une correspondance entre les blocs de base du code ASM et des codes LST et RTL. Quand cela est le cas, nous avons annoté par un tiret (-) les blocs de base concernés du code ASM. De plus, pour certains blocs de base, le nombre d'exécutions est légèrement différent (bloc de base B_3, B_5 et B_{16}). Ceci montre qu'il n'est pas possible pour cet exemple de comparer directement tous les blocs de base entre eux.

La méthode que nous avons suivie s'est donc ramenée à localiser dans le code RTL les traitements (i.e. les instructions) qui n'apparaissent pas lors d'un codage manuel. Pour le *Bloc17* de G.728, ces traitements sont les suivants :

- Gestion non optimisée des registres d'adresse dans une boucle
  1,2 MIPS (24 %)
- Instruction conditionnelle non retrouvée ou non optimisée
  2,2 MIPS (44 %)
- Décalage du registre P après une multiplication (lié au problème de gestion du registre PS à l'intérieur d'une boucle)
  1 MIPS (20 %)
- Affectations inutiles de registre
  0,6 MIPS (12 %)

	Nombre d'exécutions[35]	**Code Optimisé (ASM)**	**Code généré (LST)**	**Code intermédiaire (RTL)**
B_1	1600	14	26	22
B_2	204800	2	8	7
B_3	1024000	1 (819200[36])	1	1
B_4	204800	2	2	2
B_5	102893	1 (204800[36])	1	1
B_6	204800	11	7	5
B_7	614400	-	7	5
B_8	91006	-	2	1
B_9	614400	-	1	1
B_{10}	204800	14	6	5
B_{11}	0	-	2	2
B_{12}	204800	-	28	10
B_{13}	6930	-	8	3
B_{14}	204800	-	5	3
B_{15}	1600	18	19	10
B_{16}	8000	1 (6400[36])	1	1
B_{17}	1600	3	4	2
B_{18}	836	2	4	4
B_{19}	1600	8	12	8
TOTAL		**7,05 MIPS**	**17,86 MIPS**	**11,96 MIPS**
Taux d'expansion[37]			**253%**	**(+70%)**

Table 4 : Etude du bloc17 de G.728

On remarque que ces différences apparaissent de manière diffuse dans l'ensemble des blocs qui composent le Bloc17.

5.3.2.3. Le Bloc14 de G.728

La Table 5 montre les résultats pour le Bloc14 de G.728 composé d'environ 25 lignes de code C. Pour cette fonction, les codes LST, RTL et ASM possèdent les mêmes structures de contrôle ce qui facilite la recherche des causes d'inefficacité.

[35] Nombre d'itérations pour le code LST et RTL
[36] Nombre d'itérations pour le code ASM
[37] Par rapport au code assembleur optimisé

- Gestion non optimisée des registres d'adresse dans une boucle
  1,8 MIPS (70 %)
- Décalage du registre P après une multiplication
  0,78 MIPS (30 %)

	Nombre d'exécutions	Code Optimisé (ASM)	Code généré (LST)	Code intermédiaire (RTL)
B_1	401	11	13	6
B_2	51328	3	7	4
B_3	256640	2	11	11
B_4	769940	1	1	2
B_5	256640	5	8	4
B_6	51328	8	5	3
B_7	401	6	7	4
TOTAL		**3,14 MIPS**	**6,27 MIPS**	**5,75 MIPS**
Taux d'expansion[37]			**(+99%)**	**(+83%)**

Table 5 : Etude du bloc14 de G.728

La comparaison des performances du code généré (LST) et du code assembleur optimisé (ASM) montre que l'inefficacité du compilateur se concentre sur les blocs de base B_3 et B_5. Ce sont les parties de code entourant la boucle la plus imbriquée du programme (9 cycles de plus pour le préambule et 3 cycles de plus pour le postambule). Pour le bloc B_4 le compilateur génère le même nombre d'instructions ce qui n'est pas le cas pour le niveau RTL.

5.3.2.4. La fonction Hwmcore de G.728

Le code assembleur optimisé possède une structure complètement différente du code LST ou RTL obtenu par compilation, ce qui ne permet pas une comparaison bloc de base par bloc de base (Table 6).

Comme pour le Bloc17, la méthode suivie consiste à localiser dans le code RTL les traitements (i.e. les instructions) qui ne sont généralement pas effectués lors d'un codage manuel. Pour la fonction *Hwmcore* ces traitements sont les suivants :

- Gestion non optimisée des registres d'adresse dans une boucle
  0,1 MIPS (8 %)
- Décalage du registre P après une multiplication
  0,95 MIPS (78 %)
- Décalage inutile de 16 bits d'une donnée lue en mémoire
  0,03 MIPS (2 %)
- Affectations inutiles de registre
  0,14 MIPS (12 %)

Le code de niveau intermédiaire (RTL) et le code généré (LST) ont des performances très similaires. L'analyse du code LST montre en effet que ce dernier ne comporte pratiquement pas d'instructions de *spilling*.

	Nombre d'exécutions	Code optimisé (ASM)	Code généré (LST)	Code intermédiaire (RTL)
B_1	1200	-	13	18
B_2	17600	-	1	1
B_3	1200	-	9	4
B_4	823	-	12	4
B_5	377	-	5	3
B_6	0	-	10	2
B_7	1200	-	61	34
B_8	823	-	2	1
B_9	1200	-	18	11
B_{10}	28000	-	10	7
B_{11}	496000	-	1	1
B_{12}	28000	-	29	18
B_{13}	1200	-	13	7
B_{14}	34000	-	1	1
B_{15}	1200	-	9	4
B_{16}	822	-	11	4
B_{17}	378	-	5	3
B_{18}	0	-	10	2
B_{19}	1200	-	76	40
B_{20}	28000	-	12	7
B_{21}	900000	-	1	2
B_{22}	28000	-	21	14
B_{23}	1200	-	7	4
TOTAL		**2,567 MIPS**	**3,75 MIPS**	**3,8 MIPS**
Taux d'expansion[37]			**(+46%)**	**(+48%)**

Table 6 : Etude de la fonction Hwmcore de G.728

5.3.2.5. Le Bloc50 de G.728

Pour le Bloc50 composé d'environ 160 lignes de code C, il est possible d'étudier les performances bloc de base par bloc de base des codes ASM, LST et RTL (Table 7). L'analyse détaillée des blocs de base du code assembleur optimisé avec le code de niveau RTL (étude des DAG) a permis de retrouver les raisons de l'écart de performance de 19 MIPS observé:

- Gestion non optimisée des registres d'adresses dans une boucle
  0,21 MIPS (1 %)
- Décalage du registre P après une multiplication
  1 MIPS (5 %)
- Décalage inutile de 16 bits d'une donnée lue en mémoire
  0,52 MIPS (3 %)
- Affectations inutiles de registre
  5,27 MIPS (27 %)
- Instruction conditionnelle non détectée ou non optimisée
  1,1 MIPS (6 %)

- Instruction « rnd » non détectée
  1,15 MIPS (6 %)
- Test de dépassement de capacité (*overflow*) nécessitant des opérations flottantes
  10 MIPS (52 %)

	Nombre d'exécutions	Code optimisé (ASM)	Code généré (LST)	Code intermédiaire (RTL)
B_1	*400*	7	27	11
B_2	*2*	0	4	3
B_3	*398*	4	8	9
B_4	*0*	1	1	1
B_5	*398*	6	19	21
B_6	*398*	1	3	4
B_7	*398*	21	31	28
B_8	*19502*	9	16	10
B_9	*487550*	1	5	2
B_{10}	*19502*	14	28	17
B_{11}	*0*	2	0	3
B_{12}	*19502*	7	16	15
B_{13}	*4295*	1	4	1
B_{14}	*19502*	6	12	8
B_{15}	*0*	2	0	3
B_{16}	*19502*	16	22	21
B_{17}	*248750*	4	37	34
B_{18}	*223*	6	10	8
B_{19}	*248*	2	3	3
B_{20}	*223*	5	10	6
B_{21}	*248527*	0	10	2
B_{22}	*248750*	4	53	34
B_{23}	*0*	6	10	8
B_{24}	*0*	2	3	3
B_{25}	*0*	6	22	8
B_{26}	*248750*	0	6	2
B_{27}	*248750*	4	10	12
B_{28}	*19502*	8	21	13
B_{29}	*4295*	1	3	1
B_{30}	*19502*	2	13	4
B_{31}	*400*	0	4	3
B_{32}	*398*	4	10	7
B_{33}	*398*	2	3	3
B_{34}	*2*	4	7	6
B_{35}	*2*	0	2	2
B_{36}	*2*	0	0	1
TOTAL		**4,71 MIPS**	**33,86 MIPS**	**23,6 MIPS**
Taux d'expansion[37]			**(710%)**	**(501%)**

Table 7 : Etude du bloc50 de G.728

Le Bloc50 met en évidence une cause essentielle de l'inefficacité des compilateurs DSP, à savoir la difficulté voire l'impossibilité de générer des instructions spécifiques. On retrouve ce problème au niveau RTL avec tout d'abord l'instruction d'arrondi « rnd » qui n'est pas détectée. Deuxièmement, le problème du test de dépassement de capacité d'une

donnée sur 32 bits est très coûteux, tant au niveau RTL que LST. En effet, tester en C un dépassement de capacité sur 32 bits consiste à passer les données dans un format flottant, ce qui est très coûteux en cycles (conversion des données entières en flottantes, calculs flottants puis conversion des données flottantes en format entier). La Table 7 montre enfin que certains blocs de base, B_4 par exemple, ne sont jamais exécutés (nombre d'exécutions égal à 0). Ceci est dû à la séquence de test utilisée qui ne représente qu'une seconde de temps-réel et qui ne permet donc pas d'exécuter tous les blocs de base de la fonction.

5.3.3. Vers une représentation orientée schéma de calcul DSP

De l'étude que nous avons mené, nous tirons quatre principales raisons de l'écart entre le code RTL et un code assembleur optimisé. La Table 8 montre une grande diversité de pourcentages entre applications pour une même cause. Par exemple, la gestion du registre P représente 5% du taux d'expansion pour le Bloc50, alors que ce taux est de 78% pour la fonction Hwmcore.

	FFT	Bloc17	Bloc14	Hwmcore	Bloc50
Instructions spécifiques (e.g. conditionnelles, test de dépassement de capacité)	25%	44%	-	2%	67%
Gestion des registres d'adresse (tableau circulaire par exemple)	37,5%	24%	70%	8%	1%
Gestion du registre P	37,5%	20%	30%	78%	5%
Affectation inutile de registre	-	12%	-	12%	27%

Table 8 : Synthèse de l'analyse des différences entre un code RTL et ASM

De manière encore plus générale on peut dire qu'il existe deux types de raison (pour lesquelles le pourcentage varie en fonction de l'application) expliquant la différence entre un code RTL et ASM.

- Soit ce sont des **traitements inutiles** (i.e. des instructions en trop) que nous déterminons dans la plupart des cas grâce à l'expérience du programmeur.
- Soit ce sont des **instructions plus ou moins spécifiques** qui ne sont pas générées par le compilateur.

Pour estimer les performances d'un code optimisé à partir d'une représentation de niveau RTL il s'agit donc :

- de simplifier le DAG en supprimant les nœuds correspondant aux traitements inutiles,
- ou de modifier le DAG voire le CFG afin de retrouver plus facilement des instructions spécifiques au processeur DSP.

En appliquant un ensemble de règles de réécriture sur la représentation RTL, nous cherchons à obtenir une représentation orientée schéma de calcul DSP (DIR[38]) facilitant le processus d'estimation. Les règles de réécriture sont issues de l'analyse des performances précédente et nous en décrivons quelques unes dans les paragraphes suivants. Les règles de réécriture du code RTL sont appliquées dans un ordre précis (notion de priorité) et certaines sont effectives uniquement à l'intérieur de structure itératives.

[38] **D**SP **I**ntermediate **R**epresentation

5.3.3.1. Affectation de registre

La représentation de niveau RTL comporte de nombreux nœuds « Set_Reg » (affectation de registre) inutiles. Le GNU-C, sur lequel se base le compilateur du OakDSPCore, a en effet été conçu pour des processeurs à usage général comme les RISC. Ces derniers possèdent un modèle « *load-store* » : tout échange de données avec la mémoire se fait via des registres. Ce modèle est très pénalisant pour les processeurs DSP qui possèdent généralement des modes d'adressage permettant d'optimiser les échanges avec la mémoire. Ces instructions supplémentaires font partie de ce que nous appelons l'héritage RISC issu de l'utilisation du GNU-C. Notons que ceci ne concerne pas uniquement les échanges avec la mémoire. L'application de cette règle simplifie considérablement le DAG et permet de se rapprocher davantage d'un modèle de calcul orienté DSP (des chemins de données en particulier). La Figure 51 montre le DAG après l'application de la règle sur le DAG de la Figure 50 pour le bloc de base B_3 de la fonction Hwmcore. Les arcs sont annotés par le nom du ou des registres concernés par la simplification de manière à conserver les dépendances de données.

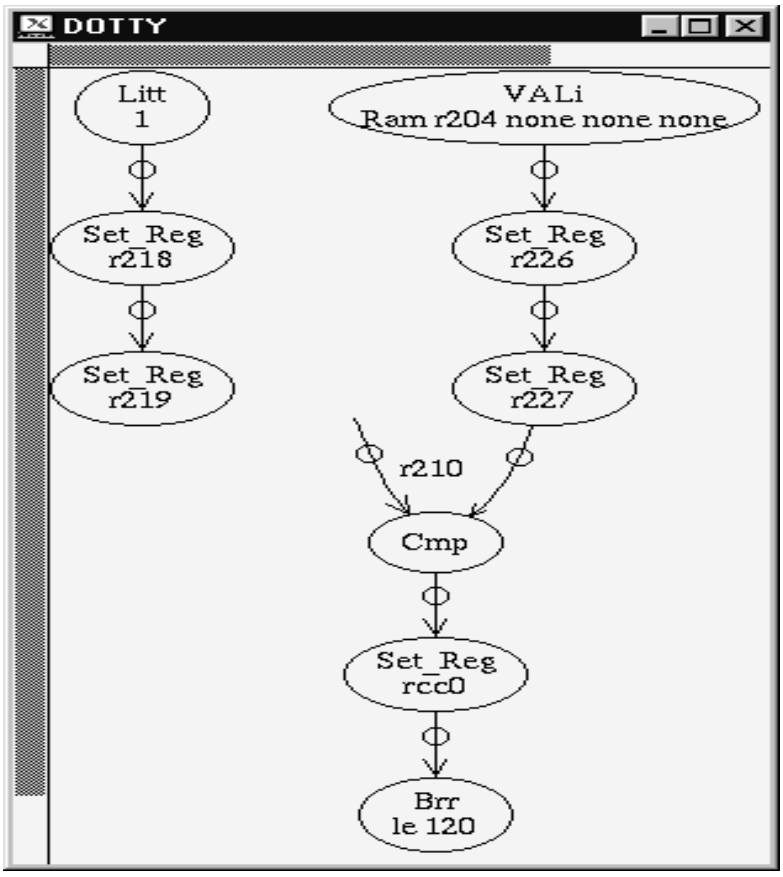


Figure 50 : Exemple de code RTL avec des affectations de registres

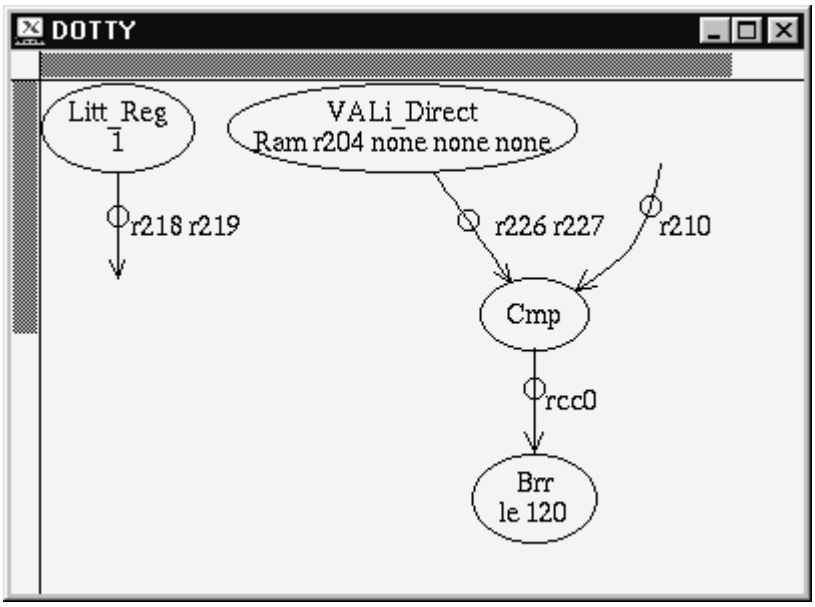


Figure 51 : Exemple de suppression des affectations de registre

5.3.3.2. Décalage inutile du registre P

Le décalage du registre P après une opération de multiplication à l'intérieur d'une boucle est inutile puisqu'il peut être effectué automatiquement par le DSP si le registre PS a été correctement initialisé. La Figure 53 montre le DAG après l'application de la règle sur le DAG de la Figure 52 pour le bloc de base B_{14} de la fonction Hwmcore.

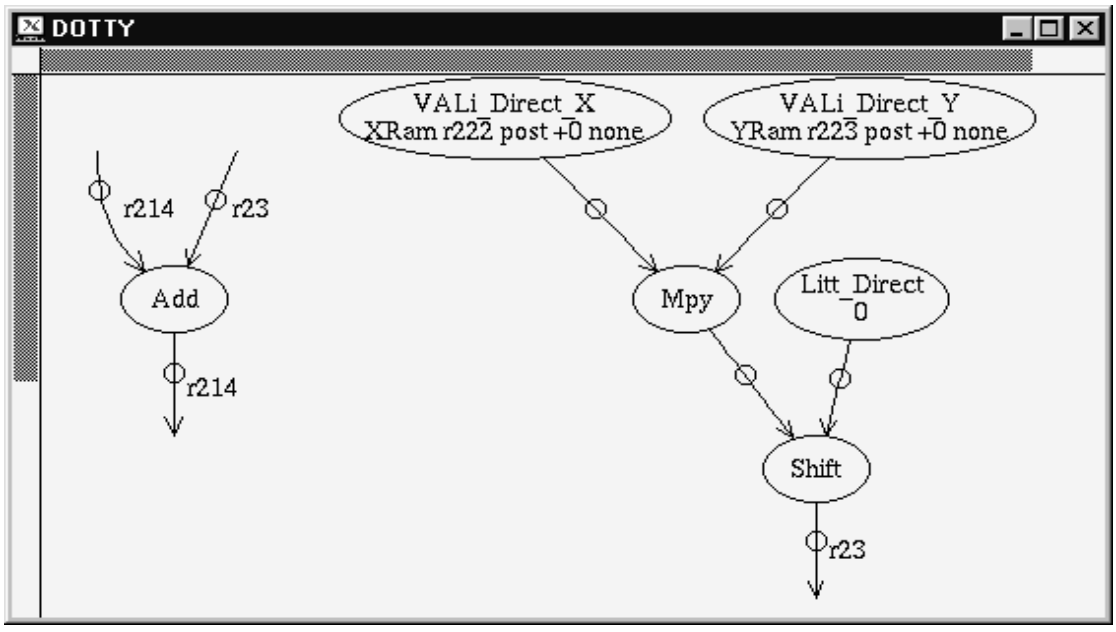


Figure 52 : Exemple de code RTL avec décalage du registre P

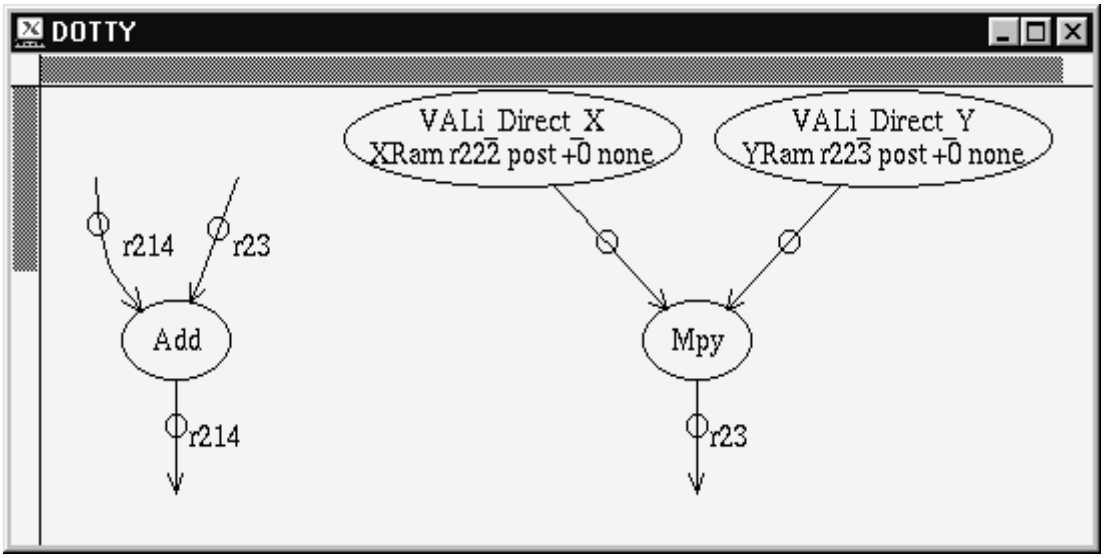


Figure 53 : DAG après suppression d'une opération de décalage du registre P

5.3.3.3. Les instructions conditionnelles

Retrouver une instruction conditionnelle revient à supprimer un test inutile. Les Figure 54 et Figure 55 représentent deux blocs de base consécutifs (les blocs B_2 et B_3 du Bloc17 de G.728). L'instruction « Neg » ne s'exécute que si le résultat du test « Brr » du bloc de base précédent est faux. Comme l'instruction « Neg » du OakDSPCore est conditionnelle, le test « Brr » est inutile.

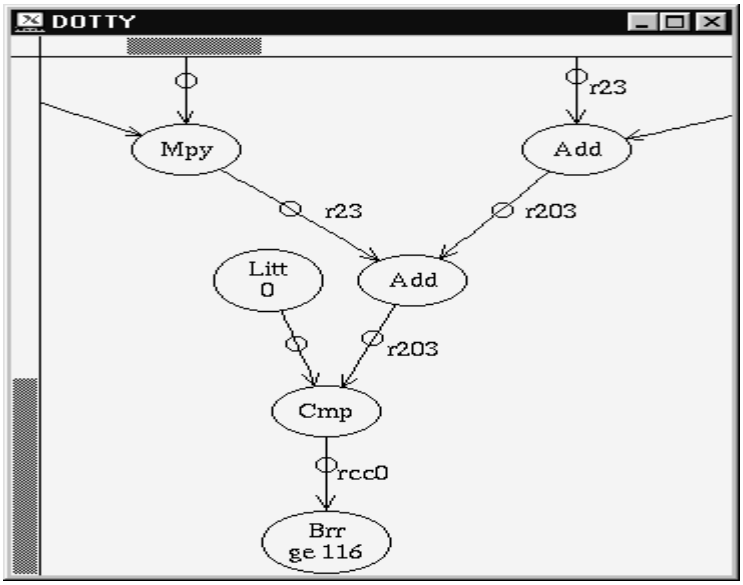


Figure 54 : Exemple d'instruction de test inutile

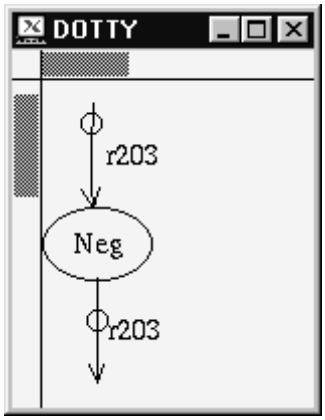


Figure 55 : Instruction qui peut s'exécuter conditionnellement

La Figure 56 montre le DAG après la suppression du test. Notons que la comparaison avec 0 est également supprimée (une autre règle). L'instruction « Neg » est à présent une instruction conditionnelle comme le montre la Figure 57. Elle s'exécute si la non condition du test précédent est vraie.

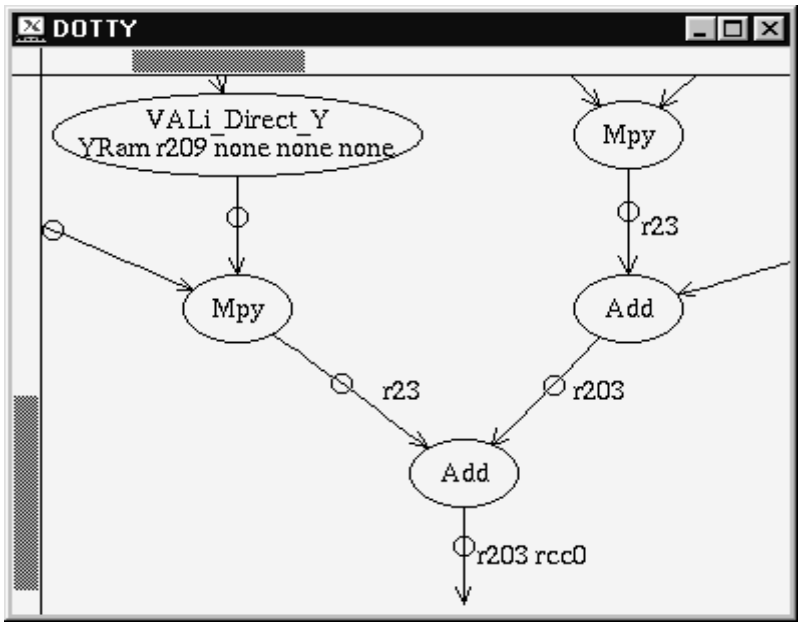


Figure 56 : DAG après suppression du test inutile

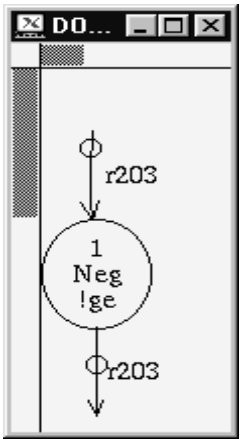


Figure 57 : L'instruction est conditionnelle

L'application de cette règle entraîne une modification du graphe de flot de contrôle. La Figure 58 montre le CFG avant l'application de la règle. Si la condition de l'instruction « Brr » du bloc B_2 est vraie, alors le bloc B_3 n'est pas exécuté et le programme se poursuit en B_4.

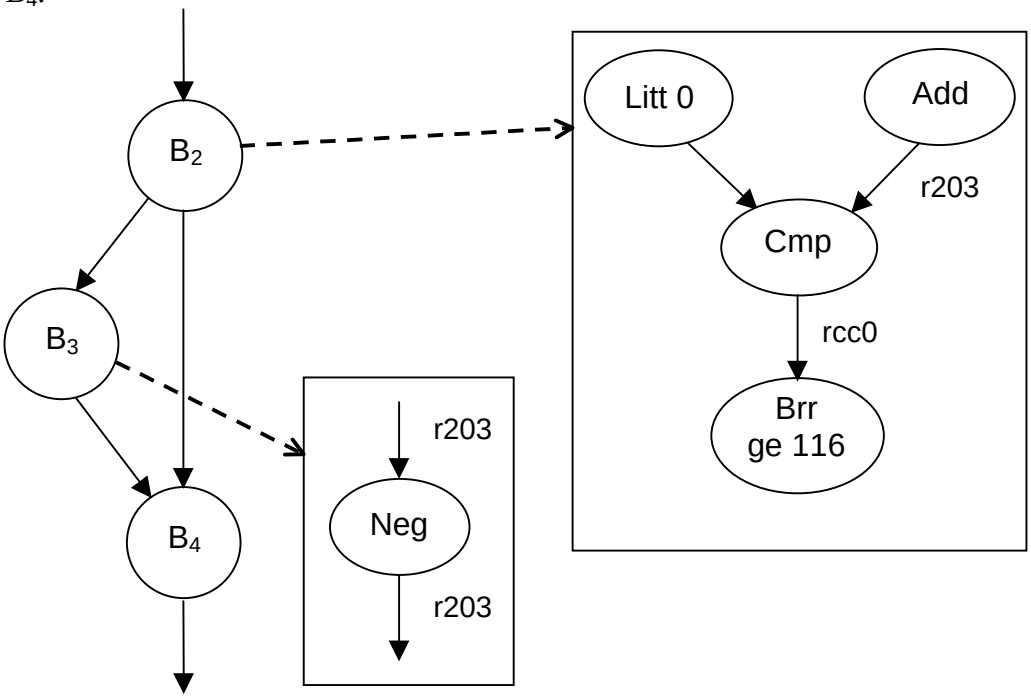


Figure 58 : CFG avant l'application de la règle

En supprimant l'instruction « Brr », il n'existe plus d'alternative à la sortie du bloc B_2. L'instruction à l'intérieur du bloc B_3 s'exécute toujours mais la modification du registre r203 est conditionnelle.

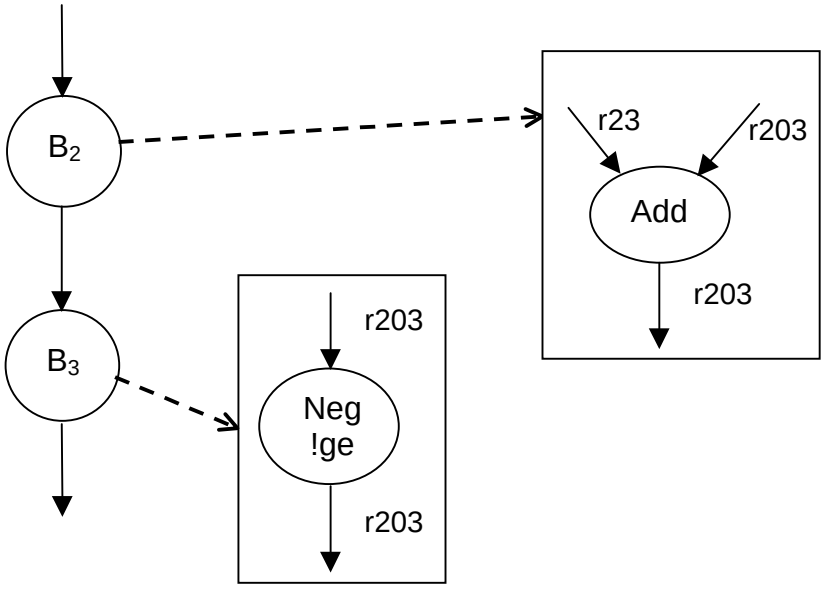


Figure 59 : CFG après l'application de la règle

5.3.3.4. Suppression des calculs effectués sur les registres d'adresse

Cette règle de réécriture est tirée essentiellement de l'expérience des programmeurs et du schéma de calcul utilisé dans les DSP. Elle est effective à l'intérieur des boucles dont le niveau d'imbrication est supérieur ou égal à un. Cette règle ne s'applique que sur les registres d'adresse. Ceci nécessite de déterminer si un registre est utilisé pour le calcul des adresses. Les mises à jour de registres d'adresse sont généralement très coûteuses dans les préambules et postambules des boucles. C'est d'ailleurs sur ces parties de code que la règle est le plus souvent appliquée.

La Figure 60 montre le bloc de base B_9, préambule du Bloc50 de G.728, qui se situe juste avant une structure de boucle. Dans ce bloc, une multiplication est effectuée entre deux données lues en mémoire XRAM et YRAM, via un mode d'adressage indirect. Les registres d'adresse sont r314 pour la mémoire XRAM et r211 pour la mémoire YRAM. Comme le bloc B_9 est dans une boucle de niveau 1, nous faisons l'hypothèse que les calculs effectués sur ces registres sont inutiles.

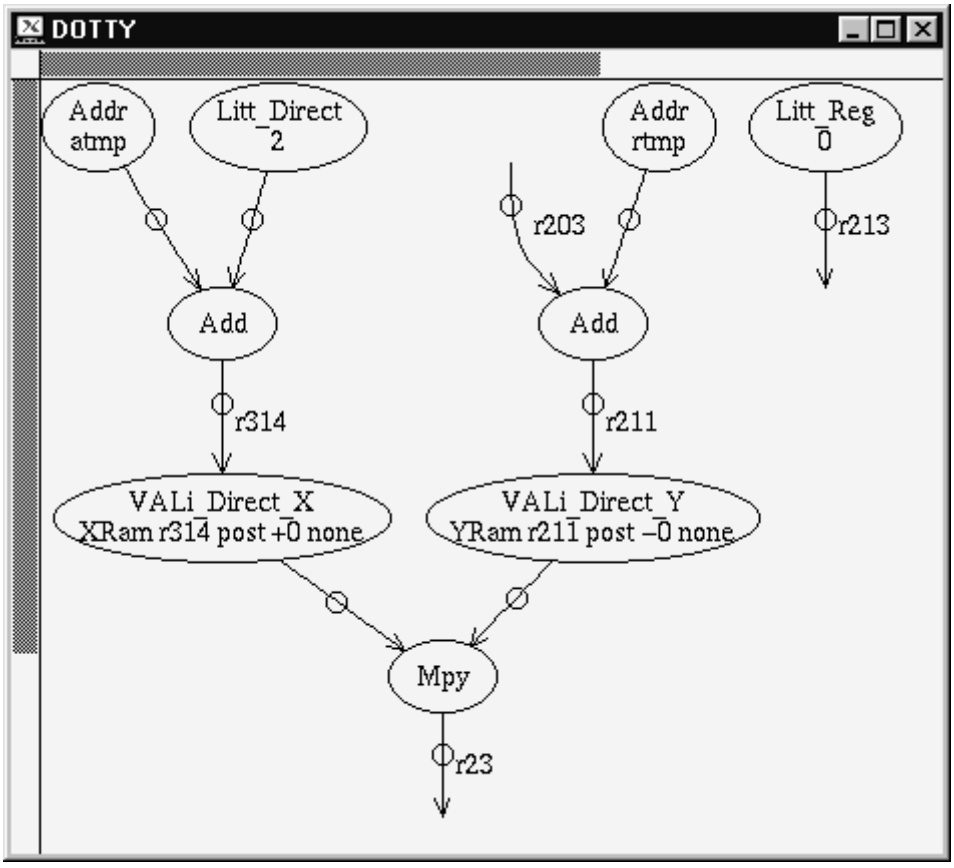


Figure 60 : DAG avec des calculs sur des registres d'adresse

Les opérations d'addition qui mettent à jour les registres d'adresse à chaque nouvelle itération sont supprimées. Nous considérons en effet, que ces calculs peuvent être faits automatiquement par le DSP ou, au pire, à l'extérieur de la boucle.

La Figure 61 montre le DAG après l'application de la règle : les calculs sur les registres d'adresse ont disparu.

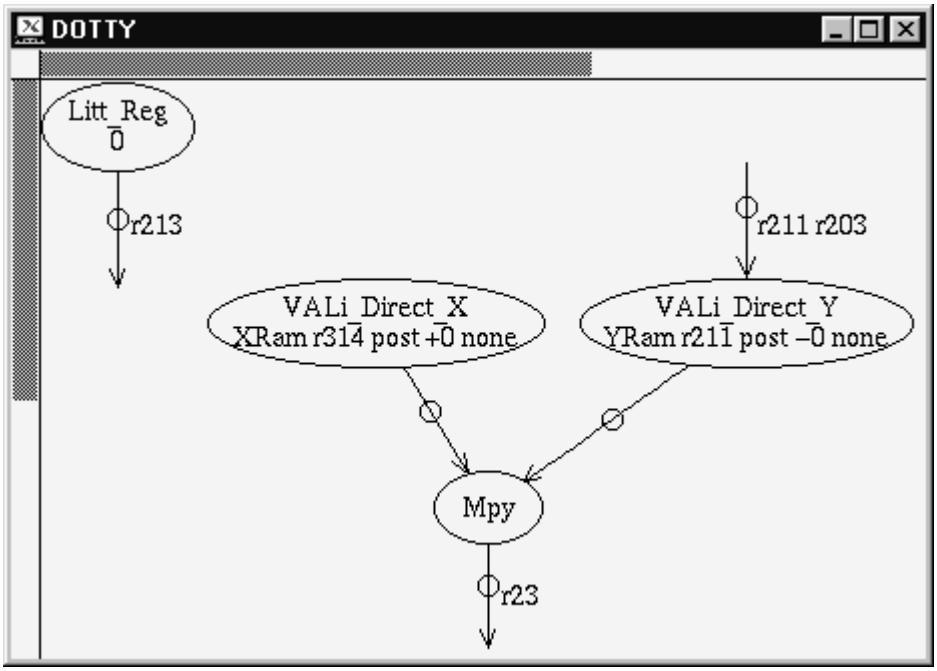


Figure 61 : Le DAG après suppression des calculs sur les registres d'adresses

5.3.4. Conclusion

Dans cette partie, nous avons étudié les différentes raisons d'inefficacité d'un code de niveau RTL par rapport à un code assembleur optimisé. Pour effectuer des estimations précises de performance, la représentation RTL est modifiée en se basant sur un schéma de calcul orienté DSP et des règles déduites de l'expérience des programmeurs. Pour estimer les performances d'un code optimisé il s'agit en effet de supprimer des traitements inutiles ou de retrouver des instructions plus ou moins spécifiques au DSP. On applique pour cela un ensemble de règles qui permettent de passer d'une représentation RTL principalement orientée RISC à une représentation intermédiaire proche d'un modèle de calcul orienté DSP (DIR). Nous allons décrire dans le prochain chapitre le processus d'estimation de performance d'un code optimisé pour une classe de DSP.

Un modèle d'estimation multicible

Le modèle d'estimation utilisé est multicible : il est valable pour une classe de processeurs DSP. Notre approche s'inspire de celle utilisée dans [Gon93] illustrée par la Figure 62. La partie gauche montre un modèle d'estimation spécifique pour un processeur donné. Le modèle décrit sur la partie droite de la Figure 62 utilise quant à lui un modèle d'instructions génériques (à trois adresses), où chaque processeur est caractérisé par un fichier décrivant le processeur cible (fichier technologique).

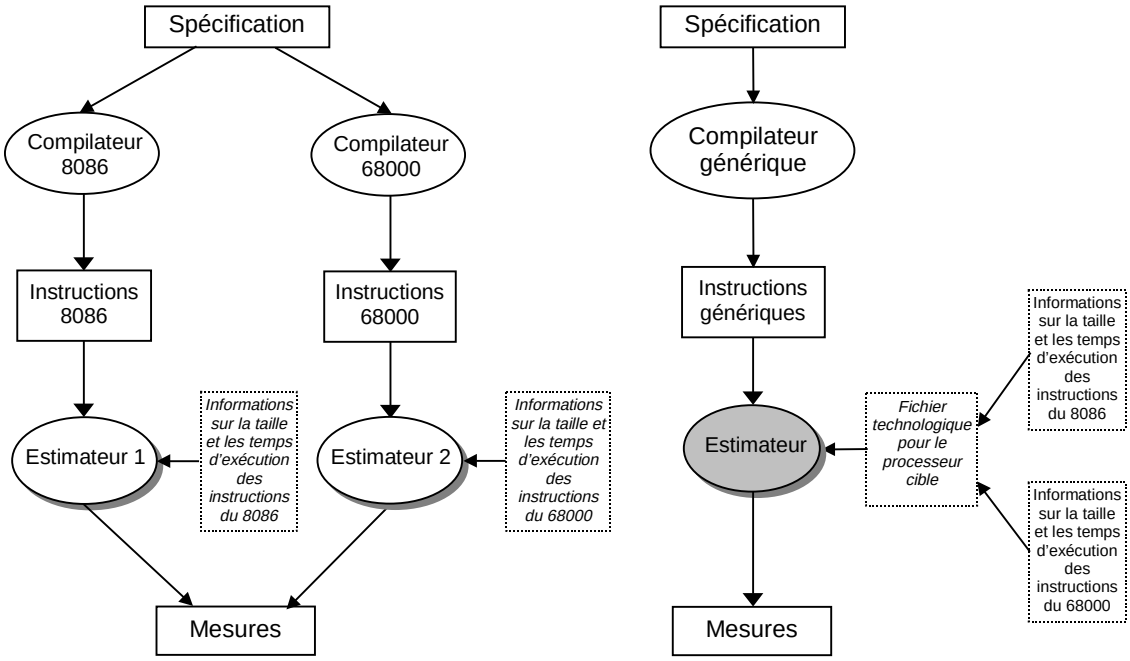


Figure 62 : Modèle d'estimation spécifique ou générique proposé dans [Gon93]

Les avantages d'un modèle générique par rapport à un modèle spécifique à un processeur sont les suivants.

- Un seul compilateur, un seul estimateur et un jeu de fichiers décrivant le processeur cible sont nécessaires pour l'estimateur.
- Le modèle générique permet d'étendre plus facilement l'estimateur à d'autres processeurs cibles puisque seul l'ajout d'un fichier décrivant le processeur cible est nécessaire par processeur.

On peut en revanche attendre des estimations plus précises avec le modèle spécifique car celui-ci peut intégrer et exploiter des détails architecturaux du processeur cible. L'importance de la propriété de portabilité et la facilité d'extension du modèle générique conduisent à préférer ce modèle. La Figure 63 montre le modèle d'estimation générique que nous utilisons (partie *back-end* de la Figure 42 du flot global de l'estimateur). La représentation intermédiaire (DIR) adaptée à une classe de DSP est indépendante du processeur cible (modèle générique interne du processeur) et de l'algorithme d'ordonnancement. Cela permet d'étendre facilement cet estimateur à d'autres DSP en changeant simplement la description du processeur cible. Le OakDSPCore et le PalmDSPCore sont les deux premiers processeurs inclus dans notre estimateur. Comme nous l'avons vu dans la partie précédente, un graphe de flot de contrôle (CFG) pour chaque fonction de l'application est construit et chaque bloc de base est représenté sous la forme d'un DAG. L'application d'un ensemble de règles de réécriture permet d'obtenir une représentation adaptée à un schéma de calcul DSP (DIR). Le processus d'estimation [Peg99a] consiste alors à déterminer le nombre de cycles nécessaire à l'exécution de chaque

bloc de base sur le processeur cible. Nous utilisons pour cela une heuristique d'ordonnancement par liste (*List scheduling*) qui opère au niveau des nœuds du graphe de flots de données, i.e. au niveau opération.

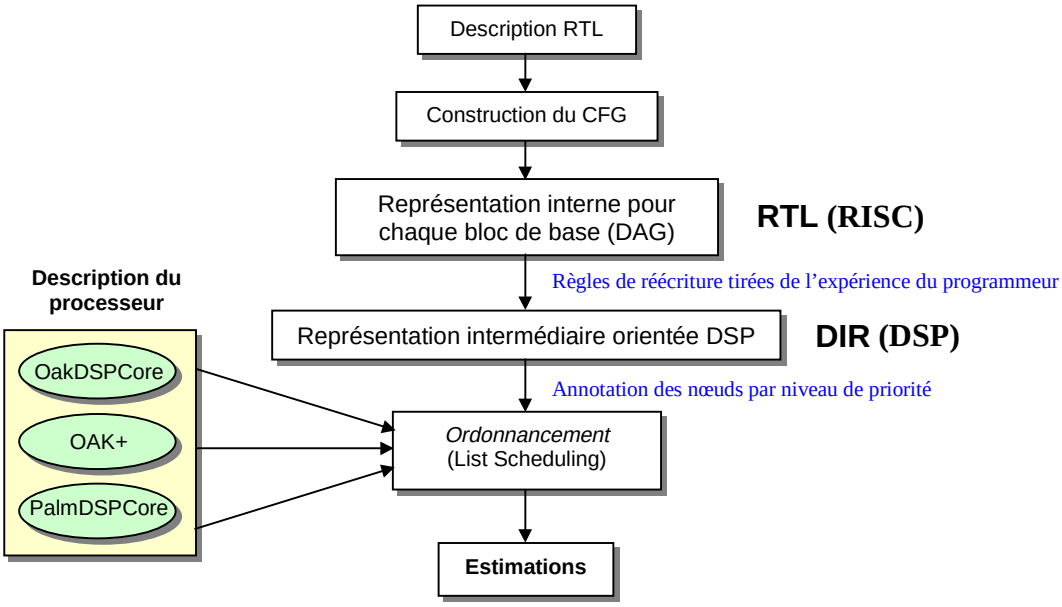


Figure 63 : Le processus d'estimation utilisé

Dans la suite de cette partie, nous décrivons l'algorithme d'ordonnancement, la phase d'annotation du DAG ainsi que le modèle de description du processeur cible utilisé. Nous décrivons alors le processus d'estimation en utilisant quelques exemples représentatifs.

6.1. L'ordonnancement des tâches

L'objectif de la phase d'ordonnancement est d'assembler les opérations élémentaires de la représentation DIR en un nombre de cycles correspondant aux instructions du processeur DSP cible. Nous utilisons une heuristique d'ordonnancement par liste [Lan80] qui opère à un niveau opération et non instruction, car une instruction DSP peut être composée de plusieurs opérations. Une instruction MAC par exemple, est composée de quatre opérations élémentaires : multiplication, addition et deux accès à la mémoire. L'utilisation de méthodes de reconnaissance de motifs (*pattern matching*) au niveau instruction, comme réalisée dans le compilateur GNU, ne nous semble pas adaptée aux architectures DSP récentes possédant beaucoup de parallélisme (e.g. les architectures dual-MAC). Notre méthode s'apparente au modèle de compilation pour architecture VLIW.

6.1.1. Description de l'algorithme d'ordonnancement par liste

L'algorithme d'ordonnancement par liste opère en deux étapes :

- annotation du DAG par niveau de priorité,
- parcours du DAG pour déduire une allocation et un ordonnancement des nœuds du graphe.

L'algorithme d'ordonnancement par liste manipule une liste L d'opérations prêtes à s'exécuter (i.e. ordonnançables) triées par ordre décroissant des niveaux de priorité. Une tâche ordonnançable est un nœud de la DIR pour lequel tous les nœuds prédécesseurs ont été

ordonnancés. Quand une tâche est ordonnancée, ses successeurs sont insérés dans la liste des tâches de même priorité (nous verrons plus tard dans quel ordre). Au début, toutes les ressources (mémoires, registres ou opératoires) sont disponibles.

L'algorithme est :

```
début

n = 0 ;      /* n représente l'instant d'ordonnancement */

    tantque (L != 0) faire
       pour (toute tâche A de L) faire
          si (A peut-être ordonnancée au cycle n) alors
          {
              Ordonnancer A ;
              L = L – {A} ;
              Mettre à jour la table d'allocation des ressources;
              Mettre à jour la liste L des tâches exécutables;
          }
          finsi
       finpour
       n = n+1 ;
    fintantque ;

fin ;
```

Figure 64 : Algorithme d'ordonnancement par liste

La fonction « *peut-être ordonnancée au cycle n* » rend un résultat vrai si toutes les ressources nécessaires à l'exécution de l'opération A (suivant son schéma de calcul d'exécution) sont disponibles et si toutes les opérations qui ont pour prédécesseur A ont rendu leur résultat. Chaque opération de la DIR possède un ou plusieurs schémas de calcul défini dans la description du processeur cible et que nous appelons **opération de base**. La fonction « *mettre à jour la table d'allocation (ou table de réservation) des ressources* » consiste à réserver les ressources qui interviennent dans la réalisation d'une opération de base. Toutes les informations nécessaires à la mise à jour de la table d'allocation sont définies pour chaque opération de base dans la description du processeur cible.

6.1.2. L'annotation du DAG par niveau de priorité

La première étape (avant la phase d'ordonnancement) consiste à associer à chaque nœud du graphe de flots de données un niveau de priorité. Classiquement, ce nombre est égal à la distance en nombre de cycles entre ce nœud et le nœud feuille (ou terminal) le plus éloigné. Les nœuds avec le niveau de priorité le plus élevé sont ordonnancés en premier. Ce type d'annotation privilégie les opérations qu'il est souhaitable de prendre en compte pour arriver à ordonnancer au plus tôt les nœuds feuilles les plus éloignés. Les résultats d'ordonnancement sont fortement dépendants de l'annotation des nœuds du graphe. Une annotation trop simple du graphe ne permet pas de retrouver le parallélisme ou le pipeline interne du processeur.

6.1.2.1. Annotation simple du graphe

Prenons le calcul du papillon de la FFT (partie réelle uniquement) comme exemple. L'annotation s'effectue en partant des nœuds feuilles du graphe auxquels on affecte un niveau de priorité égal à 1. A chaque nœud prédécesseur, on incrémente le niveau de priorité.

Cette annotation simple des nœuds du graphe donne les niveaux de priorité présentés sur la Figure 65. Montrons à présent qu'une telle annotation ne permet pas d'extraire le parallélisme supporté par le DSP.

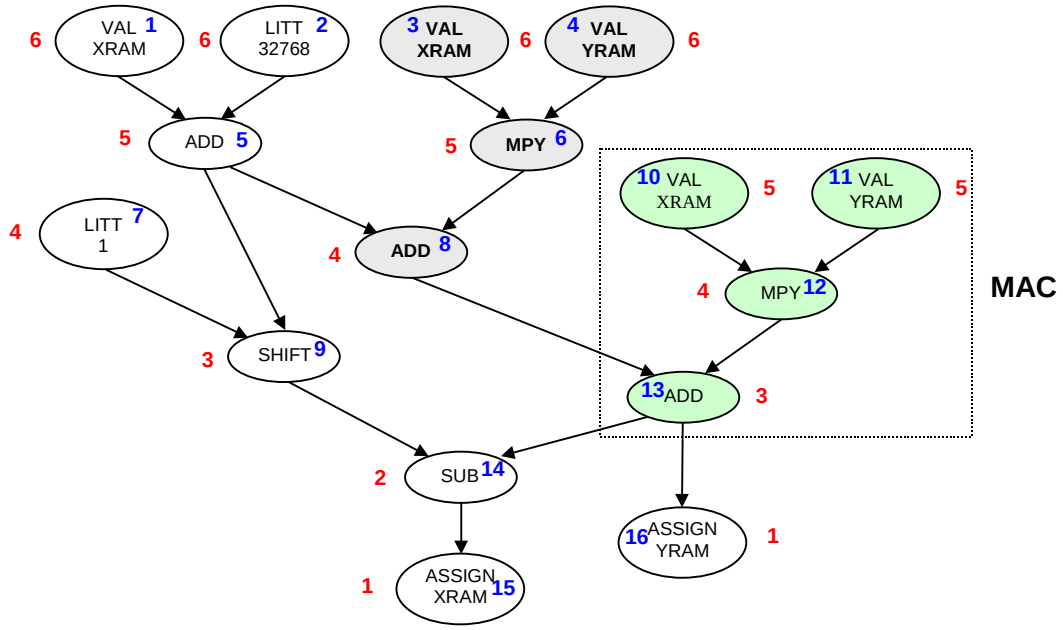


Figure 65 : Annotation simple des niveaux de priorité

La liste L des tâches ordonnançables à n = 0 est donc la suivante :

Niveau de priorité	6	6	6	6	5	5	4
Liste L (numéro du nœud)	1	2	3	4	10	11	7

Notons que les nœuds 1, 2, 3 et 4 (c'est le cas également pour les nœuds 10 et 11) ont des niveaux de priorité identiques égaux à 6. Leur ordre d'apparition dans la liste L dépend uniquement de l'algorithme de parcours du graphe. Ceci n'est vrai que lors de la création de la liste L à l'instant n = 0.

Sur le OakDSPCore, un code assembleur optimisé écrit manuellement nécessite huit cycles pour exécuter le graphe représenté sur la Figure 65. Parmi ces huit cycles, il y a deux instructions de type MAC (nœuds grisés). Ces instructions ne peuvent pas se retrouver si l'annotation du graphe ne tient pas compte du parallélisme pipeline interne du DSP. Voici en effet l'ordonnancement de ce graphe si l'on considère que les nœuds 1 et 2 ont déjà été ordonnancés.

Niveau de priorité	6	6	5	5	5	4
Liste L (numéro du nœud)	3	4	5	10	11	7

Si l'on ordonnance les opérations 3 et 4, la tâche « MPY » correspondant au nœud 6 de la Figure 65 devient ordonnançable et on l'insère (en tête du fait de sa priorité égale à 6) dans la liste L des opérations de même niveau de priorité. On obtient la liste suivante :

Niveau de priorité	5	5	5	5	4
Liste L (numéro du nœud)	6	5	10	11	7

Après ordonnancement de l'opération « MPY », son successeur le nœud « ADD » n'est pas ordonnançable puisque l'opération « ADD » de niveau de priorité 5 (nœud 5) n'a toujours pas été ordonnancée. La liste L mise à jour est en effet la suivante :

Niveau de priorité	5	5	5	4
Liste L (numéro du nœud)	**5**	**10**	**11**	**7**

Cet exemple montre que, sans une annotation adaptée du graphe, l'algorithme d'ordonnancement ne parvient pas à retrouver les instructions MAC du graphe de la Figure 65. L'instruction « ADD » (numéro 8) ayant un niveau de priorité égal à 4, les nœuds 5, 10 et 11 seront ordonnancés avant. La situation est identique pour le second MAC constitué des opérations 10, 11, 12 et 13. Notons que ceci ne concerne pas uniquement les instructions de type MAC. Nous avons donc cherché à annoter les nœuds du DAG avec des niveaux de priorité tenant compte du parallélisme pipeline interne du DSP cible. C'est ce que nous appelons une annotation orientée schéma de calcul DSP.

6.1.2.2. Une annotation orientée schéma de calcul DSP

Le calcul de la distance entre un nœud et le nœud feuille prend en compte le parallélisme pipeline du DSP : deux nœuds consécutifs de la DIR ont le même niveau de priorité s'ils peuvent avoir une exécution pipeline ou parallèle sur le DSP.

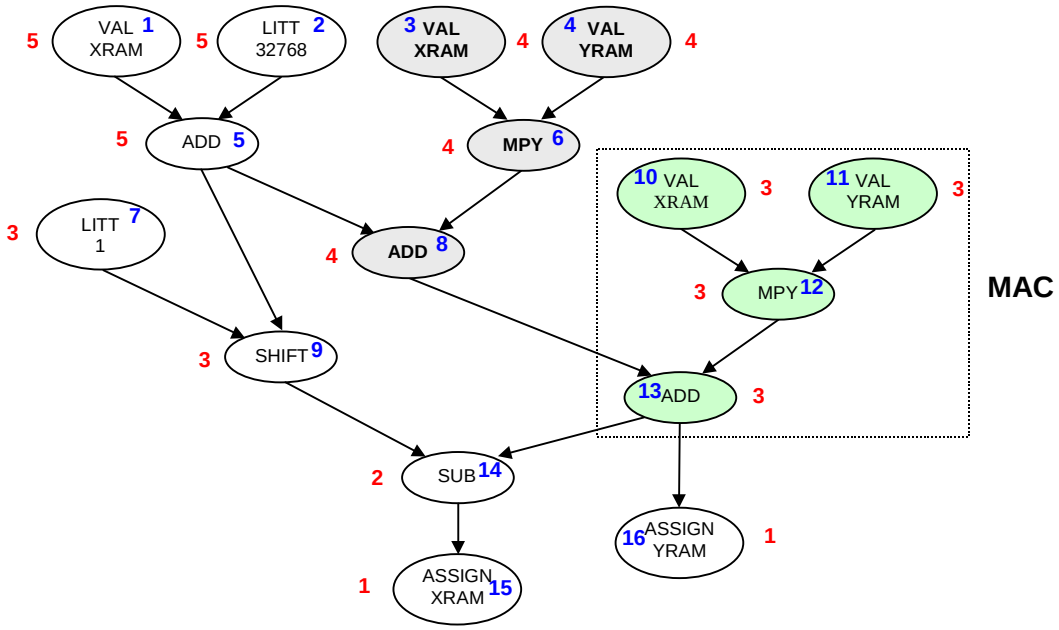


Figure 66 : Annotation du DAG orientée schéma de calcul DSP

La Figure 66 montre pour le calcul du papillon de FFT, les niveaux de priorité ainsi obtenus en utilisant une annotation orientée schéma de calcul DSP. On remarque à présent que les quatre opérations formant un MAC ont les mêmes niveaux de priorité, ce qui favorise leur ordonnancement dans le même cycle. En pratique, lorsqu'un nœud « ADD » est précédé d'un nœud « MPY », ce dernier a le même niveau de priorité que son successeur puisqu'ils peuvent s'exécuter en parallèle sur le DSP. De même, lorsqu'une opération « MPY » est précédée de deux opérations de lecture en mémoire XRAM et YRAM (nœud « VAL »)[39], ces derniers ont également un niveau de priorité identique. Grâce à cette

[39] Vrai uniquement pour certains modes d'adressage

annotation, les nœuds 1, 2 et 5 sont d'abord ordonnancés, puis les nœuds 3, 4, 6 et 8 formant un premier MAC. Pour retrouver le second MAC constitué des nœuds 10, 11, 12 et 13, il est nécessaire de gérer le cas où il existe d'autres nœuds de niveau de priorité égale à 3 (nœuds 7 et 9).

6.1.2.3. Insertion des tâches en tête de liste

L'insertion d'une nouvelle tâche ordonnançable dans la liste L doit se faire en tête des tâches de même niveau de priorité. On montre ceci très facilement à l'aide du graphe de la Figure 66. Soit la liste L des tâches ordonnançables lorsque toutes les tâches de niveau de priorité 4 et 5 ont été ordonnancées.

Niveau de priorité	**3**	**3**	**3**
Liste L (numéro du nœud)	**10**	**11**	**7**

Si les tâches 10 et 11 sont ordonnancées, l'opération numéro 12 « Mpy » devient ordonnançable et il faut mettre à jour la liste L. Or, cette dernière possède une opération (« LITT ») de même niveau de priorité. Si le nœud numéro 12 est inséré **après** la tâche 7 dans la liste L, soit :

Niveau de priorité	**3**	**3**
Liste L (numéro du nœud)	**7**	**12**

on constate que l'instruction MAC formée des opérations 10, 11, 12 et 13 n'est pas retrouvée. Au contraire, si les opérations 12 puis 13 sont insérées en tête des tâches de même niveau de priorité, dans ce cas les quatre nœuds du DAG formant l'instruction MAC sont ordonnancés dans le même cycle.

Cette technique combinée avec l'annotation de la DIR permet également de réduire la durée de vie des variables. C'est un facteur essentiel lors de la programmation des DSP puisque ces derniers ont généralement un jeu de registres restreint et spécifique. D'ailleurs, lors de l'écriture manuelle d'un code assembleur un programmeur cherche généralement à réduire la durée de vie des variables afin d'éviter des sauvegardes inutiles en mémoire.

6.1.2.4. Le traitement des composantes connexes

Un DAG peut-être formé de plusieurs composantes connexes comme le montre l'exemple de la Figure 67. Cet exemple représente le graphe de flots de données d'un algorithme FIR pour lequel la boucle a été dépliée une fois afin de faire apparaître plus de parallélisme (deux MAC).

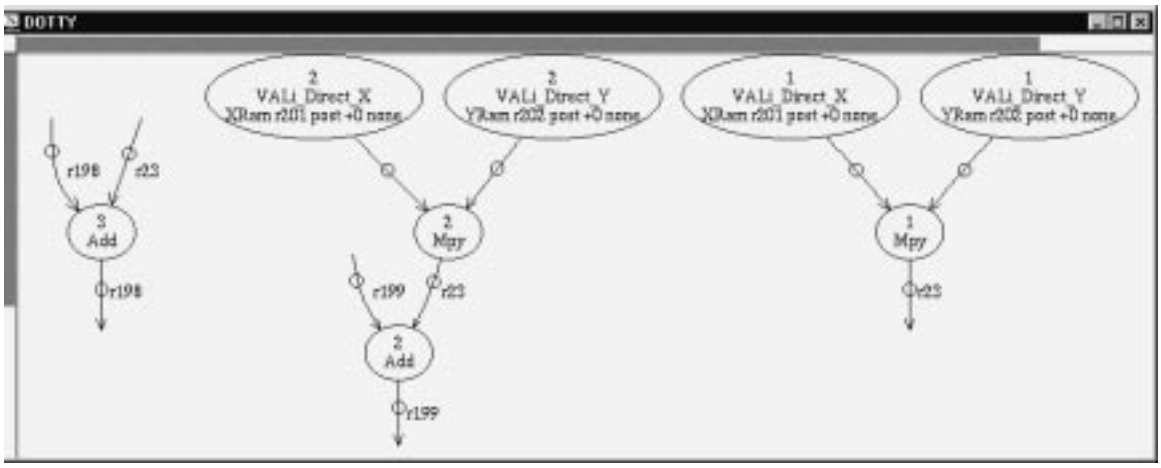


Figure 67 : Graphe de flots de données possédant plusieurs composantes connexes

Le traitement des composantes connexes (leur ordonnancement) dépend du parallélisme potentiel du processeur cible et est pris en compte lors de l'annotation du graphe par niveau de priorité (l'algorithme d'ordonnancement est ainsi indépendant du processeur cible). Pour les processeurs avec peu de parallélisme comme le OakDSPCore, chaque composante connexe est traitée séparément afin de favoriser un ordonnancement local des nœuds du graphe. Pour cela, lors de l'annotation du graphe, le niveau de priorité[40] du nœud feuille de chaque nouvelle composante connexe est égal au niveau de priorité maximum de la composante connexe précédente. C'est l'annotation réalisée sur la Figure 67. Cette annotation, bien que triviale, donne d'excellents résultats comme le montrent les études avec le OakDSPCore présentées dans le chapitre suivant.

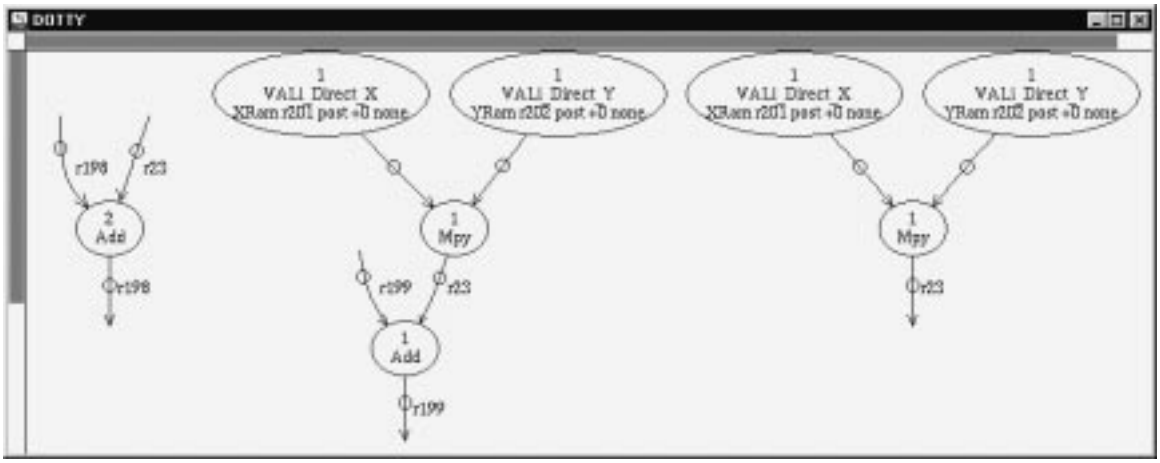


Figure 68 : Annotation pour un processeur dual-MAC

Pour une architecture avec plus de parallélisme comme le PalmDSPCore, il est nécessaire de modifier l'annotation du graphe par niveau de priorité. Des études

[40] Les niveaux de priorité sont affichés à l'intérieur des nœuds du graphe au dessus du nom de l'opération.

préliminaires sur un algorithme FIR ou le papillon de FFT ont montré en effet l'intérêt de traiter en parallèle plusieurs composantes connexes afin d'exploiter avec plus d'efficacité le parallélisme potentiel du DSP (i.e. obtenir de meilleures estimations). Pour le PalmDSPCore en particulier, il semble intéressant de traiter en parallèle deux composantes connexes. Le problème revient à partitionner au préalable les composantes qu'il est souhaitable d'ordonnancer en parallèle. L'algorithme d'annotation du graphe par niveau de priorité opère alors sur chacune des parties ainsi définies. Sur la Figure 68, les deux composantes connexes de droite ont à présent les mêmes niveaux de priorité, ce qui favorise leur ordonnancement en parallèle. Une étude plus complète est cependant nécessaire pour valider le partitionnement des composantes connexes.

6.2. Le modèle de description du processeur cible

6.2.1. Etude des langages de description de processeurs existants

Différentes approches sont généralement considérées pour la modélisation du processeur cible [Mar95b]. Les modèles comportementaux[41] décrivent le processeur à partir de son jeu d'instructions et sont adaptés aux méthodes de génération de code basées sur la reconnaissance de motifs. Leur principal inconvénient est leur difficulté à modéliser les effets du pipeline et l'occupation des unités fonctionnelles pour des architectures complexes. Les modèles structurels décrivent quant à eux les caractéristiques internes des processeurs (connectivité, mécanisme de contrôle, …) et sont généralement utilisés pour la synthèse d'architecture. Ils sont ainsi mieux adaptés pour exploiter le parallélisme de niveau instruction des processeurs. Avant de décrire le modèle de description du processeur utilisé par notre méthode d'estimation, nous formulons les besoins en terme d'estimation logicielle et décrivons les modèles de processeurs existants.

6.2.1.1. Les besoins pour une méthode d'estimation logicielle

L'objectif du modèle de description est de pouvoir modéliser rapidement le processeur cible en trouvant un moyen d'éviter une énumération complète du jeu d'instructions. C'est donc un modèle simplifié mais qui doit permettre de décrire un large éventail d'architectures DSP avec un niveau de détail suffisant pour effectuer des estimations précises de performance. Bien que notre objectif principal ne soit pas l'exploration architecturale, nous considérons comme important le fait de pouvoir aisément et rapidement modifier le nombre de registres, d'unités fonctionnelles, de bancs de mémoire ou encore les fonctionnalités (e.g. parallélisme entre unités fonctionnelles) du processeur. Le langage de description doit être également adapté pour une utilisation aisée par une personne ne connaissant pas le processus d'estimation (en particulier la représentation intermédiaire ou l'algorithme d'ordonnancement).

Par contre, il n'est pas utile que la description contienne toutes les informations nécessaires à la production automatique d'un générateur de code. En effet, dans le processus d'estimation, seule la phase d'ordonnancement dépend directement de la description du processeur cible. Nous détaillerons plus tard les informations importantes pour effectuer des estimations précises de performance. Dans le prochain paragraphe sont décrits les différents modèles de description existants.

6.2.1.2. Les modèles de description existants

[41] Le compilateur C du GNU utilise ce type de description [Sta94].

Tous les langages de description présentés dans la suite (sauf LISA) sont utilisés dans le cadre de travaux sur la compilation flexible et/ou l'exploration architecturale. Cet état de l'art s'inspire des études présentées dans [Mes99b] et [Had99].

MIMOLA

Le langage MIMOLA [Leu98a] est un modèle de description structurelle de bas niveau du processeur utilisé par les compilateurs MSSQ et Record [Leu97]. Une description dans un langage structurel peut servir de point d'entrée privilégié à un outil de synthèse, un compilateur et un simulateur (par contre un simulateur basé sur une description de ce type est lent mais peut être précis). Elle s'adapte naturellement à la description du processeur utilisé par un architecte. Cependant, la précision des descriptions et la capacité d'expression de MIMOLA font que l'élaboration d'une description complète du processeur est un travail long et difficile. La modification du jeu d'instructions par exemple est une opération délicate au niveau structurel mais facile à effectuer au niveau comportemental. De plus, dans le cadre de l'estimation logicielle ou de l'exploration architecturale, l'utilisation de MIMOLA conduit à des descriptions longues et complexes inadaptées par rapport aux objectifs visés. D'un autre côté, le niveau de détail rend beaucoup plus facile la synthèse du matériel à partir de cette description.

LISA

Le langage de description LISA[Ziv96] a été conçu spécifiquement pour la génération automatique de simulateurs compilés au niveau cycle et rapides. Basé sur un modèle de description comportemental, LISA permet de décrire les détails architecturaux et le pipeline des processeurs DSP récents [Pee99]. La génération automatique de matériel n'a pas encore été montrée comme possible à partir d'une représentation du processeur en langage LISA. De plus, LISA n'est pas très bien adapté à la production automatique de générateurs de code ou d'assembleurs.

ISDL

Le langage de description des processeurs ISDL[42] [Had97][Had99] a été conçu spécifiquement pour l'exploration architecturale. ISDL est un langage comportemental qui liste explicitement le jeu d'instructions du processeur cible. Il permet une modélisation extrêmement détaillée du jeu d'instructions et une description de l'ensemble des mécanismes présents dans les architectures spécialisées (niveau élevé de détail). Il est basé sur une grammaire où les règles de production sont utilisées pour abstraire les motifs communs dans la définition des opérations (voir l'exemple fourni pour le langage nML). Le langage ISDL a pour objectif de couvrir un large éventail d'architecture avec une emphase plus particulière sur les architectures de type VLIW. Une instruction dans une architecture VLIW consiste en une combinaison d'opérations, une pour chaque unité fonctionnelle. Pour modéliser ceci, le jeu d'instructions est décrit comme une liste de champs, chacun d'eux étant une liste de définition d'opération. Un champ correspond au jeu d'opérations exécutables par une unité fonctionnelle. Ainsi les opérations à l'intérieur d'un champ sont mutuellement exclusives et ne peuvent pas être utilisées en parallèle puisqu'elles utilisent toutes la même unité fonctionnelle. Pour former une instruction VLIW les opérations sont sélectionnées, une par champ, et groupées ensemble. Les combinaisons valides (i.e.

[42] Instruction Set Description Langage

instructions valides) sont décrites en listant pour chaque instruction un ensemble de contraintes qui doivent toutes être satisfaites. Si une seule contrainte n'est pas respectée alors l'instruction n'est pas valide. Les contraintes peuvent également fournir des informations sur l'implémentation du jeu d'instructions de manière à faciliter la génération efficace du matériel. La description est concise et intuitive mais un des inconvénients de ce langage est qu'il est organisé en fonction de la structure du mot d'instruction (format binaire), ce qui rend difficile la modélisation d'un processeur avant d'avoir spécifié en détail le codage binaire.

nML

Le langage nML [Fau95] est un langage de description de haut niveau qui peut être utilisé pour supporter des outils générés automatiquement. C'est le langage de description utilisé dans le système de génération de code reciblable CHESS [Lan95]. Le jeu d'instructions est représenté sous une forme comportementale en utilisant une grammaire. La structure du langage sous forme de grammaire a l'avantage de factoriser les regroupements d'informations puisqu'une règle de grammaire suffit à modéliser différentes alternatives. Par exemple, pour décrire les transferts de registres effectuant une addition et une soustraction, une solution est d'énumérer tous les cas possibles [Mes99b]:

```
Acc = Acc  +  Reg1
Acc = Acc  +  Reg2
Acc = Acc  +  Reg3
Acc = Acc  -  Reg1
Acc = Acc  -  Reg2
Acc = Acc  -  Reg3
```

Une règle de grammaire suffit cependant à modéliser les alternatives du second opérande, en déclarant un non-terminal **operand2** qui se dérive de trois manières différentes :

```
Acc = Acc  +  Operand2
Acc = Acc  -  Operand2
Operand2 : Reg1 | Reg2 | Reg3
```

Cette représentation hiérarchique permet de réduire la taille des descriptions, améliore la lisibilité et limite les risques d'erreurs ou d'oublis. nML est très similaire à ISDL excepté dans la manière dont les contraintes sont prises en compte. nML ne peut décrire que des instructions valides, ce qui le contraint à utiliser des règles supplémentaires pour décrire des combinaisons invalides. Il en résulte des descriptions plus longues et moins intuitives. Finalement, il n'y a pas d'information claire sur la manière dont nML est adapté à la synthèse de matériel.

ARMOR

Le langage ARMOR[43] [Mes99b] utilise la philosophie de nML : une description est une grammaire dont chaque dérivation est un comportement possible du processeur. La grammaire définit des *composants* assimilables à des instructions et crée des groupes de composants parallèles pour refléter le parallélisme d'instructions. ARMOR ne décrit pas une liste d'instructions, mais plutôt la liste des comportements possibles du processeur mis à

[43] Architecture Modeling for Retargetability

disposition du programmeur via le jeu d'instructions. Dans une description du processeur cible, il est nécessaire de connaître à quel cycle une instruction utilise les ressources. ARMOR attribue pour cela des dates d'utilisation aux ressources opératoires ou registres. Cette technique de description est intéressante puisque généralement les instructions utilisent les ressources du processeur dans les mêmes conditions.

6.2.1.3. Techniques de modélisation utilisées pour l'estimation de performance

La description du processeur dans notre méthode d'estimation n'utilise aucun des langages décrit précédemment. Les langages structurels comme MIMOLA contiennent beaucoup trop d'informations inutiles pour effectuer des estimations. Le langage comportemental ISDL n'est pas adapté à notre cahier des charges puisqu'il est organisé sur la base du format binaire d'un mot d'instruction. D'autre part, nML est incapable de décrire des combinaisons invalides d'instructions ce qui le rend difficilement exploitable pour des architectures comportant beaucoup de parallélisme. ARMOR semble être le langage le mieux adapté à nos besoins. Lors de l'élaboration de notre modèle de processeur, les caractéristiques de ce langage n'avaient pas encore été publiés. Notons que le modèle utilisé dans notre méthode d'estimation est une description comportementale abstraite utilisant certaines techniques utilisées dans les langages présentés.

Une description au niveau opération

Une des principales propriétés est que notre algorithme d'ordonnancement traite des opérations élémentaires et non des instructions. En conséquence, le modèle du processeur décrit toutes les opérations et les possibilités de les assembler pour former une instruction. Pour cela, on attribue à chaque opération des contraintes sur l'utilisation des ressources afin de limiter le parallélisme. Rappelons que le parallélisme d'un processeur est limité soit par ses chemins de données, soit par l'encodage du jeu d'instructions.

Modélisation du parallélisme

Les méthodes par énumération sont adaptées au processeur pour lequel le parallélisme est restreint à un nombre limité de cas particulier. Comme on souhaite décrire des DSP avec beaucoup de parallélisme comme les architectures dual-MAC ou VLIW, et que de plus on manipule des opérations et non des instructions (i.e. granularité plus fine), on utilise une méthode par restriction. Ces méthodes consistent à définir des contraintes limitant le parallélisme plutôt que d'énumérer l'ensemble des opérations qu'il est possible d'effectuer en parallèle. On comprend aisément le gain en temps que cela représente pour des architectures VLIW par exemple. Cependant, nous avons constaté avec le OakDSPCore qu'une description par restriction peut être aussi très efficace pour décrire un processeur avec très peu de parallélisme.

Un modèle simplifié du processeur

La simplification du modèle de processeur se caractérise par trois principales mesures.

- *Utiliser des classes.*

Quand des opérations impliquent l'utilisation de ressources communes, ces dernières sont regroupées au sein d'une même classe. La classe MUL du PalmDSPCore par exemple, est composée des deux multiplieurs de ce processeur.

De la même manière, nous définissons des classes de registres. Tous les registres qui appartiennent à la même classe ont les mêmes contraintes sur les chemins de données pour chaque opération de base utilisant cette classe. Le registre P par exemple, est une classe représentant un des deux registres P0 ou P1 du PalmDSPCore. L'utilisation des classes simplifie de manière significative la description du processeur : notre fichier de description du OakDSPCore est environ dix fois plus petit que le fichier de description du même processeur utilisé par le compilateur C du OakDSPCore.

Notons enfin que l'utilisation de classes est une approche similaire à celle utilisant des non-terminaux pour des langages représentés sous forme de grammaire.

- *Simplifier le jeu de registres.*

Les registres décrits dans notre modèle de processeur se limitent à ceux nécessaires à la modélisation pour l'estimation de l'ensemble des instructions et des contraintes liées à l'exécution parallèle des opérations.

Pour le OakDSPCore, voici l'ensemble des registres définis par le manuel utilisateur:

> rI, rJ, aX (aXl, aXh, aXe), bX (bXl, bXh, bXe), cfgI, cfgJ, sv, sp, pc, lc, extX (X = 0, 1, 2, 3), x, y, p (pH), rb, st0, st1, st2, mixp, icr, repc et dvm.

L'ensemble des registres manipulés par notre estimateur est restreint à la liste suivante :

> x, y, p, ACC, rI, rJ et RR.

Les registres x, y, p, rI et rJ sont les mêmes que ceux décrits dans le manuel utilisateur. Par contre ACC représente un des accumulateurs aX ou bX pour lesquels nous ne décrivons pas les mots formés des 16 bits de poids forts ou de poids faibles, ni la partie extension de ces accumulateurs. RR définit quant à lui tous les autres registres utilisés par le processeur (e.g. cfgI, cfgJ, mixp, sv, lc, ..).

- *Ignorer certaines caractéristiques du processeur.*

Le modèle utilisé ne prend pas en compte toutes les caractéristiques architecturales du processeur ou de son jeu d'instructions. Par exemple, il n'existe aucune information sur le pipeline ou les registres de mode. En effet, dans le cas du OakDSPCore et du PalmDSPCore le pipeline n'est que très peu apparent pour le programmeur et n'a pas beaucoup d'influence sur les performances. Notons néanmoins qu'il serait facile d'étendre notre modèle de processeur s'il apparaissait que le pipeline a une influence non négligeable sur les performances. La méthode utilisée dans ARMOR par exemple [Mes99b] s'applique très facilement à notre modèle. Les registres de mode sont utilisés avant tout pour réduire la taille du mot d'instruction. Compte tenu des simplifications effectuées sur le jeu de registres du processeur, il n'est plus possible de décrire les instructions utilisant les registres de mode. Ce n'est pas gênant puisque ces registres ont eux aussi très peu d'influence sur les performances lors de l'exécution d'un programme.

6.2.2. Syntaxe du modèle de processeur

La description du processeur cible représente les ressources du DSP cible incluant les compatibilités ou les conflits résultant du jeu d'instructions du processeur. Grâce à ces informations, l'algorithme d'ordonnancement doit être en mesure de retrouver des instructions parallèles comme l'instruction MAC. La description du processeur doit permettre de réaliser l'ordonnancement des opérations de la DIR. Par conséquent, chaque opération de la DIR (nœuds du DFG) doit avoir au moins un modèle de réalisation dans la description du processeur.

Pour décrire le jeu d'instructions du processeur cible nous utilisons un modèle comportemental simplifié basé sur une liste **d'opérations de base** représentant toutes les implémentations possibles d'un nœud du graphe de flots de données. La Figure 69 montre la forme générale du fichier de description du processeur cible utilisé par notre estimateur.

```
Proc_Name: nom_du_processeur

Short_immediate: entier
Resources : res0, res1, res2, …
Res_Class
{
res_class1 = (res3, res4);
}

Register : reg0, reg1, reg2, …
Reg_Class
{
reg_class1 = (reg0, reg1, reg2);
reg_class2 = (reg3, reg4);
}

Virtual : virt0, virt1, virt2, …
Virt_Class
{
virt_class1 = (virt0, virt3);
}

Opération : Addition
{
[op_base1]
[op_base2]
..
}

Opération : Incrément
{
[op_base1]
[op_base2]
..
}
.
.
.
```

Figure 69 : Structure générale du fichier de description du processeur cible

Le point de départ d'une description est un ensemble de définitions spécifiant

- Le nom du processeur (*Proc_Name*),
- Le nombre de bits pour un entier court (*Short_immediate*),
- Les ressources mémoires et opératoires (*Resources*),
- Les ressources registres (*Register*),
- Et les éléments dits virtuels (*Virtual*).

Ces définitions sont complètement indépendantes de l'algorithme d'ordonnancement ou de l'annotation du DAG, ce qui offre une grande souplesse d'utilisation favorisant l'exploration architecturale. La description se poursuit par la définition d'un ensemble d'opérations (e.g. addition, soustraction, incrément, etc.) spécifiant toutes les implémentations possibles sur le processeur cible, i.e. une liste d'opérations de base.

6.2.2.1. Les ressources du processeur

Lors de la définition des ressources du processeur, on distingue les registres des unités fonctionnelles et de la mémoire.

Les ressources opératoires et mémoires

Le mot clé « **Resources** » permet de définir l'ensemble des ressources opératoires ou mémoires du processeur cible. Les ressources opératoires décrivent toutes les unités fonctionnelles dont dispose le processeur. Les ressources mémoires permettent de définir le nombre de bancs de mémoire de données disponibles (i.e. le nombre d'accès simultanés à la mémoire). Voici par exemple l'ensemble des ressources définies pour le OakDSPCore.

```
Resources : MEMX, MEMY, MULT, ALU, BS, CTRL, CG, ADR, EXP;
```

- MEMX et MEMY représentent les deux bancs de mémoire de données dont dispose le OakDSPCore, respectivement XRAM et YRAM.
- MULT représente le multiplieur.
- ALU représente l'unité arithmétique et logique.
- BS (Barrel Shifter) décrit la présence d'une unité de décalage.
- CTRL désigne l'utilisation d'une opération de contrôle du processeur.
- CG (Constant Generator) représente l'unité permettant de générer des valeurs immédiates. Dans le cas du OakDSPCore, une instruction peut prendre deux cycles si la valeur immédiate ne peut être codée sur 8 bits (valeur supérieure à 127 ou inférieure à –128). Dans le cas contraire on dit que la valeur est un entier court (short). Cette information est fournie dans la description du processeur grâce au mot clé « **Short_immediate** » qui précise le nombre de bits d'une valeur entière courte :

```
Short_immediate = 8
```

- ADR désigne la ressource permettant de générer une adresse. Cette ressource est très utilisée en début de boucle lors de l'initialisation de pointeurs par exemple.
- EXP représente enfin une unité spécialisée de détection d'exposant.

Voici à présent l'ensemble des ressources définies pour le PalmDSPCore.

```
Resources : MEMX0, MEMX1, MEMY0, MEMY1, MUL0, MUL1, ALU0, ALU1, BS,
            CTRL, CG, ADR, EXP ;
```

Comme le PalmDSPCore possède une architecture de type dual-MAC, deux multiplieurs (MUL0 et MUL1) et deux unités arithmétiques et logiques (ALU0 et ALU1) sont définis. La mémoire est décomposée en deux bancs de mémoire XRAM (MEMX0 et MEMX1) et deux bancs de mémoire YRAM (MEMY0 et MEMY1). Finalement on y trouve les mêmes

unités fonctionnelles que celles définies pour le OakDSPCore (BS, CTRL, CG, ADR et EXP).

Les ressources registres

Le mot clé « **Register** » permet de spécifier de manière simplifiée l'ensemble des registres utilisés par le processeur. Voici la description fournie pour le OakDSPCore.

```
Register  : X, Y, P, ACC, RI, RJ, RR;
```

- X et Y représentent les registres d'entrée du multiplieur.
- P est le registre à la sortie du multiplieur.
- ACC désigne un des accumulateurs.
- RI et RJ définissent les registres d'adresse utilisés lors d'un accès en mémoire via un mode d'adressage indirect.
- RR regroupe l'ensemble de tous les autres registres du DSP (ce n'est pas une classe).

Les ressources mémoires, opératoires et registres définies dans la description du processeur cible constituent les éléments de la table d'allocation utilisée lors de l'ordonnancement.

Les classes de ressources

Classes de ressources mémoires ou opératoires

Le langage de description permet de définir des classes regroupant des ressources dont le comportement est identique pour un sous-ensemble d'opérations de base. Grâce au mot clé « **Res_Class** » on définit des classes de ressources opératoires ou mémoires. Dans le cas du OakDSPCore, une seule classe a été définie regroupant les deux bancs de mémoire de données.

```
Res_Class
{
MEM = (MEMX, MEMY);
}
```

L'utilisation de classes est beaucoup plus utile pour le PalmDSPCore qui dispose de plusieurs unités fonctionnelles similaires et plusieurs bancs de mémoire. Voici les classes définies pour ce processeur :

```
Res_Class
{
MEM = (MEMX0, MEMX1, MEMY0, MEMY1);
MEMX = (MEMX0, MEMX1);
MEMY = (MEMY0, MEMY1);
MUL = (MUL0, MUL1) ;
ALU = (ALU0, ALU1)
}
```

Classes de registres

Il est également possible de définir des classes de registres grâce au mot clé « **Reg_Class** ». Dans le cas du OakDSPCore on définit par exemple les classes *REG* et *REG1* comme suit :

```
Reg_Class
{
REG = (ACC, RR, P);
REG1 = (ACC, RR);
}
```

Prise en compte des classes lors de l'ordonnancement

Quand une classe est manipulée par la fonction « *peut être ordonnancée au cycle n* », l'ordonnanceur recherche une ressource libre parmi celles qui sont définies dans la classe. Pour la classe (MEM) définie pour le OakDSPCore par exemple, si la mémoire XRAM est déjà allouée on teste si la mémoire YRAM est libre.

Quand une classe est utilisée par la fonction « *mettre à jour la table d'allocation des unités fonctionnelles* », on alloue la ressource qui a été choisie parmi celles définies dans la classe.

6.2.2.2. Les modes d'adressage du processeur

Les différents modes d'adressage sont pris en compte dans la description du processeur cible. C'est une information indispensable pour effectuer des estimations précises de performance. En effet, pour le OakDSPCore et plus encore pour le PalmDSPCore, le parallélisme entre les opérations est fortement contraint par le type de mode d'adressage utilisé. L'utilisateur peut définir très simplement tous les modes d'adressage à la suite du mot clé « **Virtual** ». Comme le OakDSPCore et le PalmDSPCore possèdent les mêmes modes d'adressage, leur définition est identique.

```
Virtual   : RMi, RMix, RMsd, RMld, CV;
```

- RMi représente un mode d'adressage mémoire indirect
- RMix désigne un mode d'adressage mémoire indexé
- RMsd spécifie un mode d'adressage mémoire direct court
- RMld représente un mode d'adressage mémoire direct long

Les modes d'adressage du processeur ne représentent pas des ressources utilisables par une opération (d'où le terme *Virtual*). Ils sont utilisés pour annoter les arcs du graphe de flots de données après une opération de lecture en mémoire, ce qui permet d'indiquer le type d'accès effectué mais surtout de contraindre la sélection de l'opération qui suit cet accès mémoire. On définit également dans ce champ une valeur entière CV[44] issue du générateur de constante CG.

De la même manière que pour les ressources du processeur, il est possible de définir dans une classe les modes d'adressage pour lesquels le comportement d'un sous ensemble

[44] Constant Value

d'opérations est identique. L'exemple suivant illustre les classes obtenues pour le OakDSPCore (le PalmDSPCore ne possède que la classe RMd) :

```
Virt_Class
{
RMd = (RMsd, RMld);
RMslx = (RMix, RMsd, RMld);
RM = (RMi, RMix, RMsd, RMld);
}
```

RMd correspond ainsi à la classe des modes d'adressage mémoire direct (long et court).

6.2.2.3. Les opérations de base

A chaque nœud du graphe de flots de données correspond un schéma de calcul d'exécution, ou **opération de base**, défini par un nom (e.g. « add », « sub » ou « mpy ») et des informations qui permettent de représenter (Figure 70) :

- les contraintes sur l'utilisation des ressources mémoires ou opératoires (UT_{libre}),
- les contraintes sur l'utilisation des registres (REG_{libre}),
- le nombre de cycles nécessaires à l'exécution de l'opération (Nb_Cycle),
- la possibilité d'exécuter conditionnellement ou non l'opération de base (Cond).

Comme indiqué sur la ligne (1) de la Figure 70 chaque opération de base est caractérisée par ses opérandes en **Entrées**, une ressource opératoire ou mémoire (**ut**[45]) et par ses opérandes de **Sorties**. Dans (2) et (3), on définit des contraintes qui sont utilisées lors de l'ordonnancement de l'opération de base. On ne décrit pas les effets a posteriori de l'exécution de l'opération mais les ressources dont elle a besoin pour s'exécuter. UT_{libre} (respectivement REG_{libre}) représente ainsi toutes les ressources mémoires et opératoires (respectivement les registres) qui doivent être libres pour ordonnancer l'opération. UT_{libre} et REG_{libre} permettent de décrire les contraintes sur les chemins de données mais également les contraintes liées à l'encodage du jeu d'instructions. Il est par exemple impossible d'ordonnancer dans le même cycle une opération OU logique avec une multiplication bien que les chemins de données le permettent. C'est donc une contrainte imposée par l'encodage du jeu d'instructions. Il est enfin possible de définir des contraintes utilisant des classes de ressources.

```
Operation :  Nom
[
{entrées} ; ut ; {sorties}                    (1)
UT_libre      =    {ut}                       (2)
REG_libre     =    {reg / reg ∈ Register }    (3)
Nb_Cycle      =    entier                     (4)
Cond          =    yes | no                   (5)
]

où

entrées ∈ (Register ∪ Virtual)
sorties ∈ (Register ∪ Virtual)
ut ∈ Resources
```

Figure 70 : Modèle de description d'une opération de base

[45] Unité de Traitement

Afin de supporter des instructions multi-cycles, on définit pour chaque opération de base le nombre de cycles (4) nécessaire à son exécution sur le processeur cible. Si plusieurs opérations sont ordonnancées simultanément, le nombre de cycles correspondant est égal à la valeur maximum du nombre de cycles de l'ensemble des opérations. Enfin, le champ Cond (5) permet de spécifier si l'opération peut être exécutée conditionnellement ou non.

Comme le montre la Figure 69, à une même opération (i.e. un même motif) peut correspondre plusieurs opérations de base (i.e. schéma d'exécution) suivant le type des opérandes ou les contraintes sur les ressources. Lors de la recherche d'une opération de base, l'algorithme d'ordonnancement parcourt la liste de toutes les opérations de base. Dès qu'une d'elles correspond au motif du nœud du graphe, elle est ordonnancée si les ressources dont elle a besoin sont disponibles. Dans le cas contraire, on essaie d'ordonnancer une autre opération de base. Finalement si aucune d'elles n'a un champ *Entrées* équivalent au motif issu du graphe, le choix se porte sur l'opération de base dont les *Entrées* sont les plus proches du motif. L'ordre d'apparition des opérations de base dans le fichier de description du processeur est par conséquent important. On définit en premier lieu les opérations de base susceptibles d'offrir les meilleures opportunités de parallélisme (i.e. possédant le moins de contraintes).

6.3. Processus d'estimation

L'algorithme ordonnance un nœud du graphe de flots de données (DIR) :

- quand il existe une opération de base correspondant au motif associé à ce nœud,
- quand le ou les opérandes de sortie et les ressources (ut, UT_{libre} et REG_{libre}) nécessaires à son exécution sont disponibles dans la table d'allocation.

Lorsque plus aucune tâche de la liste L ne peut être ordonnancée au cycle courant, on considère que l'ensemble des opérations de base allouées dans ce cycle forme une instruction du processeur cible. Cette hypothèse résulte de la validité de la description des contraintes sur les ressources issues de la description du processeur cible. Le processus d'estimation se poursuit en rendant libre l'ensemble des ressources de la table d'allocation et en passant au cycle suivant. Ceci serait à faire évoluer dans le cas d'architecture comportant plus d'étages de pipeline[46].

6.3.1. L'ordonnancement d'une instruction de type MAC

Reprenons l'exemple du graphe de flots de données du calcul d'un papillon de FFT (Figure 71).

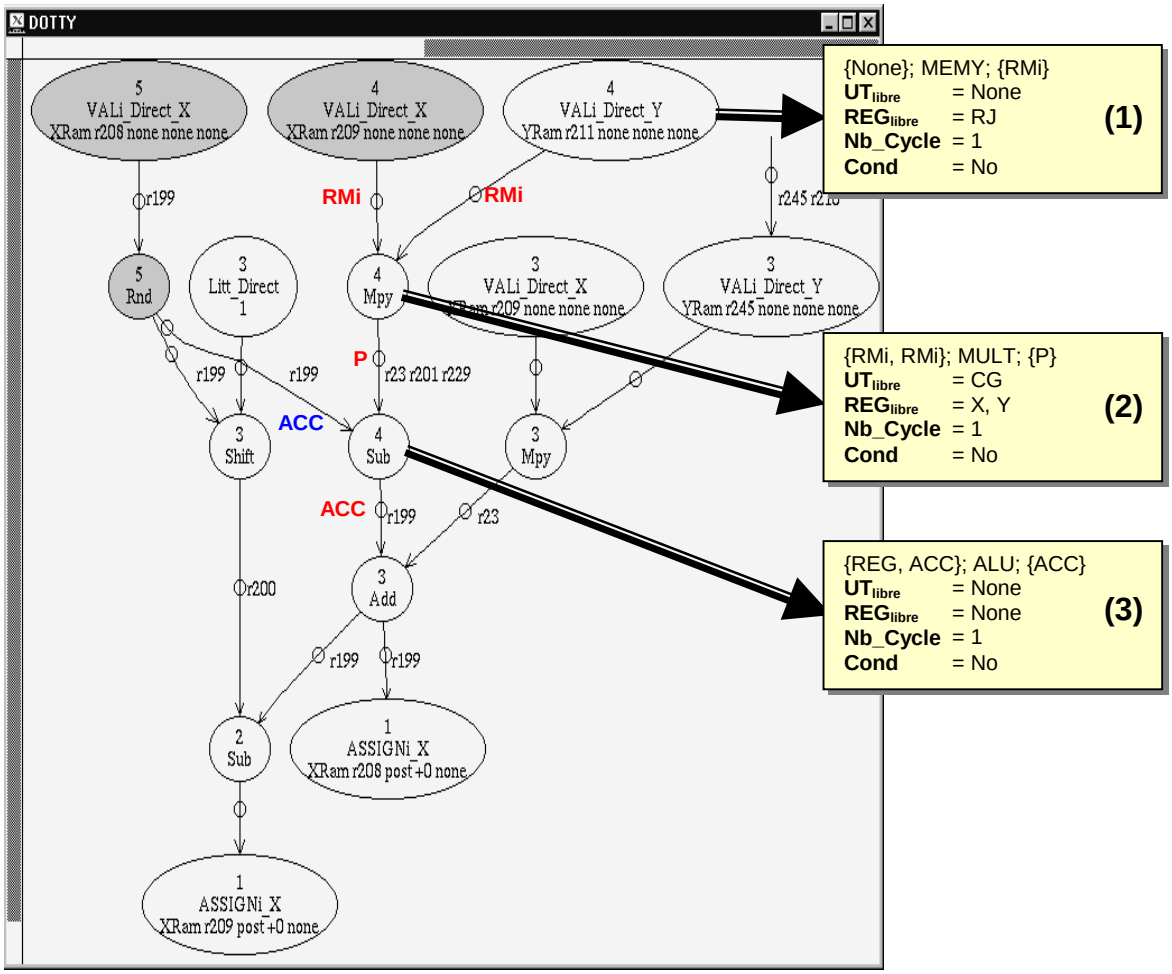


Figure 71 : Processus d'estimation

[46] Ce n'est pas une difficulté majeure.

Considérons que tous les nœuds de niveau 5 ont été ordonnancés ainsi que le nœud « VALi_Direct_X » de niveau 4 (nœuds grisés sur le graphe). Le prochain nœud dans la liste L des tâches ordonnançables est le nœud « VALi_Direct_Y », également de niveau de priorité égal à 4. L'algorithme d'ordonnancement manipule pour chaque nœud du graphe un motif constitué par le nom de l'opération et de ses opérandes en entrée. Ainsi pour le nœud « VALi_Direct_Y » le motif correspondant est :

```
Nom     :   VALi_Direct_Y
Entrées :   None
```

Comme le nœud du graphe ne possède aucun opérande en entrée, on affecte « None » au champ *Entrées*. Un motif est utilisé pour sélectionner une opération de base dans le fichier de description du processeur. Aucune autre information (nom et opérandes d'entrée) n'est utilisée dans la recherche d'une opération de base. Un nœud du graphe possède obligatoirement une opération de même nom dans la description du processeur. Il se peut néanmoins que les entrées ne soient pas définies dans la description du processeur cible pour cette opération, ce qui signifie qu'il n'existe pas d'instructions permettant d'exécuter cette opération en un seul cycle. Pour traiter ce cas, nous avons défini la notion de **pénalité** que nous détaillerons plus tard à l'aide d'un exemple. Dans le cas contraire, c'est-à-dire quand il existe une opération de base dont les entrées correspondent à celles du motif, l'ordonnancement de l'opération de base est effectué si les ressources définies par « ut », UT_{libre}, REG_{libre} et « sorties » sont libres. L'opération de base (1) de la Figure 71 correspondant au motif défini précédemment est :

```
{None}; MEMY; {RMi}
UT_libre    = None
REG_libre   = RJ
Nb_Cycle = 1
Cond     = No
```

Pour ordonnancer cette opération de base, les ressources MEMY et RJ doivent être libres. Il n'y a aucune autre contrainte puisque UT_{libre} à la valeur « None » et que la sortie (RMi) ne correspond à aucune ressource de la table d'allocation. Avant de passer au nœud suivant du graphe,

- la fonction « *mettre à jour la table d'allocation* » va allouer les ressources *ut*, *sorties*, UT_{libre} et REG_{libre} définies par l'opération de base, ici MEMY et RJ,
- le ou les arcs sortants du nœud sont annotés par la valeur du champ *Sorties*, ici RMi, ce qui permet de spécifier le type des opérandes d'entrées du nœud suivant.

L'allocation du nœud « VALi_Direct_Y » rend ordonnançable le nœud « Mpy » de niveau de priorité égal à 4. Cette opération effectue une multiplication entre deux données lues en mémoire XRAM et YRAM via un mode d'adressage indirect. Le motif correspondant à ce nœud est le suivant :

```
Nom     :   Mpy
Entrées :   RMi, RMi
```

Après une recherche dans le fichier de description du processeur, l'opération de base correspondant (2) à ce motif est la suivante :

```
{RMi, RMi}; MULT; {P}
UT_libre    = CG
REG_libre   = X, Y
Nb_Cycle = 1
Cond     = No
```

Pour ordonnancer cette opération, les ressources MULT, CG, X, Y et P doivent être libres au cycle courant. Dans cet exemple, l'algorithme d'ordonnancement tente d'ordonnancer cette opération de base dans le même cycle que les deux opérations de lecture en mémoire XRAM et YRAM. Si l'on fait l'hypothèse que les ressources sont libres, l'opération est supprimée de la liste L, les ressources MULT, CG, X, Y et P sont allouées dans la table d'allocation et l'arc sortant est annoté par « P », indiquant le type d'opérande d'entrée pour le nœud successeur (Sub). Comme le nœud « Rnd » de niveau 5 a déjà été ordonnancé, le nœud « Sub » de niveau 4 est ordonnançable (tous ses prédécesseurs ont rendu leur résultat). Le motif associé à ce nœud est le suivant :

```
Nom     :   Sub
Entrées :   ACC, P
```

L'opération de base correspondant (3) à ce motif est la suivante puisque REG est une classe de registre incluant P :

```
{REG, ACC}; ALU; {ACC}
UT_libre    = None
REG_libre   = None
Nb_Cycle = 1
Cond     = No
```

Il est possible d'allouer cette opération si les ressources ALU et ACC sont disponibles au cycle courant. Comme UT_{libre} et REG_{libre} ont la valeur « None », il n'y a pas d'autres contraintes pour ordonnancer cette opération de base.

Avec l'ordonnancement dans le même cycle de deux lectures en mémoire (VALi_Direct_X et VALi_Direct_Y), d'une opération de multiplication (Mpy) et d'une soustraction (Sub), une instruction « MSU[47] » est trouvée puisqu'aucune autre opération ne peut être ordonnancée dans le même cycle.

L'exemple précédent fait apparaître l'importance de déterminer précisément les contraintes sur les ressources pour estimer au mieux le nombre d'instructions et de cycles d'exécution. Nous détaillons ces aspects dans le paragraphe suivant.

[47] Multiply and SUbtract

6.3.2. Détermination des contraintes

Les champs UT_{libre} et REG_{libre} permettent de contraindre le parallélisme entre opérations. La détermination de ces champs s'effectue en premier lieu sur les opérations les plus contraignantes, i.e. celles offrant peu ou pas de parallélisme. Pour le OakDSPCore par exemple, les instructions de contrôle n'offrent que très peu de parallélisme avec les autres opérations de base. Prenons par exemple l'opération « Bkrep » représentant une opération de boucle matérielle dont une implémentation possible est la suivante :

```
{RR}; CTRL; {None}
UT_libre    = ALL
REG_libre   = ALL
Nb_Cycle = 2
Cond     = No
```

Pour cette opération de base, UT_{libre} et REG_{libre} ont la valeur « ALL », ce qui signifie que toutes les ressources de la table d'allocation doivent être libres pour ordonnancer cette tâche au cycle courant. En d'autres termes, cette opération de base ne peut être ordonnancée avec aucune autre, ce qui correspond finalement à une instruction du processeur cible. Dans ce premier exemple le registre RR représente le nombre d'itérations de la boucle. Il est néanmoins possible pour une instruction « Bkrep » de spécifier directement le nombre d'itérations via une valeur immédiate. Dans ce cas, le graphe de flots de données simplifié correspondant est représenté par la Figure 72. L'ordonnancement dans le même cycle des deux nœuds de ce graphe permet de retrouver l'instruction de boucle correspondante.

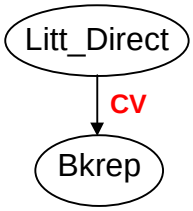


Figure 72 : Graphe de flots de données pour une boucle matérielle

L'opération de base correspondant au nœud « Litt_Direct » est la suivante :

```
{None}; CG; {CV}
UT_libre    = None
REG_libre   = None
Nb_Cycle = 1
Cond     = No
```

Pour ordonnancer cette opération de base, seule la ressource CG est allouée (puisque CV ne correspond à aucune ressource de la table d'allocation). L'opération de base correspondant au nœud « Bkrep » est :

```
{CV}; CTRL; {None}
UT_libre    = ALL – (CG)
REG_libre   = ALL
Nb_Cycle = 2
Cond     = No
```

Comme il existe une instruction de boucle matérielle correspondant à l'ordonnancement de ces deux opérations en parallèle, il est nécessaire pour l'opération de base « Bkrep »

d'omettre dans UT_{libre} la ressource CG allouée par le nœud prédécesseur « Litt_Direct ». On utilise pour cela une syntaxe particulière, « ALL – (CG) », qui signifie que toutes les ressources, sauf CG, doivent être libres pour ordonnancer l'opération de base. Cette syntaxe peut aussi s'utiliser pour le champ REG_{libre} et est très utile pour décrire une opération de base utilisant *presque toutes* les ressources du processeur. Notons qu'il est aussi possible pour ces deux champs de fournir une liste de ressources (e.g. UT_{libre} = MUL, ALU, EXP).

Finalement si les deux opérations de la Figure 72 sont ordonnancées en parallèle au cycle n, l'ordonnancement de la prochaine instruction commencera au cycle n+2 puisque le nombre de cycles maximum entre les deux opérations est deux.

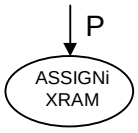
6.3.3. Notion de pénalité

L'algorithme d'ordonnancement par liste tel qu'il a été défini (Figure 64), tente tant qu'il est possible d'allouer les tâches de la liste L au cycle courant. Quand aucune des opérations de la liste L ne peut être ordonnancée dans le cycle courant on passe au cycle suivant.

La notion de pénalité permet d'affiner l'estimation en prenant en compte les nœuds du graphe pour lequel il n'existe pas d'opération de base dont les opérandes d'entrée correspondent à celles des arcs entrants du nœud du graphe. La raison peut venir soit de l'encodage du jeu d'instructions, soit des contraintes sur les chemins de données, soit des choix effectués par l'ordonnancement des nœuds prédécesseurs. Lors de la programmation d'un DSP, ceci se traduit généralement par des transferts de données supplémentaires de manière à acheminer la donnée vers la bonne unité de traitement. Un exemple typique avec le OakDSPCore est le transfert du registre P (résultat de la multiplication) en mémoire. Il n'existe pas de chemins de données permettant d'effectuer directement l'écriture (i.e. en un cycle) de ce registre en mémoire. Le code assembleur correspondant est donc le suivant :

```
mov   p, a0       ;écriture du registre P dans l'accumulateur a0
mov   a0l, (r0)   ;écriture de a0 dans la mémoire via un mode
                   d'adressage indirect
```

Le graphe flot de données (simplifié) correspondant à ce type de transfert est le suivant :



Dans la description du processeur cible il existe une opération de base effectuant un transfert vers la mémoire :

```
{REG1}; MEM; {None}
UT_libre    = ALL
REG_libre   = ALL
Nb_Cycle = 1
Cond     = No
```

Toutefois, la classe *REG1* ne contient que deux types de registres, RR et ACC. Il n'existe donc pas d'opération de base *ASSIGNi* avec comme opérande d'entrée le registre P. L'ordonnancement de ce nœud engendre donc une pénalité correspondant au transfert du

registre P dans un des accumulateurs avant l'écriture en mémoire. Si l'opération de base décrite précédemment est ordonnançable au cycle courant, la pénalité est égale au temps nécessaire pour acheminer la donnée (1 cycle), sinon elle est équivalente au temps nécessaire pour libérer les ressources et acheminer les données (2 cycles). Notons enfin qu'après une pénalité toutes les ressources de la table d'allocation sont libres.

6.3.4. Traitements post-ordonnancement

6.3.4.1. Traitement des instructions conditionnelles

Chaque opération de base possède un champ « Cond » permettant de spécifier s'il est possible de l'exécuter conditionnellement. Comme nous l'avons vu dans le chapitre précédent, il est très important de retrouver les instructions conditionnelles pour effectuer des estimations de performance précises. On applique pour cela une règle après la phase d'ordonnancement, i.e. après le choix d'une opération de base. Si cette dernière est conditionnelle il est alors parfois possible d'appliquer la règle supprimant des instructions de branchement inutiles.

6.3.4.2. Traitement des boucles matérielles

Pour le OakDSPCore et le PalmDSPCore, il existe une instruction de boucle matérielle « rep » qui s'utilise pour répéter une seule instruction. Or, le graphe de flots de données ne possède que des opérations de boucle du type « Bkrep », qui répètent un bloc d'instructions. Comme le « rep » prend un cycle pour s'exécuter et le « Bkrep » deux cycles, il est intéressant de retrouver (i.e. d'estimer) les « rep » de l'application. Pour cela il est nécessaire de vérifier que le nombre d'instructions à l'intérieur de la boucle est égal à 1. C'est pour cette raison que le traitement correspondant ne peut s'effectuer là aussi qu'après la phase d'ordonnancement.

6.3.5. Conclusion

En conclusion, l'algorithme d'ordonnancement est capable de tenir compte du parallélisme offert par les chemins de données du DSP, mais aussi de prendre en compte les restrictions de parallélisme dues à l'encodage du jeu d'instructions (modèle ISA du DSP). Ces restrictions sont exprimées sous la forme de contraintes sur l'utilisation en parallèle de classes de ressources. Cette approche est adaptée aux architectures DSP récentes intégrant par exemple des unités dual-MAC. Nous limitons cependant l'ordonnancement des opérations à l'intérieur d'un bloc de base. Des techniques de génération de code qui font appel à des traces [Ell85] ont été proposées pour augmenter le parallélisme entre instructions. Pour le OakDSPCore, le faible parallélisme entre instructions ne justifie pas une approche inter bloc de base. Le PalmDSPCore possède quant à lui beaucoup plus de parallélisme, ce qui pourrait justifier l'étude de techniques d'ordonnancement opérant sur plusieurs blocs de base. Toutefois, le problème de la taille du code obtenu doit être pris en compte. Enfin, la description du processeur cible est rapide et permet, malgré les simplifications du modèle d'architecture, d'obtenir des estimations de performance précises. C'est ce que nous allons montrer dans la prochaine partie.

Résultats

Cette partie présente les résultats obtenus pour différentes applications en utilisant l'outil d'estimation, VESTIM, développé au cours de cette thèse. Dans un premier temps, nous décrivons brièvement le principe de fonctionnement de ces applications ainsi que leur usage actuel ou futur dans le monde industriel. Puis, nous montrons comment l'utilisation de l'outil d'estimation permet de guider le programmeur dans l'optimisation du code C [Peg99c]. Pour cela, nous détaillons les résultats fournis par VESTIM en utilisant G.728 [Rec94] comme application test. Puis, pour chaque application nous décrivons les modifications successives apportées au code C (permettant une amélioration significative des performances) et les performances correspondantes fournies par l'outil. Enfin, en conclusion nous montrons la stabilité des estimations par rapport aux différentes versions du code C mais aussi les limitations de l'outil.

7.1. Description des applications tests

Voici l'ensemble des applications qui ont été testées en utilisant l'outil d'estimation VESTIM :

- Transformée de Fourier rapide (FFT)
- G.728 [Rec94]
- G.723.1 [Rec96]
- G.729 [Rec95]
- V22bis [Ste91]
- AC3 (décodeur seulement) [Ac3]

La transformée de Fourier rapide a déjà été décrite dans le paragraphe 2.2.2.3.

7.1.1. G.728

La norme G728 est un algorithme de compression de la parole à 16 kbits/s (taux de compression de 4) utilisant un LD-CELP[48] comme modèle de codeur. Normalisée en 1992, elle est écrite à l'origine pour des processeurs DSP à virgule flottante. En 1994, l'ITU édite l'Annexe G qui est la version pour les processeurs DSP à virgule fixe. Contrairement à la norme G.721 par exemple, l'algorithme de compression utilisé dans G.728 est spécifique au signal de parole (d'où le terme *vocoder* pour *voice coder*). L'algorithme est en effet basé sur les mécanismes de production de la parole et utilise les propriétés de perception de l'oreille humaine (phénomène de masquage) pour ne transmettre que des données audibles. La norme G.728 travaille sur des trames de quatre vecteurs de cinq échantillons chacun et la seule donnée transmise est le meilleur index du dictionnaire d'excitation. Afin d'éviter l'utilisation de techniques de suppression d'écho de ligne coûteuses en temps, G.728 possède un temps de reconstruction (codage/décodage) inférieur à 5ms (Low Delay).

La norme G.728 est utilisée dans les téléphones numériques et est compatible avec la norme H.320 (partie vidéo de la visioconférence).

7.1.2. G.723.1

La recrudescence des services multimédia sur les réseaux numériques et analogiques nécessite l'utilisation de codeurs très bas débit. L'algorithme de compression de parole G723.1[49] [Rec96], lui même inclus dans les normes de vidéo-conférence H.324 a été

[48] **L**ow **D**elay **C**ode **E**xcited **L**inear **P**rediction
[49] True Speech

normalisé par l'ITU en 1996. Pour cet algorithme, l'organisme de normalisation fournit un code C et des séquences de test. Le codeur de parole possède deux taux de compression : 5.3kbits/s et 6.3kbits/s. Le taux le plus haut est bien sûr de meilleur qualité. Il est possible de changer de taux d'échantillonnage à chaque trame. Dans le cas d'une compression à 6.3kbits/s, le signal résiduel est quantifié suivant une méthode dite MP-MLQ, alors que dans le cas d'une compression à 5.3kbits/s, c'est une méthode ACELP[50] qui est utilisée. L'algorithme travaille sur des trames de 240 échantillons (soit 30ms) échantillonnées à 8kHz.

La norme G.723.1 est le standard de codage de parole pour les applications de type « Voice over IP », c'est à dire aux applications Internet, en particulier la visioconférence et les téléphones vidéo. Il est à noter que l'utilisation de cet algorithme nécessite le paiement de *royalties* à plusieurs entreprises et laboratoires tels que DSP Group, Lucent Technology ou le CNET.

7.1.3. G.729A

La recommandation G729 [Rec95] normalisée par l'ITU en 1995 est un algorithme de compression de parole à 8 kbit/s basé sur un codage CS ACELP[51]. Le codeur travaille dans une première partie sur des trames de 10 ms à un taux de 8kbits/s, pour se concentrer ensuite sur des sous-trames d'une durée de 5ms. L'annexe A de la recommandation (G.729A) décrit une variante moins complexe de la norme. La complexité de l'algorithme est en effet réduite de 50% tout en préservant une totale interopérabilité avec l'algorithme de base (un signal de parole codé par G729 peut-être décodé par G729A et vice versa). La norme G729A peut être ainsi utilisée à la place de G.729 quand une réduction forte de complexité est nécessaire dans les équipements terminaux.

Le faible débit et la qualité du codeur font que la norme G.729 est très intéressante pour les applications sans fil ou la téléphonie sur Internet. Bien que G.729 fut spécifiquement recommandée pour les applications multimédia DSVD[52], le codeur ne se limite pas à ce type d'applications. La faible complexité et le court délai de G729A en font un choix attractif pour de telles applications comparé à G723.1.

7.1.4. V22bis

La norme V22bis [Ste91] est le standard modem pour une connexion à 2400 bit/s. V22bis effectue une conversion numérique-analogique (modulation) et analogique-numérique (démodulation). Son principe de fonctionnement est basé sur une modulation d'amplitude de deux porteuses en quadrature (MAQ[53]). Il existe une porteuse pour la transmission et une porteuse pour la réception. En modulation, l'algorithme traite des données numériques par paquets de 4 bits et transmet donc 600 symboles par seconde. La démodulation consiste à retrouver le point de la constellation correspondant au signal analogique reçu. Pour cela, un calcul de distance euclidienne est effectué entre les différents points de la constellation et le point représentant le signal reçu.

La norme V22bis est très utilisée aujourd'hui surtout du fait de la faible complexité de l'algorithme.

[50] Asynchronous CELP
[51] Conjugate Structure Algebraic Code Excited Linear Prediction
[52] Digital Simultaneous Voice and Data
[53] **M**odulation **A**mplitude **Q**uadrature

7.1.5. Le décodeur AC-3

Le système numérique AC-3 développé par les laboratoires Dolby, compresse le signal à des débits binaires compris entre 32 kbit/s (pour un simple canal monophonique) et 640 kbit/s. Dans la pratique, la technique Dolby AC-3 est adaptée au codage et à la compression de cinq canaux sonores. Le système offre des voies gauche, droite et centrale à l'avant de l'auditeur, ainsi que deux canaux *surround* droit et gauche totalement indépendants, permettant une localisation plus précise des sons et une ambiance plus réaliste. Un sixième canal contenant des informations additionnelles dans le domaine des sons graves permet de renforcer certains effets (e.g. explosion). Ce canal a été baptisé « .1 », le système à six canaux étant alors dénommé « 5.1 ». Pour le format « 5.1 », le débit binaire associé est de 384 Kbit/s (taux de compression de 12). Pour parvenir à un tel résultat, Dolby a mis au point une technique d'allocation binaire originale en utilisant au maximum les propriétés psychoacoustiques de l'oreille humaine, en particulier les phénomènes de masquage. Cette technique permet de ne pas transmettre les composantes spectrales inaudibles, c'est-à-dire les coefficients dont l'énergie est inférieure au « seuil perceptuel » (i.e. la courbe de masquage). Par contre, les coefficients dont l'énergie est largement supérieure à ce seuil se voient attribuer un nombre de bits maximum. Le modèle perceptuel se trouve dans l'encodeur et le décodeur ce qui permet de consacrer l'intégralité du flux binaire aux données audio. Par ailleurs, l'AC-3 appartient à la famille des algorithmes de codage audio par transformée. Après échantillonnage, le signal est découpé en trames de quelques millisecondes sur lesquelles est appliquée une transformation temps-fréquence TDAC[54], variante de la transformée de Fourier, qui délivre pour chaque bande spectrale un jeu de coefficients représentatifs du spectre du signal. Le décodeur accomplit une transformation inverse sur ces coefficients quantifiés pour revenir dans le domaine temporel de départ.

Bien qu'initialement imaginée en réponse au besoin de la télévision à haute définition, la première réalisation du procédé AC-3 a été effectuée pour répondre aux besoins du cinéma (Star Trek VI en 1991). L'ATSC[55] a adopté le système de compression audio AC-3 pour la télévision haute définition aux Etats-Unis. Celui-ci a depuis trouvé d'autres domaines d'applications comme le DVD[56].

7.1.6. Complexité des applications tests

Afin de donner un ordre d'idée de la complexité des applications utilisées, on fournit dans la Table 9, le nombre total de lignes de code C ainsi que le pourcentage en taille de code responsable d'environ 80% du temps de calcul total.

Application	Nombre de lignes de Code C	Temps d'exécution (en %)	Taille de code (en %)
G.728 (encodeur)	1,1 k	78	29
G.723.1	2,7 k	79	15,6
G.729	5,5 k	80	28
AC-3 (décodeur)	4,2 k	83	22
V22bis	10,1 k	78	5,6

Table 9 : Complexité des applications tests

[54] Time Domain Alias Cancellation
[55] Advanced Television Systems Committee
[56] Digital Video Disc

Cette table montre que l'hypothèse selon laquelle 80% du temps de calcul est localisée dans 20% de la taille totale du code est à peu près respectée (même si l'on observe quelques disparités). L'intérêt de localiser puis optimiser les parties critiques de l'application est donc évident.

7.2. Utilisation de l'outil d'estimation

L'outil d'estimation VESTIM est basé sur les logiciels du projet GNU : *gcc*, *flex*, *bison*, *gmake*, etc. L'outil est ainsi disponible sous UNIX comme sous Windows95/98/NT et son utilisation s'effectue de manière identique sous ces différentes plates-formes. VESTIM utilise le programme *make* pour éviter des compilations inutiles.

7.2.1. Création d'un projet

Avant l'appel à VESTIM, l'utilisateur peut créer automatiquement le *makefile* de son application grâce au programme « *mkcreate* » dont les arguments sont les fichiers sources en langage C de l'application. Sur l'exemple suivant, tous les fichiers écrit en langage C dans le répertoire courant sont passés en paramètre au programme.

Exemple:

```
mkcreate *.c
```

Le programme « *mkcreate* » produit un fichier « makefile » exécutable grâce à la commande « make » et dont la syntaxe est la suivante :

```
$> make [paramètres] [Options]
```

7.2.2. Appel du programme VESTIM

Voici ce qui s'affiche à l'écran si aucun paramètre n'est passé au programme « *make* »:

```
Usage :
        Build tree structure.............make 0
        Display version number...........make 1
        Save C source files..............make 2
        GCC compilation..................make 3
        Dynamic annotation...............make 4
        Check 'ifdef OAK'................make 5
        OCC compilation..................make 6
        ARM compilation..................make 7
        OAK RTL parsing..................make 8
        OAK LST parsing..................make 9
        Build OAK stats files............make 10
        ARM estimation...................make 11

        Whole OAK flow...................make oak
        Whole ARM flow...................make arm


        Clean OAK files..................make clean_oak
        Clean ARM files..................make clean_arm
        Clean all files..................make clean
```

Figure 73 : Messages à l'exécution de VESTIM

Comme le montre la Figure 73, suivant le ou les paramètres, on peut exécuter séparément les différentes phases de l'estimateur pour les processeurs OakDSPCore ou Arm7TDMI. Cependant, les trois paramètres les plus fréquemment utilisés sont :

- ✓ **oak :** analyse des performances pour le OakDSPCore
- ✓ **arm :** analyse des performances pour le processeur RISC Arm7TDMI[57]
- ✓ **clean :** suppression de tous les fichiers générés par VESTIM.

Le champ « *Options* » permet de spécifier par exemple le type de compilateur utilisé, les *flags* de compilation (niveau d'optimisation) ou encore le nom de l'exécutable. Les trois options les plus intéressantes de notre point de vue sont les suivantes.

- ✓ **ARGS**: arguments de l'exécutable de l'application (e.g. nom de la séquence de test).
- ✓ **TESTSEQLEN**: longueur de la séquence de test en seconde utilisée lors de l'exécution. Cette option permet de ramener les résultats en nombre de cycles par seconde.
- ✓ **PROC_DESC_FILE**: nom du fichier de description du processeur cible.

[57] L'outil ne fournit que les performances du code assembleur généré par le compilateur.

7.2.3. Analyse des résultats

Les résultats fournis par VESTIM sont des tables regroupant les évaluations de performance du code assembleur généré, les estimations d'un code assembleur optimisé et d'autres informations utiles pour l'optimisation du code. L'outil permet de visualiser les résultats à deux niveaux de granularité.

7.2.3.1. Niveau fonction

Comme le montre la Figure 74, les résultats sont triés par ordre décroissant du nombre de cycles par seconde (colonne LST). Chaque ligne correspond à une fonction de l'application pour laquelle les informations suivantes sont fournies :

- ✓ **Name** : nom de la fonction,
- ✓ **LST** : nombre de cycles par seconde pour exécuter la fonction en utilisant le code assembleur généré par le compilateur,
- ✓ **%** : pourcentage du nombre de cycles du code assembleur généré par rapport au nombre total de cycles de l'application,
- ✓ **C** : nombre de lignes de code C de la fonction,
- ✓ **ASM** : nombre de lignes du code assembleur généré,
- ✓ **Words** : nombre de mots mémoires programme[58].
- ✓ **RTL** : estimation du nombre de cycles par seconde pour exécuter la fonction en utilisant un code assembleur optimisé,
- ✓ **%** : pourcentage du nombre de cycles du code assembleur optimisé par rapport au nombre total de cycles estimé de l'application,
- ✓ **Delta** : différence relative entre le code généré par le compilateur (LST) et l'estimation d'un code assembleur optimisé (RTL).

Le **Delta** permet de localiser les fonctions dont le code assembleur généré est particulièrement inefficace par rapport à l'estimation d'un code assembleur optimisé. Son calcul prend en compte les temps d'exécution LST et RTL de la fonction considérée et le temps total d'exécution de l'application :

$$Delta = [(LST - RTL) / (Total_{LST})] * 100$$

Le fait de diviser la différence entre le code LST et RTL par le nombre total de cycles du code assembleur généré ($Total_{LST}$) permet de mieux repérer les fonctions qui ont une influence notable sur les performances globales de l'application. Si les performances entre le code généré LST et l'estimation d'un code optimisé RTL sont très proches et si la fonction n'a pas d'influence sur les performances globales, la valeur de *Delta* sera proche de zéro. Au contraire plus la fonction est coûteuse et plus le nombre de cycles du code LST est supérieur au code RTL, plus la valeur de *Delta* sera grande. Quand la valeur de *Delta* est grande, la fonction correspondante à une influence sur les performances et nécessite une optimisation du code C.

[58] Une instruction assembleur peut prendre plusieurs mots de la mémoire programme.

Name	LST	%	C	ASM	Words	RTL	%	Delta
bloc50m	**36558207**	**30.2**	**75**	**342**	**422**	**14781363**	**25.1**	**18.0**
bloc17m	**22159221**	**18.3**	**29**	**138**	**172**	**8186600**	**13.9**	**11.5**
hwmcore	**19827598**	**16.4**	**128**	**898**	**1053**	**14008109**	**23.8**	**4.8**
bloc14m	**7607170**	**6.3**	**18**	**54**	**65**	**3853610**	**6.5**	**3.1**
bloczirm	**6693897**	**5.5**	**53**	**270**	**346**	**3233977**	**5.5**	**2.8**
rnd	**6254250**	**5.2**	**5**	**5**	**6**	**4169500**	**7.1**	**1.7**
simpdiv	**6069965**	**5.0**	**100**	**21**	**22**	**2795870**	**4.7**	**2.7**
bloc9m	1990920	1.6	71	337	436	1014346	1.7	0.8
bloc37m	1859021	1.5	80	342	422	740813	1.3	0.9
bloc44m	1857291	1.5	77	342	422	734479	1.2	0.9
bloc49m	1382176	1.1	24	110	146	772016	1.3	0.5
bloc51m	1306435	1.1	12	101	118	585458	1.0	0.6
bloc4m	1253600	1.0	18	91	118	646400	1.1	0.5
vscale	1224761	1.0	39	198	215	774481	1.3	0.4
findnls	677881	0.6	29	150	158	272687	0.5	0.4
bloc19m	632000	0.5	11	75	101	128000	0.2	0.4
divide	573253	0.5	24	77	88	372065	0.6	0.2
bloc38m	378451	0.3	16	113	142	170772	0.3	0.2
bloc43m	366200	0.3	15	81	107	170000	0.3	0.2
bloc13m	365600	0.3	12	53	67	171200	0.3	0.2
bloc45m	356011	0.3	13	104	128	143678	0.2	0.2
bloc46m	355220	0.3	43	120	152	217625	0.4	0.1
bloc36m	284200	0.2	16	65	92	214800	0.4	0.0
main	266108	0.2	50	174	258	205258	0.3	0.0
vscalel	222036	0.2	29	120	128	117964	0.2	0.1
bloc12m	179400	0.1	20	100	121	60000	0.1	0.1
bloc11m	148000	0.1	10	25	35	107200	0.2	0.0
bloc16m	143200	0.1	10	46	63	136000	0.2	0.0
Reads	63249	0.1	8	19	27	59246	0.1	0.0
bloc96m	48800	0.0	7	23	32	43200	0.1	0.0
init	308	0.0	44	145	191	313	0.0	-0.0
Total	121104433	100.0	1086	4739	5853	58887030	100.0	51.4

Figure 74 : Résultats au niveau fonction pour G.728 (encodeur)

La Figure 74 montre pour l'encodeur de G.728 les performances au niveau fonction fournies par VESTIM. Cette table permet de localiser très rapidement les fonctions critiques de l'application. Sur cet exemple les fonctions *bloc50m*, *bloc17m*, *hwmcore* et *bloc14m* sont évidemment les fonctions à optimiser en priorité puisqu'elles représentent 71% du temps de calcul total de l'encodeur pour seulement 23% de la taille totale du code C[59]. Pour ces fonctions d'ailleurs, les valeurs de *Delta* sont les plus élevées par rapport aux autres fonctions de l'application.

Afin d'affiner l'analyse des résultats, VESTIM fournit également les performances LST et RTL au niveau des blocs de base de chaque fonction.

[59] Information obtenue grâce à la quatrième colonne de la table (C).

7.2.3.2. Niveau blocs de base

	LST					

Block	LST	% Cycles	Words	C	ASM	
B1	59200	0.3	37	37	4	27
B2	2252800	10.2	11	11	3	10
B3	3072000	13.9	3	3	2	3
B4	614400	2.8	3	3	2	3
B5	1843200	8.3	9	9	1	8
B6	5529600	25.0	9	8	1	8
B7	614401	2.8	1	1	1	1
B8	614400	2.8	1	1	0	1
B9	546036	2.5	6	5	2	4
B10	0	0.0	2	2	1	1
B11	5836800	26.3	28	27	2	24
B12	55440	0.3	8	8	3	6
B13	1024000	4.6	5	5	0	3
B14	28800	0.1	18	18	3	13
B15	16000	0.1	2	2	2	2
B16	6400	0.0	4	3	0	3
B17	3344	0.0	4	4	2	2
B18	27200	0.1	17	17	0	11
B19	15200	0.1	9	8	0	8
Total	22159221	100.0	178	172	29	138

	RTL	

Block	RTL	%
B1	22400	0.3
B2	614400	7.5
B3	1024000	12.5
B4	0	0.0
B5	204800	2.5
B6	614400	7.5
B7	2457600	30.0
B8	91006	1.1
B9	0	0.0
B10	1024000	12.5
B11	0	0.0
B12	1638400	20.0
B13	34650	0.4
B14	409600	5.0
B15	20800	0.3
B16	8000	0.1
B17	3200	0.0
B18	3344	0.0
B19	16000	0.2
Total	8186600	100.0

Figure 75 : Résultats au niveau bloc de base pour la fonction bloc17m

La Figure 75 montre les résultats au niveau bloc de base pour la fonction *bloc17m*. La table du haut représente les performances du code généré par le compilateur, alors que celle du bas donne pour chaque bloc de base une estimation d'un code assembleur optimisé. Grâce à ces résultats il est possible de localiser les lignes de code C les plus coûteuses dans la fonction *bloc17m*. Pour cette dernière, 75,4% du temps total d'exécution du code généré

est regroupé dans les blocs de base B_2, B_3, B_6 et B_{11}. Les résultats au niveau RTL montrent que la structure du programme (i.e. le CFG) n'est plus tout à fait la même suite à des optimisations effectuées par l'estimateur (e.g. la synthèse d'instructions conditionnelles). Ceci n'est cependant pas gênant pour le programmeur lors de l'analyse des résultats. Les blocs de base critiques au niveau RTL sont B_2, B_3, B_7, B_{10} et B_{12} avec un pourcentage cumulé de 82,5%. Il est bien évidemment possible de connaître pour chaque bloc de base le numéro de la ligne correspondante du code C ou du code assembleur généré par le compilateur.

7.3. Expérimentations avec l'application G.728

Dans ce paragraphe on propose d'étudier avec plus de détail, l'évolution des performances du code LST comme des estimations de niveau RTL en fonction des modifications apportées au code C pour l'encodeur de l'application G.728 sur le OakDSPCore. La Figure 74 représente les résultats obtenus à partir du code C d'origine, c'est-à-dire sans optimisation spécifique pour ce DSP.

7.3.1. La première phase d'optimisation

La table présentée sur la Figure 76 montre les résultats obtenus après cette première phase d'optimisation.

7.3.1.1. Des optimisations classiques

Les performances fournies par VESTIM (Figure 74) permettent de focaliser le travail d'optimisation sur les fonctions les plus critiques de l'application. Pour ces dernières le premier travail d'optimisation consiste à appliquer les règles de réécriture indépendantes du processeur cible qui sont principalement :

- la réduction de la durée de vie des variables,
- l'utilisation de pointeurs pour l'accès aux tableaux,
- et la localisation des données en mémoire XRAM ou YRAM.

Comme ces optimisations sont relativement rapides à implémenter, toutes les fonctions dépassant 1% du temps d'exécution global ont été modifiées.

Durant cette première phase, d'autres optimisations ont été effectuées. Pour la fonction « *simpdiv* » par exemple, les boucles « *do…while* » ont été transformées en « *for* ». On constate cependant que la transformation n'a aucun effet aussi bien sur le code généré LST que sur l'estimation de niveau RTL.

7.3.1.2. Des modifications algorithmiques

Pour la fonction « *hwmcore* » le code C a été modifié afin de simplifier la structure du programme. Le code d'origine comportait trois structures de boucles indépendantes qui ont été regroupées dans une seule structure limitant ainsi le nombre de variables utilisées. C'est une modification algorithmique qu'il est impossible d'anticiper pour l'estimateur. Pour cette raison, les estimations RTL sont affectées significativement par cette modification.

7.3.1.3. L'utilisation de macros du langage C

En comparant les tables des Figure 74 et Figure 76, on constate que la fonction « *rnd* » a disparu dans la seconde. Nous avons en effet remplacé cette fonction par une macro du langage C. Ceci est particulièrement intéressant quand la fonction est de petite taille et très fréquemment appelée. On évite en effet le coût associé à l'appel puis au retour de cette fonction. Le coût relatif au traitement lui-même est ajouté aux performances des fonctions où l'appel de la macro est réalisé. L'utilisation de macros fait bien sûr partie des modifications algorithmiques.

Name	LST	%	C	ASM	Words	RTL	%	Delta
bloc50m	**33671246**	**35.2**	**77**	**313**	**388**	**14782290**	**33.2**	**19.7**
bloc17m	**18791988**	**19.6**	**39**	**114**	**143**	**8594596**	**19.3**	**10.6**
hwmcore	**11540968**	**12.1**	**77**	**372**	**416**	**4201283**	**9.4**	**7.7**
bloc14m	**6270236**	**6.6**	**20**	**40**	**51**	**3699225**	**8.3**	**2.7**
simpdiv	**6069965**	**6.3**	**98**	**21**	**22**	**2795870**	**6.3**	**3.4**
bloczirm	**3434697**	**3.6**	**51**	**205**	**265**	**2337977**	**5.3**	**1.1**
bloc37m	**2175242**	**2.3**	**81**	**354**	**435**	**850783**	**1.9**	**1.4**
bloc44m	**2173456**	**2.3**	**80**	**354**	**435**	**845517**	**1.9**	**1.4**
bloc9m	2034015	2.1	71	339	439	1030346	2.3	1.0
bloc51m	1276585	1.3	12	101	117	585458	1.3	0.7
vscale	1224761	1.3	39	198	215	774481	1.7	0.5
bloc49m	1197996	1.3	23	101	137	773216	1.7	0.5
bloc4m	1109600	1.2	18	89	116	646400	1.5	0.5
findnls	677881	0.7	29	150	158	271487	0.6	0.4
bloc19m	594400	0.6	11	72	96	136000	0.3	0.5
divide	573253	0.6	24	77	88	372065	0.8	0.2
bloc13m	365600	0.4	12	53	67	171200	0.4	0.2
bloc38m	351718	0.4	14	112	139	163989	0.4	0.2
bloc45m	350041	0.4	12	104	127	143678	0.3	0.2
bloc43m	317800	0.3	14	77	102	170000	0.4	0.1
main	266109	0.3	51	175	259	205259	0.5	0.1
bloc46m	250420	0.3	42	95	121	219225	0.5	0.0
vscalel	222036	0.2	29	120	128	117964	0.3	0.1
bloc36m	199400	0.2	15	61	87	214800	0.5	-0.0
bloc12m	179400	0.2	20	100	121	68000	0.2	0.1
bloc11m	148000	0.2	10	25	35	107200	0.2	0.1
bloc16m	143200	0.1	10	46	63	136000	0.3	0.0
Reads	63249	0.1	8	19	27	59246	0.1	0.0
bloc96m	48800	0.1	7	23	32	43200	0.1	0.0
init	697	0.0	67	145	191	687	0.0	0.0
Total	95722762	100.0	1061	4055	5020	44517442	100.0	53.5

Figure 76 : Table des performances après une première optimisation

7.3.2. La deuxième phase d'optimisation

7.3.2.1. L'optimisation du bloc50m

En partant des résultats de la Figure 76, la fonction la plus coûteuse, « *bloc50m* », a tout d'abord été optimisée. Grâce aux résultats fournis au niveau des blocs de base il a été très facile de localiser dans le code C les parties critiques de cette fonction. Ce sont des instructions effectuant un test de dépassement de capacité sur des données de 32 bits. En langage C, ceci se traduit par l'utilisation du type de données flottant (*double*). Or, avec le OakDSPCore ce traitement peut s'effectuer sans utiliser ce type (le test de dépassement de capacité sur des données de 32 bits est inclus dans l'instruction). Malheureusement, il n'est pas possible de le décrire en C standard et le compilateur n'est pas en mesure de générer de telles instructions. Il est nécessaire d'utiliser des macros DSP (différentes des macros du langage C) fournies avec le compilateur du OakDSPCore. On indique ainsi explicitement quelles instructions ce dernier doit utiliser. L'estimation de niveau RTL est capable de prendre en compte ces macros DSP dans le calcul des performances. Notons enfin que la structure du programme a été aussi légèrement modifiée afin de réduire le nombre de variables utilisées. La Figure 77 montre les performances du *bloc50* après optimisation. Le gain pour le code généré est de 17 MIPS et de plus de 9 MIPS pour l'estimation RTL. Comme les macros DSP font parties des modifications algorithmiques, il est normal que l'estimation soit affectée significativement par ces modifications. L'estimation fournie par VESTIM montre cependant qu'il est possible d'exécuter la fonction *bloc50m* en 5 MIPS (au lieu de 16,37 pour le code généré par le compilateur). Comme il semble difficile d'optimiser plus encore le code C, l'estimation de niveau RTL indique alors le gain possible si le code est directement écrit et optimisé en assembleur. Notons enfin que les *bloc34m* et *bloc44m* sont des fonctions quasiment identiques au *bloc50m*, aussi les mêmes optimisations y ont été effectuées.

7.3.2.2. L'optimisation du bloc17m

La fonction *bloc17m* a été optimisée en réduisant tout d'abord la durée de vie des variables (i.e. en augmentant le nombre de variables). Cette fonction a permis ensuite de constater qu'il n'est pas toujours vrai que l'utilisation de pointeurs pour accéder à des données stockées dans un tableau permet de réduire le nombre de cycles. Dans certain cas en effet, l'initialisation des pointeurs et l'augmentation du nombre de variables provoque une augmentation du nombre de cycles. Une optimisation algorithmique a enfin été effectuée en supprimant un test inutile. Toutes ces optimisations permettent de réduire de 7,5 MIPS les performances du code généré par le compilateur mais comme pour le *bloc50m*, l'estimation de niveau RTL montre qu'il est possible d'exécuter encore plus efficacement cette fonction quand elle est écrite à la main.

7.3.2.3. L'optimisation de hwmcore

Dans cette fonction, un des paramètres est une table dont chaque élément est multiplié par lui-même. Comme il n'est pas possible avec le OakDSPCore de lire dans le même cycle deux données dans le même banc de mémoire, cette multiplication prend trois cycles. Pour exécuter cette multiplication en un cycle, une optimisation consiste à dupliquer la table afin de doubler la bande passante avec la mémoire. Cette modification nécessite l'ajout d'un paramètre (une copie de la table), l'utilisation d'un pointeur supplémentaire et n'est intéressante que pour des tableaux de taille raisonnable. La duplication de la table est

effectuée par les fonctions appelant la fonction *hwmcore* (*bloc36m*, *bloc43m* et *bloc49m*). Ces dernières ont donc été modifiées ce qui explique que leur coût augmente légèrement.

7.3.2.4. L'utilisation d'une macro DSP pour la division

La fonction *simpdiv* effectue une division entière. La description en C de ce traitement ne permet pas au compilateur C du OakDSPCore de générer l'instruction « *divs* ». L'optimisation de cette fonction consiste donc à indiquer explicitement via une macro DSP au compilateur l'instruction qu'il doit utiliser. La performance requise du code généré diminue ainsi de 5,3 MIPS alors que l'estimation passe de 2,8 MIPS à 0,6 MIPS. Comme les performances du code LST et de l'estimation RTL sont très proches, cette fonction semble (sauf autre modification algorithmique) optimisée.

Name	LST	%	C	ASM	Words	RTL	%	Delta
bloc50m	**16372457**	**26.3**	**84**	**237**	**294**	**5058328**	**17.8**	**18.2**
bloc17m	**11204258**	**18.0**	**36**	**100**	**128**	**6754596**	**23.8**	**7.1**
hwmcore	**9875768**	**15.9**	**79**	**358**	**402**	**2694083**	**9.5**	**11.6**
bloc14m	**6270236**	**10.1**	**20**	**40**	**51**	**3699225**	**13.0**	**4.1**
bloczirm	**3434697**	**5.5**	**51**	**205**	**265**	**2337977**	**8.2**	**1.8**
bloc9m	2034015	3.3	71	339	439	1030346	3.6	1.6
bloc49m	1337576	2.1	23	115	153	816416	2.9	0.8
bloc44m	1326742	2.1	81	249	304	370770	1.3	1.5
bloc37m	1324735	2.1	81	249	304	384660	1.4	1.5
bloc51m	1276585	2.0	12	101	117	585458	2.1	1.1
vscale	1224761	2.0	39	198	215	774481	2.7	0.7
bloc4m	1109600	1.8	18	89	116	646400	2.3	0.8
simpdiv	682815	1.1	94	7	7	613140	2.2	0.1
findnls	677881	1.1	29	150	158	271487	1.0	0.7
bloc19m	594400	1.0	11	72	96	136000	0.5	0.8
divide	573253	0.9	24	77	88	372065	1.3	0.3
bloc13m	365600	0.6	12	53	67	171200	0.6	0.3
bloc38m	351718	0.6	14	112	139	163989	0.6	0.3
bloc45m	350041	0.6	12	104	127	143678	0.5	0.4
bloc43m	347400	0.6	14	83	110	185200	0.7	0.3
main	266109	0.4	51	175	259	205259	0.7	0.1
bloc46m	250420	0.4	42	95	121	219225	0.8	0.0
bloc36m	249800	0.4	15	67	95	240400	0.8	0.0
vscalel	222036	0.4	29	120	128	117964	0.4	0.2
bloc12m	179400	0.3	20	100	121	68000	0.2	0.2
bloc11m	148000	0.2	10	25	35	107200	0.4	0.1
bloc16m	143200	0.2	10	46	63	136000	0.5	0.0
Reads	63249	0.1	8	19	27	59246	0.2	0.0
bloc96m	48800	0.1	7	23	32	43200	0.2	0.0
init	697	0.0	67	145	191	687	0.0	0.0
Total	62306253	100.0	1064	3753	4652	28406680	100.0	54.4

Figure 77 : Table des résultats pour une seconde optimisation

7.3.3. Le dépliement de boucle

La dernière optimisation effectuée concerne les fonctions *bloc14m*, *bloc17m* et *bloczirm*. Pour ces dernières nous avons déplié des boucles de petite taille afin d'économiser les cycles dédiés aux instructions de boucle. Le gain peut paraître faible mais quand chaque instruction est exécutée 500 000 fois par seconde ce n'est pas négligeable. C'est d'ailleurs

ce qui est fait lorsque le code assembleur est écrit et optimisé à la main. La Figure 78 montre les résultats ainsi obtenus après le dépliement des boucles. On constate que les performances du code généré des fonctions *bloc17*m et *bloc14m* augmentent après ces optimisations. L'analyse du code assembleur généré révèle en effet que le compilateur « déplie » également les adresses (i.e. les pointeurs) ce qui provoque des sauvegardes temporaires de données en mémoire. Par contre, au niveau des estimations RTL ces modifications sont correctement gérées et permettent une estimation très proche du code assembleur optimisé écrit à la main. Notons enfin que pour la fonction *bloczirm*, le dépliement d'une boucle est géré correctement par le compilateur comme par l'estimation de niveau RTL, bien que le nombre d'instructions dépliées soit le même que pour les autres fonctions. La raison provient du nombre restreint de variables utilisées par cette fonction.

Name	LST	%	C	ASM	Words	RTL	%	Delta
bloc17m	**18128927**	**26.1**	**40**	**154**	**187**	**6368996**	**23.8**	**16.9**
bloc50m	**16372457**	**23.5**	**82**	**237**	**294**	**5058328**	**18.9**	**16.2**
hwmcore	**9875768**	**14.2**	**79**	**358**	**402**	**2694083**	**10.1**	**10.3**
bloc14m	**7044968**	**10.1**	**74**	**152**	**168**	**2903641**	**10.9**	**5.9**
bloczirm	**2994697**	**4.3**	**54**	**199**	**258**	**1865977**	**7.0**	**1.6**
bloc9m	2034015	2.9	71	339	439	1030346	3.9	1.4
bloc49m	1337576	1.9	23	115	153	816416	3.1	0.7
bloc44m	1326742	1.9	79	249	304	370770	1.4	1.4
bloc37m	1324735	1.9	79	249	304	384660	1.4	1.3
bloc51m	1276585	1.8	12	101	117	585458	2.2	1.0
vscale	1224761	1.8	39	198	215	774481	2.9	0.7
bloc4m	1109600	1.6	18	89	116	646400	2.4	0.7
simpdiv	682815	1.0	94	7	7	613140	2.3	0.1
findnls	677881	1.0	29	150	158	271487	1.0	0.6
bloc19m	594400	0.9	11	72	96	136000	0.5	0.7
divide	573253	0.8	24	77	88	372065	1.4	0.3
bloc13m	365600	0.5	12	53	67	171200	0.6	0.3
bloc38m	351718	0.5	14	112	139	163989	0.6	0.3
bloc45m	350041	0.5	12	104	127	143678	0.5	0.3
bloc43m	347400	0.5	14	83	110	185200	0.7	0.2
main	266109	0.4	51	175	259	205259	0.8	0.1
bloc46m	250420	0.4	42	95	121	219225	0.8	0.0
bloc36m	249800	0.4	15	67	95	240400	0.9	0.0
vscalel	222036	0.3	29	120	128	117964	0.4	0.1
bloc12m	179400	0.3	20	100	121	68000	0.3	0.2
bloc11m	148000	0.2	10	25	35	107200	0.4	0.1
bloc16m	143200	0.2	10	46	63	136000	0.5	0.0
Reads	63249	0.1	8	19	27	59246	0.2	0.0
bloc96m	48800	0.1	7	23	32	43200	0.2	0.0
init	697	0.0	67	145	191	687	0.0	0.0
Total	69565654	100.0	1119	3913	4821	26753496	100.0	61.5

Figure 78 : Table des performances après dépliement des boucles

7.3.4. Conclusion

Les résultats pour les différentes versions du code C de l'encodeur de G.728 sont regroupés dans la Table 10. La première colonne représente les noms que nous avons donnés aux optimisations successives apportées au code C. La colonne LST décrit les performances du code assembleur généré par le compilateur C alors que la colonne RTL

représente l'estimation d'un code optimisé fournit par VESTIM. Pour chacune des versions on fournit également une mesure de pessimisme par rapport à la performance du code assembleur optimisé écrit à la main (23 MIPS pour l'encodeur de G.728). On rappelle ci-dessous la définition utilisée pour calculer le pessimisme.

$$\text{Pessimisme} = \frac{b_e - b_m}{b_m}$$

b_e : borne estimée

b_m : borne mesurée

A partir du code C d'origine C0, l'obtention de la version C3 a nécessité deux semaines de travail. L'écriture du code assembleur optimisé entièrement à la main a nécessité un mois et demi de travail[60]. En partant de la version C2 et en réécrivant seulement 20% du code « à la main », il est possible d'obtenir un code s'exécutant en 35 MIPS. Ce résultat n'est pas acceptable par rapport au 23 MIPS d'un code optimisé et il est nécessaire d'optimiser encore le code C ou d'écrire plus de fonctions directement en assembleur. La Figure 79 montre graphiquement l'évolution des performances du code généré comme des estimations en fonction des versions du code C.

Version	LST	RTL	Pessimisme LST	Pessimisme RTL
C0	121.1	58.9	426.5	156.0
C1	95.7	44.5	316.1	93.4
C2	62.3	28.4	170.8	23.4
C3	69.5	26.7	202.1	16.1

Table 10 : Optimisation du code C pour G.728

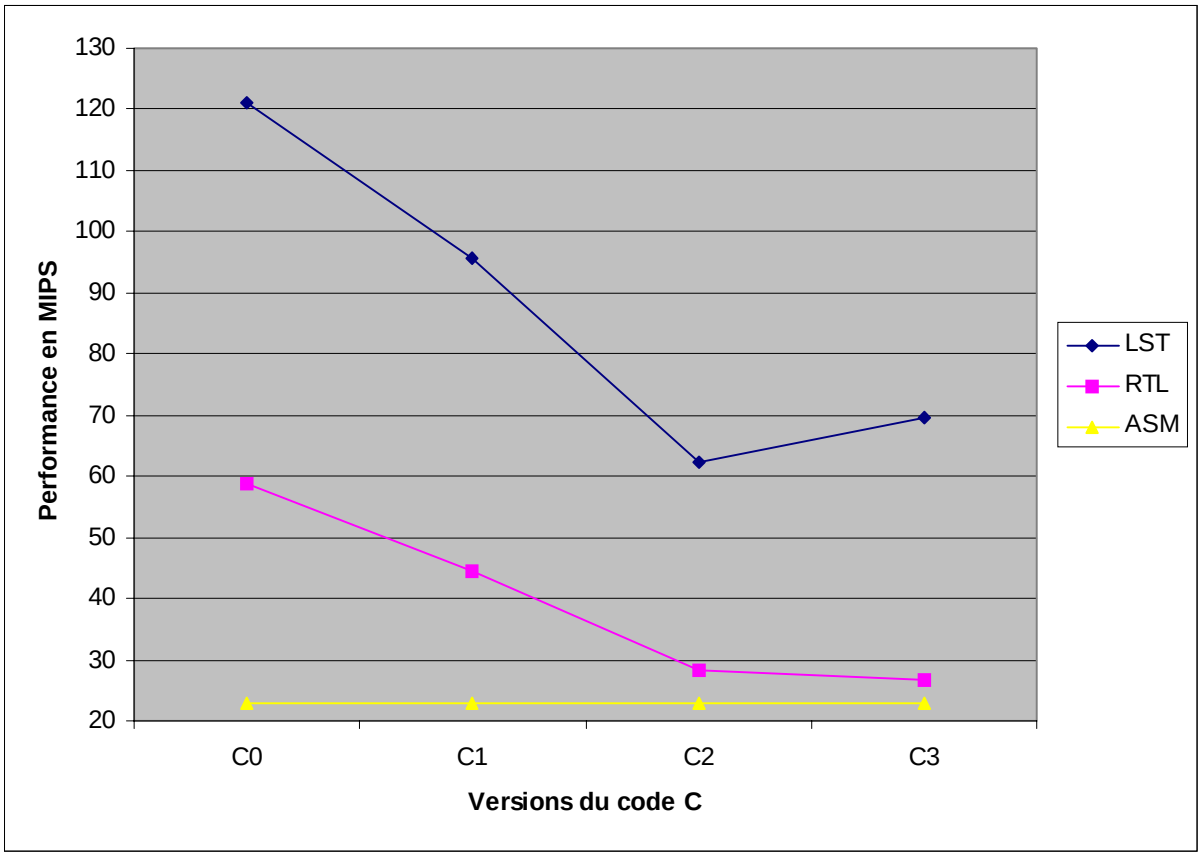


Figure 79 : Courbe de performance pour G.728

[60] Implémentation réalisée par VLSI Technology.

La Table 11 regroupe pour les fonctions les plus coûteuses de l'application (celles qui ont été optimisées) les mesures de performance pour le code généré LST[61] et l'estimation RTL[62], ainsi que le pessimisme associé par rapport à un code assembleur optimisé (colonne ASM). Cette table montre que les estimations de niveau RTL sont très précises puisqu'elles ne dépassent jamais +/-10% de pessimisme. Le pessimisme moyen pour les estimations de niveau RTL pour la version C3 est de 1,18% alors qu'il est de 12,3% pour la version C2.

Table 11 : Performances fonction par fonction

Fonction	LST	RTL	ASM	Pessimisme LST (%)	Pessimisme RTL (%)
bloc17m	11.2	6.4	7.05	58.8	-9.2
bloc50m	16.4	5.1	4.7	248.9	8.5
hwmcore	9.9	2.7	2.57	285.2	5.1
bloc14m	6.3	2.9	3.14	100.6	-7.6
bloczirm	3	1.9	1.74	72.4	9.2

[61] On a pris les meilleures fonctions de C2 et C3.
[62] Pour la version C3.

7.4. Résultats pour l'application G.723.1

Pour l'application G.723.1, un code assembleur optimisé nécessite 23,5 MIPS (courbe *ASM* sur la Figure 80) pour s'exécuter sur le OakDSPCore. Cette information provient d'une implémentation effectuée manuellement par Samsung [Lee97]. La Table 12 montre les résultats fournis par VESTIM et obtenus pour différentes optimisations du code C. L'obtention de la version C9 a nécessité deux mois de travail.

Version	LST	RTL	Optimisations effectuées
C0	86.3	23.6	Code d'origine[63]
C1	48.3	23.7	Réduction de la durée de vie des variables dans les fonctions les plus coûteuses
C2	52.7	23.7	Utilisation de l'unité de saturation automatique sur des données de 16 bits
C3	51.2	23.2	Optimisation algorithmique : accès par pointeurs d'un tableau à 2 dimensions
C4	46.4	21.6	Optimisation algorithmique : suppression de décalages inutiles
C5	43.6	21.4	Réduction de la durée de vie des variables
C6	39	21.9	Localisation des données dans les bancs de mémoire Xram ou Yram
C7	36.7	23.2	Diverses optimisations : localisation des données, réduction de la durée de vie des variables, etc.
C8	32.7	22.2	Réduction de la durée de vie des variables dans une partie critique
C9	32	21.3	Modifications algorithmiques : meilleure gestion de la mémoire des filtres

Table 12 : Optimisations du code C pour G.723.1

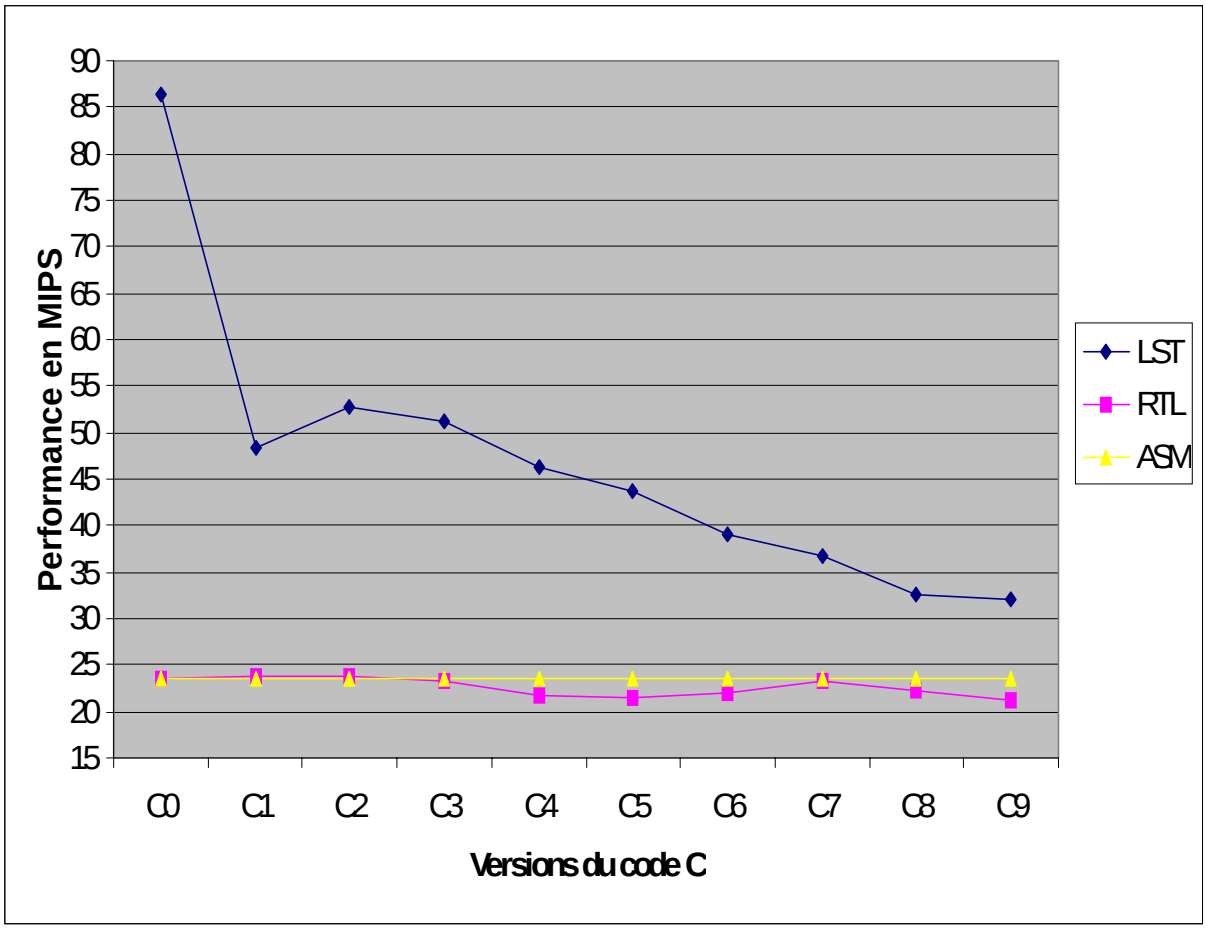


Figure 80 : Courbe des optimisations du Code C pour G.723.1

[63] Tous les opérateurs de base (addition, soustraction, …) définis comme des fonctions dans le code C d'origine (spécification ITU) sont redéfinis par des macros du langage C.

7.4.1. Réduction de la durée de vie des variables

La Figure 80 montre que l'estimation RTL fournie par VESTIM est très précise dès le code C d'origine (version C0). Les versions C1, C5 et C8 obtenues en optimisant la durée de vie de certaines variables permettent d'améliorer de manière significative les performances du code généré par le compilateur. Le gain cumulé pour ces trois versions est de 44,8 MIPS. Pour les estimations de niveau RTL au contraire, on remarque que réduire la durée de vie des variables n'a que très peu d'influence sur les performances (-1,1 MIPS en cumulé). On pouvait s'attendre à ce résultat puisque la représentation intermédiaire sur laquelle se basent les estimations travaille sur des registres virtuels en nombre très important. Il n'y a donc pas de problème lié à la sauvegarde temporaire de données en mémoire (*spilling*). C'est une caractéristique très intéressante pour le programmeur lors de l'optimisation du code C, puisque comme le montre l'application G.723.1 et comme nous l'avons vu plus tôt, les problèmes de *spilling* sont souvent responsables d'une partie importante de l'inefficacité des compilateurs.

7.4.2. Modifications algorithmiques

Les versions C2, C3, C4 et C9 sont des modifications algorithmiques. Pour les versions C3 et C9 par exemple, le code C est réécrit pour optimiser les accès à des tableaux (i.e. pour faire varier les indices linéairement). Ces optimisations permettent un gain de performance de 7 MIPS du code généré et de 3 MIPS de l'estimation de niveau RTL. Cette fois-ci, les estimations de niveau RTL varient de manière plus importante, ce qui est normal puisqu'il n'est pas possible d'estimer le gain dû à une modification algorithmique du code C.

7.4.3. Localisation des données dans les bancs de mémoires

La version C6 est une réécriture du code C utilisant les extensions permettant de localiser les données dans les bancs de mémoires XRAM ou YRAM. Grâce à ces indications, le compilateur génère plus efficacement des instructions MAC et les performances requises du code assembleur généré diminuent de 4,6 MIPS.

Les estimations de niveau RTL sont peu affectées par ce type de modifications. Le nombre de MIPS augmente même très légèrement. Cette légère augmentation n'est cependant pas surprenante puisqu'en l'absence d'indication sur la localisation des données, l'estimateur effectue lors de l'ordonnancement une allocation virtuelle des données de manière optimisée, c'est-à-dire dans les bancs de mémoires XRAM ou YRAM. Lors de cette allocation virtuelle, la cohérence entre blocs de base de la localisation des données n'est pas vérifiée. Par contre si le programmeur ajoute des informations sur la localisation des données, l'ordonnancement du graphe de flots de données de la représentation intermédiaire s'effectue en respectant les indications apportées par le programmeur. C'est ce qui se passe également pour la version C7 qui regroupe plusieurs optimisations (non algorithmiques) dont la localisation des données dans les différents bancs de mémoire.

7.4.4. Conclusion

L'expérimentation menée avec l'application G.723.1 montre que les estimations RTL sont très précises et surtout très stables par rapport à différentes versions du code C. L'estimation d'un code optimisé est en effet toujours très proche de la performance de 23,5 MIPS obtenue lors d'un codage manuel. Certaines optimisations, comme la réduction de la durée de vie des variables, ont une influence très importante sur le code généré par le compilateur alors qu'elles sont quasiment sans effet sur les estimations RTL. Seules les optimisations algorithmiques font varier plus sensiblement les estimations RTL. On peut

affirmer, pour cette application, que l'estimation de niveau RTL représente une mesure précise pour le programmeur lors de l'estimation du code C. En effet, en comparant les performances du code généré (LST) avec les estimations RTL fournies par l'outil, il est possible pour le programmeur de déterminer si le compilateur a généré un code assembleur efficace et ainsi de mieux focaliser le travail d'optimisation du code C.

7.5. Résultats pour l'application G.729

Version	LST	RTL	Pessimisme LST	Pessimisme RTL
C1[64]	32.4	19.4	149.2	49.2
C2	29.1	19.3	123.8	48.4
C3	27.4	19	110.7	46.1
C4	22.1	16.9	70	30
C5	20.9	15.7	60.7	20.8

Table 13 : Optimisations du code C pour G.729

La Table 13 montre les résultats obtenus pour l'application G.729 pour différentes optimisations du code C. La version C1 représente le code C d'origine issu directement de l'ITU[64]. L'obtention de la version C5 a demandé un mois et demi de travail. La Figure 81 représente graphiquement ces résultats montrant une fois encore la stabilité des estimations RTL par rapport aux performances du code généré LST.

7.5.1. Les optimisations apportées au code C

La version C2 est obtenue en réduisant la durée de vie des variables dans les parties les plus critiques (en nombre de cycles) de l'application. Comme nous l'avons vu pour G.723.1, ces modifications n'ont pas d'influence sur les estimations de niveau RTL alors que le gain est de 11% pour le code généré LST. La version C3 représente le code C dont les accès mémoires ont été optimisés principalement en utilisant des pointeurs pour accéder à des données stockées dans des tableaux. L'estimation RTL est très peu sensible à ces modifications alors que les performances du code LST diminuent de 6%. La version suivante, C4, regroupe un ensemble d'optimisations comme la réduction de la durée de vie des variables mais aussi l'utilisation de *macros* du langage C à la place de fonctions, évitant ainsi le coût associé à l'appel puis au retour de la fonction. C'est l'utilisation de macros qui est responsable de la baisse observée au niveau RTL (-2,1 MIPS). La diminution est encore plus sensible pour le code généré (-5,3 MIPS). Finalement la dernière version C5, regroupe différentes optimisations algorithmiques comme la réécriture de boucle « w*hile* » en « *for* » ou la copie de tableaux de petite taille afin d'optimiser les accès mémoires.

[64] Tous les opérateurs de base (addition, soustraction, …) définis comme des fonctions dans le code C d'origine (spécification ITU) sont redéfinis par des macros du langage C.

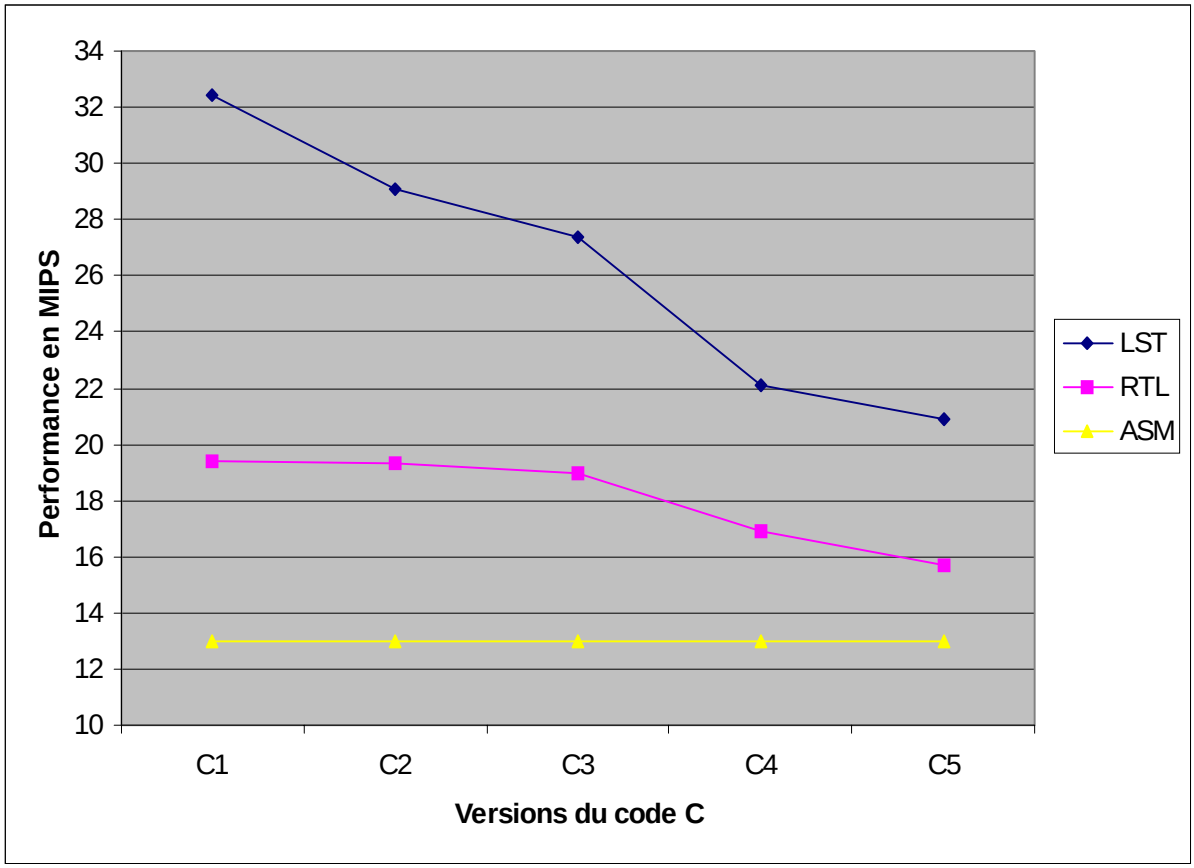


Figure 81 : Courbe des performances pour G.729

7.5.2. Conclusion

La Figure 81 montre que les estimations obtenues à partir du niveau RTL sont certes plus stables que les performances du code généré, mais restent encore assez éloignées du temps d'exécution pour un code assembleur optimisé de 13 MIPS (ASM)[65]. Il est très probable que cette différence soit due principalement à des optimisations algorithmiques.

[65] On se base sur une implémentation sur le TMS320C54.

7.6. Résultats pour l'application AC-3 (décodeur)

La norme AC-3 est une application intéressante puisqu'elle traite non plus un signal de parole mais un signal audio. Ces signaux requièrent plus de précision et donc plus de bits que des signaux de parole, ce qui entraîne souvent l'utilisation de type de données « *long* », i.e. codé sur 32 bits, difficilement exploitable par un DSP dont la largeur naturelle des données est de 16 bits. En effet, des données codées sur 32 bits entraînent l'appel de fonctions très coûteuses définies dans la librairie standard du compilateur pour effectuer des calculs en double précision. La Table 14 montre les résultats obtenus pour différentes optimisations du code C.

Version	LST	RTL	Pessimisme LST	Pessimisme RTL
C0	166.8	55.5	368.5	55.9
C1	145.3	52.5	308.1	47.5
C2	140.2	50.1	293.8	40.7
C3	138.9	50.4	290.2	41.6
C4	137	50.3	284.8	41.3
C5	135.1	50.2	279.5	41.0
C6	68	30	91.0	-15.7

Table 14 : Optimisations du code C pour le décodeur AC-3

La version C0 représente le code C d'origine issu des laboratoires Dolby. En étudiant ce code C, nous nous sommes rendus compte qu'il était déjà en grande partie optimisé pour une implémentation sur un processeur DSP (i.e. présence de fonction de type « *DSP_Limit* »). Malgré cela, le pessimisme est de 46% pour l'estimation de niveau RTL. L'obtention de la version C6 a nécessité un mois et demi de travail.

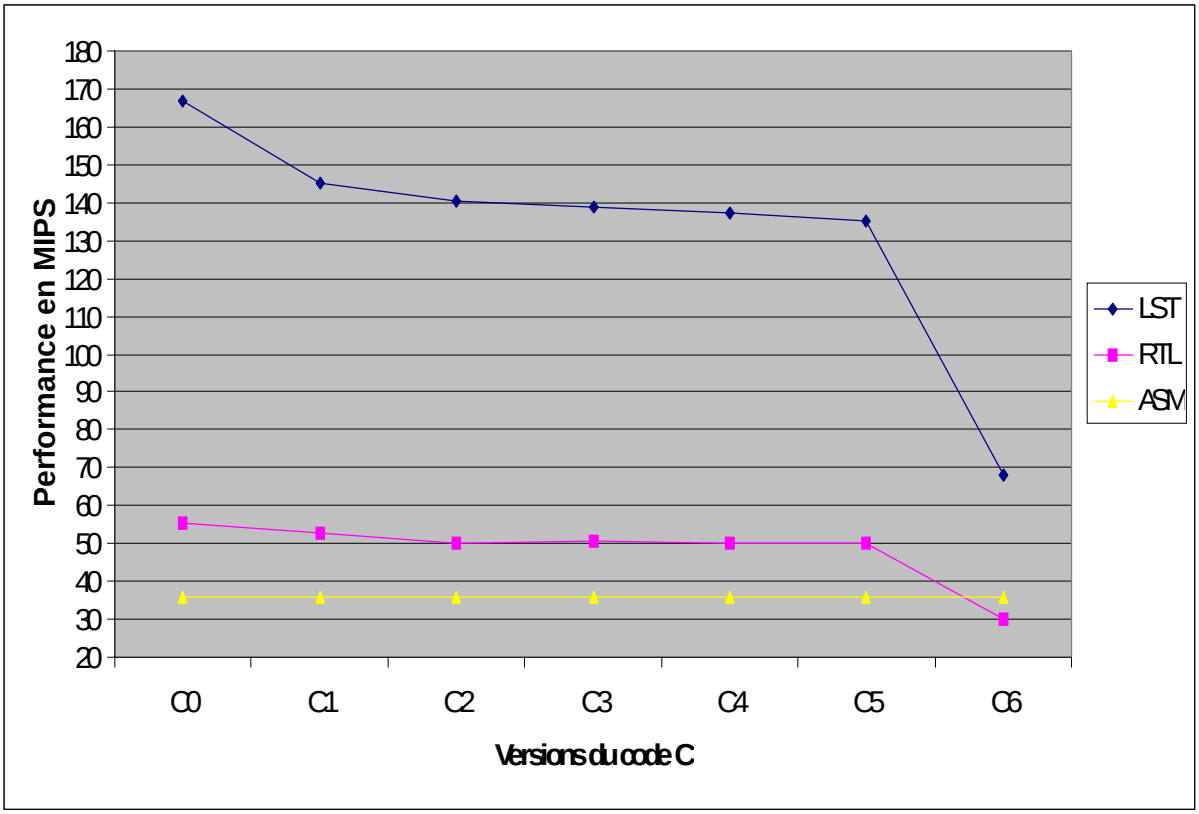


Figure 82 : Courbe des performances pour le décodeur AC-3

7.6.1. Les optimisations classiques

La première optimisation effectuée (C1) consiste à remplacer par des macros du langage C, des fonctions appelées très fréquemment mais dont les traitements sont courts. On élimine ainsi de nombreux appels et retours de petites fonctions. Comme c'est une optimisation algorithmique, l'estimation de niveau RTL est affectée par ces modifications (-2,4 MIPS) même si le gain pour le code généré est deux fois plus important (-5,1 MIPS). Les versions C3 et C5 sont des optimisations de nature similaire puisqu'elles concernent la réécriture de certaines fonctions en macros.

Les versions C2 et C4 représentent des optimisations agissant respectivement sur la réduction de la durée de vie des variables ainsi que la diminution du nombre de variables utilisées. Ces optimisations n'ont pas d'influence sur les estimations de niveau RTL (+0,2 MIPS) alors qu'elles permettent un gain de 3,2 MIPS du code généré.

Malgré toutes ces optimisations, les performances du code assembleur généré par le compilateur sont inacceptables par rapport à une implémentation optimisée[66] (36 MIPS). Notons que le pessimisme est également de 41% pour l'estimation de niveau RTL. L'amélioration des performances passe par des optimisations dans le choix des types de données. Notons que ceci est typique des applications audio.

7.6.2. Le passage en simple précision

La version C6 représente les performances obtenues après l'optimisation du choix des types de données. La méthode utilisée consiste à effectuer au préalable une étude de la dynamique des nombres lors de l'exécution du code C avec une séquence de test représentative. Notons que nous avons également utilisé les résultats présentés dans [Dav92]. Cette étude permet de déterminer tout au long des traitements effectués dans le décodeur, le nombre de bits qu'il est nécessaire de conserver pour les différentes variables. Si la donnée peut être représentée sur 16 bits, il est alors inutile d'utiliser le type « long » et on évite ainsi des calculs en double précision. Il est bien sûr impossible d'estimer a priori le gain relatif de ce type d'optimisation algorithmique. C'est pour cette raison que les estimations de niveau RTL sont très affectées par cette modification du code C (-20,2 MIPS) même si la différence est bien plus importante pour le code généré (-67,1 MIPS).

7.6.3. Un certain optimisme

L'optimisme observé pour l'estimation RTL (pessimisme négatif égal à –15,7%) est vraisemblablement dû à l'hypothèse que nous avons faite lors de la transformation du niveau RTL, selon laquelle il n'y a jamais de sauvegarde temporaire inutile de données en mémoire (*spilling*). En effet, pour le décodeur AC-3, même après l'optimisation des types de données, il existe encore de nombreux calculs en double précision, i.e. sur des données codées sur 32 bits. Comme le OakDSPCore ne possède que deux accumulateurs, ce type de données provoque inévitablement, même lors d'un codage manuel, des sauvegardes temporaires en mémoire lors de calculs intensifs. L'hypothèse que nous avons faite est donc trop forte pour une application comme AC-3. Notons enfin que les performances du code généré LST peuvent être diminuées d'environs 30 MIPS si les calculs du papillon de la FFT inverse (256 ou 512 points) sont codés manuellement. Ceci permettrait alors d'avoir une performance proche de celle obtenue lors d'un codage manuel pour seulement une vingtaine de lignes de code C réécrites en assembleur.

[66] Implémentation réalisée par Ensigma.

7.7. Résultats pour l'application V22bis

Le code C de l'application V22bis regroupe environ 280 fonctions. En considérant uniquement la partie modulation/démodulation de cette norme, ce qui représente 53 fonctions pour environ 10 000 lignes de code C, trois fonctions représentent 80% du temps d'exécution total pour seulement 5,6% de la taille totale du code. La Table 15 montre les résultats obtenus pour différentes optimisations du code C. L'obtention de la version C2 a demandé une semaine de travail

Version	LST	RTL	Pessimisme LST	Pessimisme RTL
C0	23.1	7.6	3,6	0,52
C1	15.6	7.8	2,1	0,56
C2	9.9	5.2	0,98	0,04

Table 15 : Optimisations du code C pour V22bis

La version C0 représente les performances obtenues pour le code C d'origine. Comme une implémentation optimisée est évaluée a environ 5 MIPS, le pessimisme est de 52% pour l'estimation de niveau RTL.

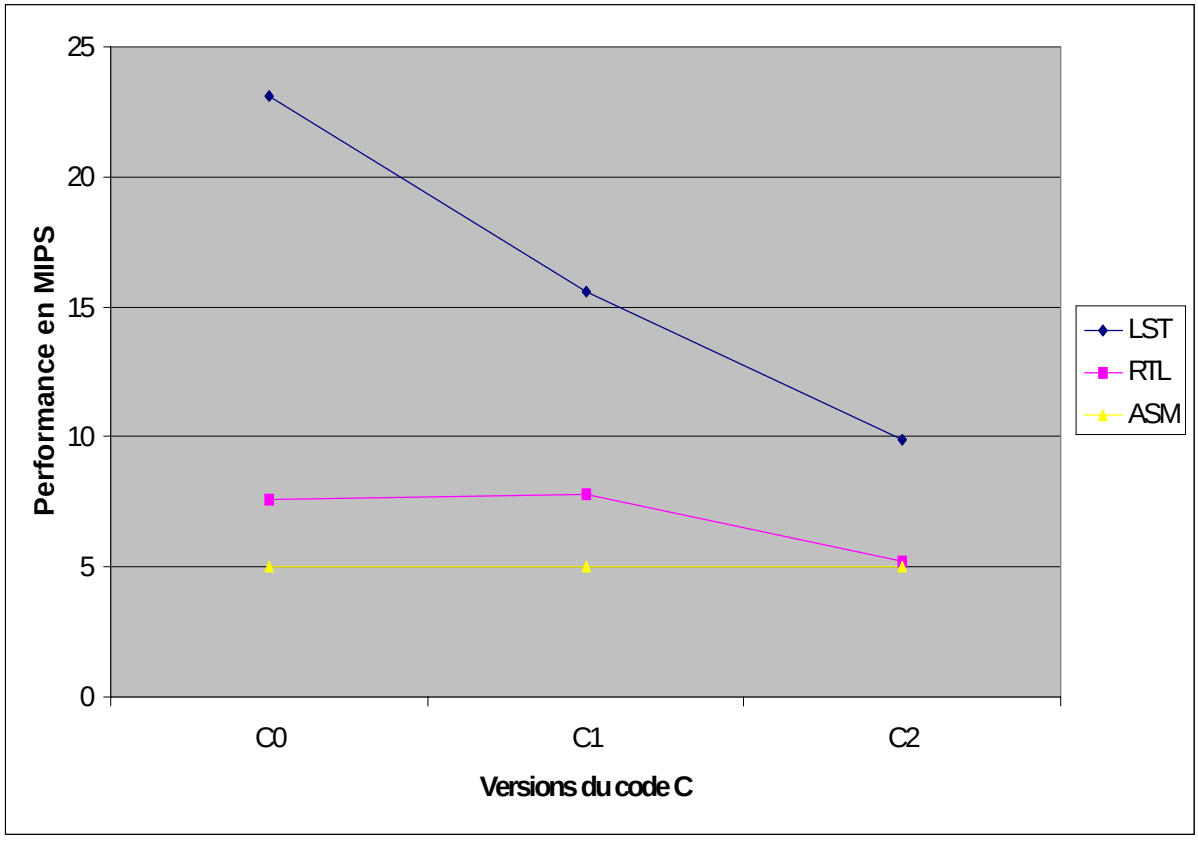


Figure 83 : Courbe des performances pour V22bis

7.7.1. Les optimisations classiques

Les optimisations classiques effectuées sur le code C (version C1) sont :

- l'utilisation de pointeurs pour l'accès aux données d'un tableau,
- la localisation explicite des données dans les bancs de mémoires XRAM ou YRAM,
- et l'indication du type accumulateur (*acc_t*).

Après ces modifications, on constate sans surprise que ces dernières n'ont quasiment pas d'influence sur l'estimation basée sur la représentation RTL (+0,2 MIPS). Par contre, ces optimisations permettent un gain significatif des performances du code généré (-7,5 MIPS).

7.7.2. Les calculs en double précision

Comme pour l'application AC-3, dans la norme V22bis des calculs sont effectués sur des données codées sur 32 bits. Ces opérations en double précision sont d'autant plus coûteuses qu'elles concernent les fonctions les plus fréquemment exécutées de l'application. Après une étude dynamique menée de manière identique à celle effectuée pour le décodeur AC-3, il s'avère que toutes les données peuvent être codées sur 16 bits sans aucune perte de précision. La version C2 représente les performances ainsi obtenues pour des calculs en simple précision. Comme c'est une modification du code C de type algorithmique, l'estimation de niveau RTL diminue de manière significative (-2,6 MIPS) et devient très proche de la performance réelle (pessimisme de seulement 4%). Le gain de performance du code LST est encore plus important (-5,7 MIPS) mais le pessimisme reste encore très élevé. Le code C nécessite donc encore des optimisations ou une partie de l'application doit être écrite manuellement.

7.8. Expérimentations avec le PalmDSPCore

Nous avons commencé à expérimenter la méthode d'estimation sur le PalmDSPCore afin de valider, sur une architecture plus complexe que le OakDSPCore, les techniques mises en œuvre pour la gestion des ressources et du parallélisme. Les difficultés rencontrées concernent tout d'abord l'exploitation du parallélisme. Comme le montrent les exemples qui suivent, il est généralement nécessaire de modifier la description de l'algorithme afin d'exploiter au maximum le parallélisme du processeur. De même, l'annotation du graphe de flots de données par niveau de priorité a été modifié pour traiter deux composantes connexes en parallèle (cf. paragraphe 6.1.2.4). De nouvelles règles de réécriture de la représentation intermédiaire ont été également nécessaires pour tirer profit des instructions triadiques en particulier (e.g. « add3 »). Bien que la description du PalmDSPCore soit moins aisée que dans le cas du OakDSPCore, le modèle de description utilisé permet de prendre en compte les caractéristiques architecturales essentielles de ce processeur.

7.8.1. Un filtre à réponse impulsionnelle finie (FIR)

L'architecture dual-MAC du PalmDSPCore permet d'exécuter un FIR d'ordre N en N/2 cycles. La partie gauche de la Figure 84 montre le code C de l'algorithme d'un filtre à réponse impulsionnelle finie d'ordre N. Si aucune modification n'est effectuée, la performance estimée par notre méthode sera de N cycles. Pour retrouver une performance de N/2 cycles, il est nécessaire de déplier une fois la boucle (le nombre d'itérations passe de N à N/2) afin de faire apparaître deux opérations MAC (partie droite de la Figure 84).

```
res = 0;
for (i=0;i<N;i++)
{
    res += (long)*pa++ * (long)*pb++;
}
```

```
res1 = 0;
res2 = 0;
for (i=0;i<N/2;i++)
{
    res1 += (long)*pa++ * (long)*pb++;    /* 1er MAC */
    res2 += (long)*pa++ * (long)*pb++;    /* 2ème MAC */
}
res = res1 + res2;
```

Figure 84 : Dépliement de boucle pour un algorithme FIR

La méthode d'estimation fournit alors une performance de un cycle pour le calcul de *res1* et *res2* à l'intérieur de la boucle, soit N/2 cycles pour un filtre d'ordre N. Notons que l'estimateur effectue seul le choix des bancs de mémoires pairs ou impairs.

7.8.2. Le papillon de FFT

L'algorithme de calcul d'un papillon de FFT regroupe un nombre important d'opérations (accès mémoires et calculs) et de nombreuses dépendances de données. Afin d'exploiter les instructions triadiques du PalmDSPCore (*add3*, *sub3* et *addsub*), la description de l'algorithme a été légèrement modifiée par rapport à celle utilisée pour le OakDSPCore. Le graphe de flots de données correspondant est décrit par les Figure 85 et Figure 86 qui représentent les deux composantes connexes de ce bloc de base. On remarque que les niveaux de priorité favorisent un ordonnancement parallèle de ces deux composantes.

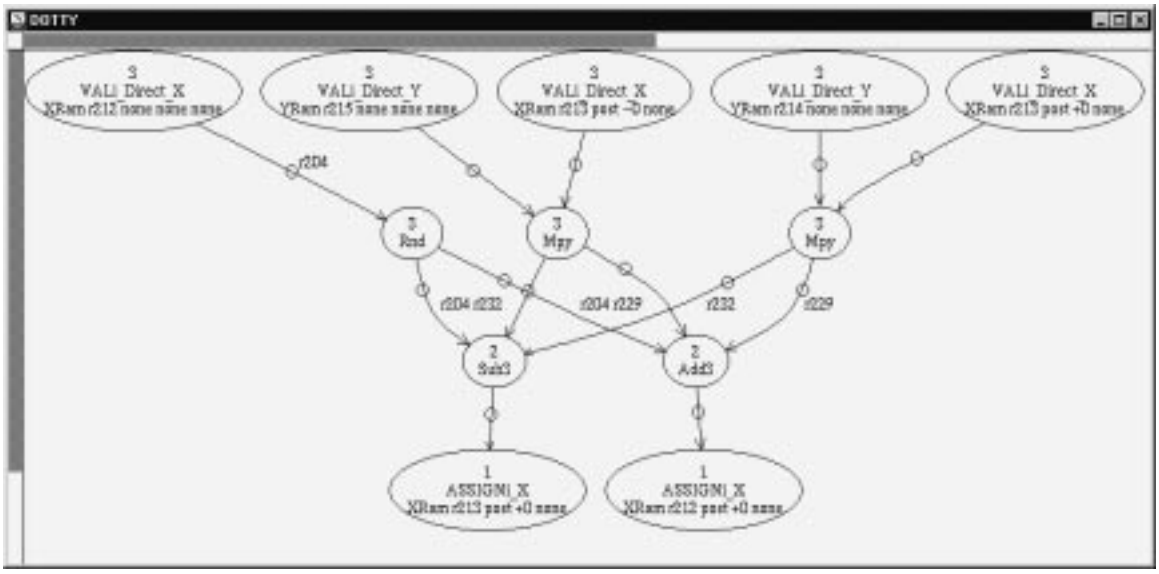


Figure 85 : Première composante connexe du DAG pour le papillon de FFT

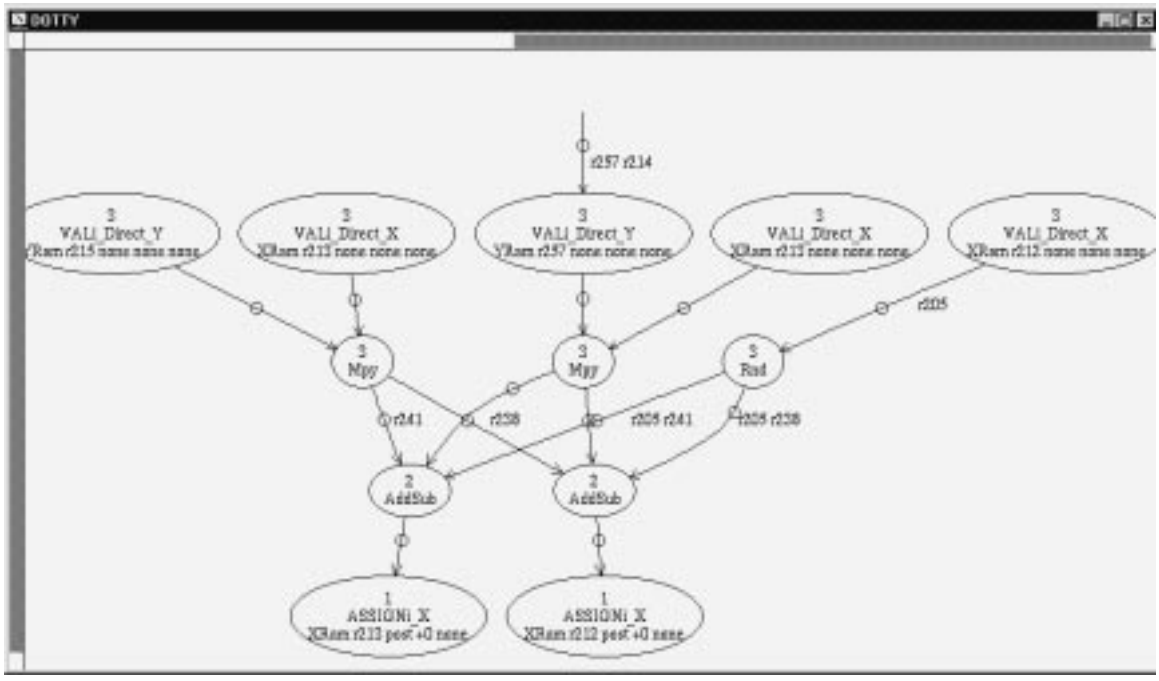


Figure 86 : Seconde composante connexe du DAG pour le papillon de FFT

Un code assembleur optimisé permet d'effectuer le calcul d'un papillon de FFT en deux cycles sur le PalmDSPCore. Ce dernier possède en effet des instructions et des registres dédiés pour cet algorithme (e.g. instruction « *bfly xy* »). Notre méthode fournit quant à elle une estimation du temps d'exécution d'un code optimisé en cinq cycles. La Figure 87 montre la table d'allocation correspondante obtenue après le processus d'estimation. La première ligne décrit les ressources du processeur[67] alors que les lignes suivantes correspondent aux cycles du processeur. Un « . » représente une ressource non utilisée alors qu'une « x » dénote une ressource qui a été désallouée par une opération ordonnancée préalablement (contrainte sur les chemins de données ou liée à l'encodage du jeu d'intsructions).

Cyc	MemX	MemX	MemY	MemY	MUL0	MUL1	ALU0	ALU1	X0	X1	Y0	Y1	P0	P1	ACC0	ACC1	RI0	RI1	RJ0	RJ1
1	ViDX	ViDX	ViDY	ViDY	Mpy	Mpy	....	....	xxxx	xxxx	xxxx	xxxx	Mpy	Mpy	....	....	xxxx	xxxx	xxxx	xxxx
2	ViDX	ViDX	ViDY	ViDY	Mpy	....	Sub3	....	xxxx	....	xxxx	....	Mpy	....	Rnd	Sub3	xxxx	xxxx	xxxx	xxxx
3	ViDX	ViDX	....	....	Mpy	....	AdSb	....	xxxx	....	xxxx	....	Mpy	....	Rnd	AdSb	xxxx	xxxx	....	....
4	AiX	AiX	....	....	....	....	AdSb	Add3	....	....	....	....	....	....	AdSb	Add3	....	....	....	....
5	AiX	AiX	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....	....

Figure 87 : Table d'allocation pour le papillon de FFT

L'analyse de cette table et des opérations exécutées en parallèle par les instructions spécifiques FFT du PalmDSPCore a permis de déterminer les raisons de la différence de trois cycles observée. La principale raison provient de la réduction du nombre d'accès à la mémoire et l'optimisation de la mise à jour des pointeurs par des mécanismes spécialisés. Le PalmDSPCore possède en effet des registres dédiés qui permettent :

- d'éviter de lire plusieurs fois les mêmes données, ici B_r et B_i (deux lectures en moins),
- de ne plus lire à l'intérieur de la boucle les coefficients W_r et W_i (quatre lectures en moins).

Le nombre d'entrées-sorties avec la mémoire passe ainsi de 14 (pour le OakDSPCore) à huit (quatre lectures et quatre écritures). Or, les registres dédiés du PalmDSPCore ne sont pas modélisés dans la description du processeur puisque ces derniers ne sont utilisés que par les instructions dédiées pour la FFT. La Figure 87 montre clairement qu'il n'est pas possible d'exécuter l'algorithme représenté par les Figure 85 et Figure 86 en moins de 5 cycles. Le nombre d'accès à la mémoire, la localisation des données dans les bancs de mémoire et les dépendances de données limitent en effet le parallélisme. En conclusion, nous n'avons pas réécrit un code C valide de la FFT pour lequel il n'existe que huit accès à la mémoire (4 lectures et 4 écritures). Si tel était le cas, l'annotation du graphe de flots de données et la modélisation des instructions (*Mpy*, Add3, Sub3, *AddSub*, …) que nous avons effectué nous permettrait de retrouver les deux cycles d'un code assembleur optimisé sur le PalmDSPCore.

[67] Seules les ressources utilisées par cet algorithme sont représentées.

7.9. Conclusion

Pour le OakDSPCore les estimations obtenues par VESTIM à partir d'une description de niveau RTL sont très précises puisque le pessimisme pour un code C optimisé est compris entre -21,7% et 20,7%, quelle que soit l'application étudiée. Les différentes figures présentées dans ce chapitre montrent de plus que :

- les estimations convergent très rapidement même à partir d'un code C d'origine,
- les estimations sont très stables par rapport aux performances du code généré.

On peut dire à la vue de l'ensemble des résultats que les optimisations suivantes n'ont quasiment aucune influence sur les estimations:

- la réduction de la durée de vie des variables,
- la limitation du nombre de variables utilisées,
- la localisation des données dans les différents bancs de mémoire,
- et l'utilisation de pointeurs pour l'accès aux valeurs d'un tableau.

Par contre, les estimations dépendent du code C dans le cas d'optimisations algorithmiques comme la copie de tableau, le dépliement de boucle, le remplacement de fonctions par des macros ou encore le choix du type de données. Il est enfin également impossible de retrouver à partir d'une description en C les instructions très spécifiques du processeur. Un exemple classique pour le OakDSPCore est l'instruction « *exp* » pour la détection d'exposant ou « *divs* » pour la division entière. Dans ces cas, un moyen pour le compilateur comme pour l'estimateur de retrouver ces instructions passe par l'utilisation de macros DSP fournies avec le compilateur du processeur cible.

Grâce à ces expérimentations, nous avons confirmation que pour les applications de traitement du signal le temps d'exécution total est localisé dans une faible partie de la taille totale du code, i.e. dans quelques fonctions. Ce sont sur ces parties critiques que nous avons effectué le travail d'optimisation. L'utilisation de VESTIM permet d'autre part d'obtenir très rapidement les performances du code généré et des estimations d'un code optimisé. Pour l'encodeur de l'application G.728 par exemple, sur un processeur Intel Celeron cadencé à 333Mhz, VESTIM fournit les résultats en 3 minutes environ. Plus de la moitié du temps est consacrée à la compilation des sources avec OCC. L'estimation d'un code assembleur optimisé ne prend que 25 secondes.

Si seul un fichier est modifié, l'utilisation de *makefile* permet d'avoir de nouvelles estimations en seulement 70 secondes puisque seul le fichier modifié est recompilé (avec « gcc » et « occ »). C'est un facteur important car l'optimisation du code C nécessite un nombre important d'itérations. En utilisant le simulateur de niveau instruction du OakDSPCore, l'exécution du code généré avec la même séquence de test nécessite environ 25 minutes, soit huit fois plus. Il est intéressant de noter aussi que le temps nécessaire à l'obtention des estimations ne dépend quasiment pas de la longueur de la séquence de test, puisque cette dernière est utilisée uniquement lors de l'exécution sur le processeur hôte. Sur cet exemple, toute la partie *front-end* de VESTIM (i.e. compilation des sources avec « *gcc* », exécution avec une séquence de test d'une seconde et annotation du code C par les informations dynamiques) requiert seulement 35 secondes. Ceci permet donc de connaître les performances pour de longues séquences de test sans presque influencer le temps d'obtention des résultats.

Pour le PalmDSPCore, nos premières expérimentations montrent que la description de l'algorithme est importante pour permettre d'exploiter au maximum le parallélisme offert par ce processeur. L'algorithme d'ordonnancement, l'annotation des graphes de flots de données par niveau de priorité ou la description du processeur donnent les résultats attendus pour les deux exemples traités. De nouvelles expérimentations sont néanmoins nécessaires pour valider le modèle d'estimation.

Conclusion et perspectives

L'augmentation de la complexité des applications et des architectures récentes DSP ainsi que la pression sans cesse plus forte du « *time-to-market* » donnent aux compilateurs une place de plus en plus prépondérante dans le choix d'un DSP. Les problèmes de génération de code efficace pour ces processeurs suscitent d'ailleurs un intérêt croissant de la part des industriels comme de la communauté scientifique. Malheureusement, l'inefficacité des compilateurs C actuels pour DSP oblige tout ou partie de l'application à être codée en assembleur, car cela reste le plus souvent la seule solution envisageable pour respecter les fortes contraintes de coût, de performance et de consommation des systèmes embarqués. Cette situation devient de plus en plus inacceptable et la réduction du temps de développement passe par une approche de plus haut niveau.

L'objectif de cette thèse était de concevoir et réaliser une méthode d'estimation de performance d'un code assembleur optimisé adapté à une classe de processeurs permettant d'utiliser plus systématiquement le compilateur du DSP cible. Une des principales contraintes de ce travail était de fournir des mesures à partir d'une description de haut niveau de l'application. Cette approche se différencie des méthodes d'estimation existantes puisque ces dernières se basent généralement sur le code généré par le compilateur, ce qui représente plus une mesure de l'inefficacité du compilateur qu'une évaluation des performances réelles que l'on peut attendre du DSP cible. En conséquence, l'essentiel du travail a consisté à déterminer tout d'abord les principales causes d'inefficacité des compilateurs actuels, puis à définir un moyen d'estimer les performances d'un code assembleur optimisé à partir d'un programme écrit en langage C.

Bilan du travail réalisé

La méthode d'estimation s'appuie sur une représentation intermédiaire du programme, entre le code C et l'assembleur, que l'on modifie de manière à obtenir un modèle d'exécution adapté à un schéma de calcul orienté DSP. A partir de cette représentation, on utilise une heuristique d'ordonnancement par liste dont la phase d'annotation du graphe de flots de données et la gestion des composantes connexes ont été adaptées afin d'exploiter efficacement le parallélisme interne du processeur. Le modèle de description du processeur est basé sur une représentation comportementale simplifiée puisque certains registres ou instructions ne sont pas décrits quand ces derniers n'ont pas ou peu d'influence sur les performances globales. Le modèle utilisé permet de décrire pour chaque opération de base des contraintes sur l'utilisation des ressources (registres ou opératoires) afin d'ordonnancer cette tâche. La modélisation du parallélisme utilise un approche par restriction adaptée à un large éventail d'architectures DSP, en particulier celles comportant beaucoup de parallélisme.

Cette méthode d'estimation a donné lieu à la conception et au développement d'un outil appelé VESTIM. Ce dernier fournit, sans aucune simulation de niveau assembleur, les performances du code généré par le compilateur ainsi qu'une estimation d'un code optimisé. Les résultats sont fournis à deux niveaux de granularité, fonctions et blocs de base. Ces mesures permettent de localiser précisément les parties critiques de l'application où doivent s'effectuer en priorité les optimisation du code C. VESTIM fournit également d'autres informations comme le pourcentage de temps de calcul par fonctions, la taille de la mémoire programme ou le nombre de lignes de code C. Enfin, des informations sur l'utilisation des ressources ou le nombre et le type d'accès mémoires sont également disponibles. Cet outil nous a permis de valider notre approche pour le OakDSPCore sur un ensemble

d'applications complètes de compression d'un signal de parole ou audio, ainsi que le code modem V22bis. Les résultats obtenus montrent que les estimations sont très proches des performances atteintes lorsque le programme est effectivement codé manuellement et représente ainsi une information utile pour le programmeur lors de l'utilisation du compilateur cible. En effet, en comparant les performances du code généré avec l'estimation d'un code optimisé, l'utilisateur est en mesure de déterminer la qualité du code généré par le compilateur. Les expérimentations ont montré cependant que les estimations peuvent dans certains cas dépendre du code C. Il est par exemple impossible avec notre approche d'estimer le gain de performances dû à une modification algorithmique. Par contre, les mesures ont clairement établi que certaines optimisations comme la réduction de la durée de vie des variables par exemple, n'ont pas d'influence sur les estimations contrairement aux performances du code généré.

VESTIM est actuellement utilisé pour caractériser les éléments constituant la bibliothèque de fonctions spécifiques (e. g. FFT) ou d'applications complètes (e. g. AC-3) de l'outil de *co-design*, CODEF [Bia99b], développé dans l'équipe. L'utilisation d'estimation de performances d'un code assembleur optimisé permet d'optimiser les solutions de partitionnement logiciel/matériel par rapport à l'utilisation de mesures issues directement de la compilation.

Amélioration de l'outil d'estimation

Actuellement, l'outil d'estimation VESTIM inclut le OakDSPCore et le PalmDSPCore. Pour le OakDSPCore, l'outil d'estimation est opérationnel et est d'ores et déjà utilisé au sein de VLSI Technology pour le développement d'applications. Pour l'application G.728, nous avons montré que cette approche permet de réduire de moitié le temps de développement. En ce qui concerne le PalmDSPCore, il reste encore de nombreuses expérimentations à effectuer afin d'affiner le modèle d'estimation et en particulier, la description du processeur cible. Un des problèmes rencontrés avec ce processeur est que contrairement au OakDSPCore, nous ne disposons pas de code assembleur optimisé de référence qui permettrait d'étalonner les estimations fournies par l'outil[68]. Il existe néanmoins quelques routines optimisées manuellement pour un ensemble de fonctions critiques d'une application de compression de la parole qui peuvent servir de base de validation des résultats. Il est important de noter que la méthode d'estimation utilisée permet d'inclure sans grande difficulté ce type de processeurs possédant beaucoup de parallélisme. Les modifications qui ont du être apportées à l'outil par rapport au OakDSPCore ne concernent en effet que le choix des règles de réécriture de la représentation intermédiaire et la gestion des composantes connexes à l'intérieur d'un bloc de base. Nous sommes en train d'élaborer un moyen d'inclure ces informations dans la description du processeur cible afin de rendre la méthode d'estimation uniquement dépendante de ce paramètre. Pour le choix des règles de réécriture cela ne pose aucun problème puisqu'il suffit de donner la liste de celles utiles pour le processeur cible. La gestion des composantes connexes est un point plus délicat qui nécessite une réflexion plus importante.

Le OakDSPCore et le PalmDSPCore sont des processeurs dont l'exécution des instructions a pas ou peu de contraintes dues au pipeline. En conséquence, nous n'avons pas modélisé dans la description du processeur l'utilisation au niveau cycle des ressources de ce

[68] Le PalmDSPCore ne sera disponible qu'à la fin de l'année 1999.

dernier par chaque instruction. Il serait néanmoins aisé d'étendre le modèle de description pour un processeur dont le pipeline aurait une influence non négligeable sur les performances. La méthode utilisée dans ARMOR [Mes99b] par exemple s'applique très facilement à notre modèle. De manière équivalente, nous n'avons pas cherché à modéliser l'influence de mémoires caches sur les performances puisque les DSP concernés par cette étude n'en possédaient pas. On pourrait cependant étendre notre méthode en utilisant par exemple l'approche décrite dans [Ott97]. Dans ce cas, le CFG du programme est annoté avec des variables représentant l'historique de l'exécution (et non plus seulement des valeurs cumulatives) afin de déterminer l'état du cache (et du pipeline) avant l'exécution de chaque bloc de base. Pour estimer précisément le gain de temps d'une mémoire cache, l'état de cette dernière est propagé de bloc de base en bloc de base. Notons également que l'extension à d'autres DSP ne semble pas poser de difficultés majeures, même pour des architectures VLIW par exemple.

Les autres améliorations possibles concernent la possibilité pour l'utilisateur de plus interagir avec l'outil afin d'améliorer la précision des estimations ou demander à l'outil d'évaluer différentes solutions d'implémentation. On peut imaginer par exemple que l'utilisateur puisse spécifier le type de données (simple ou double précision par rapport à la largeur naturelle des données du DSP cible) ou le mode d'adressage utilisé pour tout ou partie du programme. En allant encore plus loin, il serait intéressant de donner la possibilité à l'utilisateur de définir ses propres instructions (caractérisées en nombre de cycles, de mots de mémoire, etc.) afin d'évaluer leur impact sur les performances globales de l'application.

D'autre part, dès que le code C de l'application comporte des routines écrites en assembleur, l'outil d'estimation devient inutilisable. On peut imaginer plusieurs scénarios permettant de prendre en compte dans les estimations un code écrit directement en assembleur. Si cela concerne une fonction entière ou un bloc de base complet, cela ne pose aucun problème. Par contre, la situation est plus délicate quand seules quelques instructions à l'intérieur d'un bloc de base sont écrites directement en assembleur. Une des solutions envisagées consiste à donner la possibilité à l'utilisateur d'interagir avec l'outil de manière à spécifier pour ces instructions le nombre de cycles et de mots de mémoire programme correspondants.

Perspectives

La méthode et l'outil d'estimation conçus et développés dans le cadre de cette thèse peuvent servir de base à de nombreuses extensions. Une première possibilité consiste à intégrer dans l'outil un modèle d'estimation de la puissance dissipée par le cœur lors de l'exécution d'un programme. La consommation est en effet un critère très important dans les systèmes numériques portables. Des études [Tiw94] ont montré qu'il est possible d'améliorer sensiblement la consommation du processeur en réécrivant le code source par exemple. Si l'on dispose de mesures de consommation pour chaque instruction et pour les enchaînements de deux instructions, l'utilisation de VESTIM peut permettre de déterminer la puissance dissipée lors de l'exécution du programme. L'outil peut aussi aider l'utilisateur dans l'analyse (puis l'optimisation) de l'impact de l'ordre des instructions sur la consommation.

Comme nous l'avons vu dans la partie résultat, lorsque le programmeur fait explicitement le choix de la localisation des données dans les différents bancs de mémoire,

les estimations du nombre de cycles d'un code optimisé peuvent augmenter. Il existe deux raisons à cela : soit l'estimateur a fait des hypothèses trop optimistes, soit le partitionnement effectué par l'utilisateur n'est pas optimal. Dans ce second cas, on pourrait compléter l'outil par une méthode permettant de déterminer le meilleur partitionnement des données en mémoire XRAM/YRAM. L'approche peut consister tout d'abord à rechercher dans le programme les opérations qui peuvent requérir des accès simultanés en mémoire. Grâce aux informations dynamiques (les x_i), on détermine alors un ordre de priorité sur les variables concernées par ces accès mémoires de manière à optimiser le partitionnement, donc les performances.

Un problème lié à notre méthode provient de l'utilisation d'une approche dynamique basée sur l'exécution du programme avec une séquence de tests pour extraire des informations dynamiques. Les valeurs recueillies sont des données statistiques qui ne peuvent pas garantir que la performance ainsi obtenue représente une mesure dans le pire cas. Bien que cette mesure ne soit pas indispensable pour une entreprise comme VLSI Technology, elle permet néanmoins de s'assurer que les contraintes temps-réel sont respectées. D'autre part, certaines applications comme V22bis ou AC-3 par exemple ne possèdent pas de séquences de test. Dans ce cas, il faut les construire ce qui peut représenter un travail important et délicat. Les approches statiques semblent une alternative intéressante lorsqu'il n'existe pas de séquence de test. D'un point de vue industriel cependant, il n'est pas envisageable d'utiliser une approche totalement statique pour une application complète. En effet, quelques soient les méthodes existantes [Pus89] [Gon93] [Mal97], les informations que doit fournir l'utilisateur rendent ces méthodes beaucoup trop contraignantes pour une utilisation en milieu industriel. Leur champ d'action doit se limiter à quelques fonctions (critiques) de l'application. Une approche mixte statique/dynamique proche de celle présentée dans [Ern97] (projet SYMTA) pourrait ainsi compléter notre méthode [Peg99b].

Les outils d'estimation peuvent également servir lors de la conception d'une nouvelle architecture et de son compilateur [Lie97][Yan98][Sam97]. A partir de mesures de performances et de statistiques sur l'utilisation des ressources, il est en effet possible d'optimiser les choix architecturaux ou le compilateur pour un certain type d'application. En comparant les performances du code généré avec l'estimation d'un code optimisé, le concepteur dispose d'une mesure de la qualité du compilateur. Dans [Sol96], des estimations de performance guident les choix pour l'optimisation du compilateur. Le modèle de description du processeur utilisé permet en outre de modifier aisément les contraintes sur les ressources (e.g. nombre d'accumulateurs) ou les chemins de données. De plus, nous disposons de statistiques sur les instructions utilisées (code généré et estimation) lors de l'exécution d'une application. Ces informations peuvent servir de base pour une réflexion sur le jeu d'instructions.

Le fait de pouvoir estimer les performances d'un code optimisé ne signifie pas forcément que le compilateur soit en mesure de générer un code de qualité équivalente. Il ne faut en effet pas oublier que la méthode d'estimation effectue des hypothèses fortes (e.g. pas de *spilling*). Néanmoins, il n'est pas exclu pour autant de pouvoir appliquer à la génération de code certaines idées utilisées lors de l'estimation. Cette question n'a pas été abordée.

Grâce à l'aspect multicible de la méthode d'estimation, il est également possible d'étendre l'outil afin de localiser les parties du code pour lesquelles l'utilisation d'un coprocesseur améliore les performances globales. Une approche consiste à fournir un modèle optimal du DSP cible, c'est-à-dire sans aucune contrainte sur les chemins de

données. En comparant les estimations, nous pensons être en mesure de déterminer quelles parties du code pourront être candidates pour d'éventuels coprocesseurs. Notons que le travail effectué lors du partitionnement matériel/logiciel « à la main » des normes G.726/G.721 sur les PineDSPCore et OakDSPCore constitue une base d'informations importantes dans cette perspective [Peg98b].

Annexe 1 : Les règles de réécritures du code C

Il existe des règles de réécriture indépendantes et dépendantes du DSP cible.

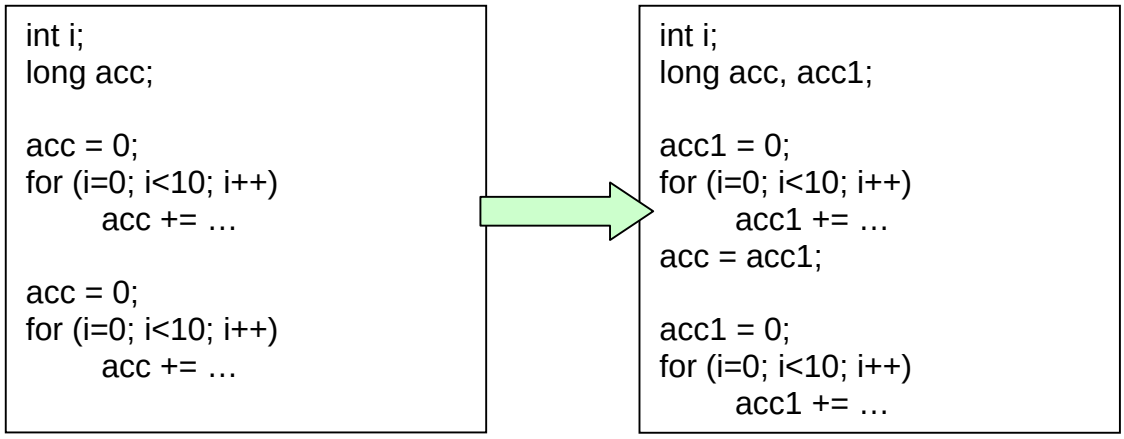
Les règles de réécriture indépendantes du DSP

- **L'utilisation des pointeurs**

Il est en reconnu [Lie96] que l'utilisation de pointeurs au lieu de tableaux indicés permet une utilisation plus efficace des modes d'adressage complexes des DSP.

- **Réduction de la durée de vie des variables**

Les DSP ont un nombre de registres très restreint. Réduire la durée de vie des variables peut aider le compilateur lors de la phase d'allocation des registres. Cette règle est particulièrement importante pour les parties de code entourant les boucles et les sections critiques de code. La réduction de la durée de vie des variables s'effectue généralement par l'utilisation de variables temporaires comme le montre l'exemple suivant.



- **Utilisation de variables locales**

Pour les parties de code critiques, il est préférable d'utiliser autant que possible des variables locales (i.e. éviter l'utilisation des variables globales statiques). Dans le cas du compilateur C du OakDSPCore par exemple, les variables servant à l'accumulation doivent obligatoirement être déclarées en variables locales. Dans le cas contraire, le compilateur effectuera une sauvegarde temporaire (*spilling*) de la donnée en mémoire. Néanmoins, dans certains cas il est souhaitable de déclarer des variables globales. L'utilisation de variables locales est par exemple très coûteux dans le cas de passage de paramètres.

- **Minimiser l'utilisation des variables de type « long »**

Pour un DSP 16 bits, le calcul sur des données de type « long » (32 bits) entraîne généralement l'appel à une routine d'une librairie (OAKLIB) très coûteuse en nombre de cycles. Ceci est particulièrement vrai pour les opérations de multiplication.

- **Limiter le nombre de variables et de pointeurs**

Afin de faciliter le travail du compilateur lors de la phase d'allocation de registres, il est préférable de garder le minimum de pointeurs et de variables. Cette règle est d'une utilisation délicate puisqu'elle augmente la durée de vie des variables. Pour cette raison la règle ne s'applique généralement que sur des parties restreintes du programme.

- **Factorisation du code**

La factorisation du code permet de localiser plus finement les parties critiques d'une application. L'appel aux fonctions n'est pas en général quelque chose de très coûteux.

Les règles de réécriture dépendantes du DSP

Nous décrivons ici les extensions que nous utilisons pour optimiser un code C pour le OakDSPCore. Nous indiquons également quelques autres extensions que nous avons pu trouver dans la littérature.

- **L'utilisation de type de données orienté DSP**

Les accumulateurs du OakDSPCore ont une largeur de 36 bits. Ils possèdent en effet 4 bits pour l'extension de signe mais offrent également une protection contre le dépassement de capacité lors d'opération sur des données de 32 bits[69]. Comme en C le type « long » est codé sur 32 bits, les 4 bits d'extension ne sont pas exploités par le compilateur. Afin de supporter cette précision sur 36 bits, un nouveau type a été ajouté au langage: acc_t [Occ96]. Ce type de données est particulièrement recommandé pour les opérations de type « MAC ».

Dans [Kre94] un type accumulateur de 40 bits est défini avec un bit de signe, 8 bits pour la partie entière et 31 bits pour la partie fractionnaire. L'utilisateur dispose aussi de nouveaux types lui permettant de représenter des données en point fixe (sur 16 ou 32 bits). Il est possible de fixer l'emplacement de la virgule, i.e. le nombre de bits pour la partie entière et fractionnaire. Dans [Hof93], l'auteur définit un type de donnée « complexe » ainsi qu'un ensemble de fonctions mathématiques utilisant ce type de données (e.g. FFT).

- **Les boucles matérielles**

Le compilateur C du OakDSPCore intègre un nouveau mot clé « __bkrep__ » qui, placé devant un « for », un « do » ou un « while », indique que le programmeur souhaite que le compilateur génère à cet endroit une instruction de boucle. En réalité, ce mot clé permet de relâcher des contraintes sur l'utilisation d'une instruction de boucle.

Cette extension est également présente dans les travaux présentés en [Hof93], [Kre94] et [Yos96].

[69] Jusqu'à 15 dépassements de capacité sont ainsi possibles.

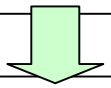
- **La localisation des données dans les différents bancs mémoires**

Le OakDSPCore possède une architecture de type Harvard modifiée conforme à celle définie dans le paragraphe 2.2.2.2. Le découpage de la mémoire donnée en deux bancs permet d'effectuer une opération « MAC » sur deux données en mémoire en un seul cycle. Les extensions (nouveaux mots clés) « __xram__ » et « __yram__ », insérées dans le code source, au moment de la déclaration des variables, permettent de renseigner le compilateur sur la localisation des données en mémoire X ou Y. Non seulement le compilateur va effectivement allouer les données dans le banc mémoire indiqué, mais la génération des instructions « MAC » est optimisée. On retrouve cette possibilité dans [Kre94] et [Yos96].

Exemple de l'utilisation de ces extensions pour le OakDSPCore avec un FIR

```
short a[10];
short b[10];
int i;
long l;

l = 0;
for (i=0; i<10; i++)
        l += (long)a[i] * (long)b[i];
```



```
__xram__ int a[10];
__yram__ int b[10];
__xram__ short *p1;
__yram__ short *p2;
int i;
acc_t l;
p1 = a;
p2 = b;

l = 0;
__bkrep__
for (i=0; i<10; i++)
        l += (long)*p1++ * (long)*p2++;
```

Références bibliographiques

[Aho76] A.V. Aho and S.C. Johnson – *Optimal code generation for expression trees*, Journal ACM, vol. 23, no. 3, pp. 488-501, July 1976.

[Aho86] A.V. Aho, R. Sethi, J.D. Ullman – *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[Aho89] A.V. Aho et al. – *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. Prog. Lang. and Syst., vol. 11, no. 4, pp. 491-516, Oct. 1989.

[Ac3] Digital Audio Compression Standard (AC-3) - *Advanced Television Systems Committee*, James C. Mc Kinney, Chairman, Dr. Robert Hopkins, executive Director. http://www.dolby.com/tech.

[AnDev] Analog Device Inc., Digital Signal Processing Applications Unsing the ADSP-2100 Family, vol. 1, 1992.

[Ara95a] Guido ARAUJO, S. Devadas, K. Keutzer, S. Liao, S. MALIK, S. Tjiang, A. Sudarsanam and A. Wang – *Challenges in Code Generation for Embedded Processors*. Code Generation for Embedded Processors : 1st International Workshop, Edited by Peter Marwedel and Gert Goossens, pp. 48-64, Kluwer Academic Publishers, 1995.

[Ara95b] Guido ARAUJO and Sharad MALIK – *Optimal Code Generation for Embedded Memory non homogeneous register Architectures*, In Proc. 8th International Symposium on System Synthesis, pp. 36-41, September 1995.

[Ara98] Guido ARAUJO and Sharad MALIK – *Code Generation for Fixed-Point DSPs*. ACM Transactions on Design Automation of Electronics Systems, Vol. 3, No 3, July 1998.

[Bar92] David H. Bartley – *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software-Practice and Experience, Vol. 22(2), pp. 101-110, February 1992.

[Bau97] Jean Claude Bauer, Etienne Closse, Eric Flamand, Michel Poize, Jacques Pulou et Patrick Venier – *SAXO: A retargetable optimized compiler for DSPs*, Proceedings ICSPAT, 1997.

[Bdt95] Berkeley Design Technology, Inc. – *Buyer's Guide to DSP Processors*, 1995.

[Bdt99] Berkeley Design Technology, Inc. – *Buyer's Guide to DSP Processors*, 1999.

[Bia98] Luc Bianco, Michel Auguin, Guy Gogniat et Alain Pegatoquet - *A Path Analysis Based Partitioning for Time Constrained Embedded Systems*, Workshop on Hardware/Software Codesign, Seattle, March 15-18, pp 84-90, 1998.

[Bia99a] Luc Bianco, Michel Auguin et Alain Pegatoquet - *A Prototyping Method of Embedded Real Time*, 25th EUROMICRO Conference, Milan, Italy, September 8 - 10, 1999

[Bia99b] Luc Bianco, Michel Auguin, Emmanuel Gresset et Alain Pegatoquet - *A System Prototyping Tool for Efficient System Architecture Exploration*, ICSPAT, Orlando, 1-4 November 1999.

[Bod97] Jim Boddie, Paul D'Arcy, Lucent Technologies, Allentown, Pa., Architectural innovations take next step, February 10, 1997, Issue: 940, Section: Digital Signal Processing, http://www.techweb.com/se/directlink.cgi?EET19970210S0093.

[Cam96] R. Camposano and J. Wilberg – *Embedded System Design*, Design Automation for Embedded Systems, an International Journal, pp. 5-50, 1996.

[Can94] M. Van Canneyt, K. Van Nieuwenhove, K. Cools, R. Jonckheere, P. Willekens and D. Genin - *Specification, Simulation and Implementation of a GSM Speech Codec with DSP Station™*, DSP & Multimedia Technology, vol. 3, No. 5, May 1994.

[Cha82] G. J. Chaitin - *Register Allocation and Spilling via Graph Colouring*, ACM SIGPLAN Notices, vol. 17, no. 6, pp. 98-105, June 1982.

[Cra97] Nicholas Cravotta, *A Look At DSP Development Tools.* Embedded Systems Programming, April 1997.

[Dav92] G. Davidson, W. Anderson and A. Lovrich – *A Low-Cost Adaptative Transform Decoder Implementation For High-Quality Audio*, Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 193-196, San Francisco, CA, USA, March 1992.

[Ele99] Electronique International N° 347 – *L'alliance Lucent-Motorola porte ses premiers fruits*, p. 19, 29 avril 1999.

[Ell85] John R. Ellis - *Bulldog: A compiler for VLIW Architectures.* The MIT Press, 1985

[Ern93] W. Ye, R. Ernst, Th. Benner and J. Henkel - *Fast Timing Analysis for Hardware-Software Co-Synthesis*, Proc. IEEE International Conference on Computer Design (ICCD), pp. 452-457, Cambridge, Massachusetts, USA, 1993.

[Ern95] W. Ye, R. Ernst and M. Trawny – *Worst Case Timing Estimation based on Symbolic Execution*, COBRA Report'95, Institute of Computer Engineering, Technical University of Braunschweig, Germany, October 1995.

[Ern97] R. Ernst and W. Ye – *Embedded program timing analysis based on path clustering and archietcture classification*, Proc. ICCAD, San Jose, USA, 1997.

[Fau95] A. Fauth. - *Beyond tool-specific machine descriptions*, In Code Generation for Embedded Processors, P. Marwedel and G. Goossens, Kluwer Academic Publisher, 1995.

[Fra92] C.W Framer, D.R Hanson and T.A. Proebsting - *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, No. 3, pp. 213-226, 1992.

[Gal97] Thierry Le Gall - *Le choix d'un compilateur devient primordial.* Electronique n° 74, pp. 44-48, Octobre 1997.

[Gon93] Jie Gong, Daniel D. Gajski and Sanjiv Narayan - *Software Estimation from Executable Specifications*, Technical Report ICS-93-5, March 8, 1993

[Goo92] Gert Goossens et al. – *Loop Optimization in register-transfer scheduling for DSP-systems*, Proc. 26th IEEE/ACM Design Automation Conference, June 1989.

[Goo97] Gert Goossens, Johan Van Praet, Dirk Lanneer, Werner Geurts, Augusli Kifli, Clifford Liem and Pierre G. Paulin, *Embedded Software in Real-Time Signal Processing Systems: Design Technologies*. Proceedings of the IEEE, Vol. 85, No. 3, March 1997.

[Got96a] Martin Gotschlich and Bernard Wess – *Automatic Generation of constrained Expression Trees for Global Optimized DSP Assembly Code*, Proceedings of ICSPAT, vol. 1, Boston, USA, pp. 732-736, October 1996.

[Got96b] Martin Gotschlich, Werner Kreutzer and Bernard Wess – *REDACO: A Retargetable Data Flow Graph Compiler for Digital Signal Processors*, Proceedings of ICSPAT, vol. 1, Boston, USA, pp. 742-746, October 1996.

[Had97] George Hadjiyiannis, Silvina Hanono and Srinivas Devadas, *ISDL: An Instruction Set Description Language for Retargetability*. Design Automation Conference 97, Anaheim, California.

[Had99] George Hadjiyiannis, Pietro Russo and Srinivas Devadas - *A Methodology for Accurate Performance Evaluation in Architecture Exploration*. Design Automation Conference 99, New Orleans, Louisiana, USA.

[Hen90] J. Hennessy and D. Patterson - *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hil90] P. Hilfinger et al. – *DSP Specification Using the Silage Language*. Proc. of ICCASP, 1990.

[Hof93] Marc Hoffman, Analog Devices, Inc, Numerical C Enhances Coding of Signal Processing Algorithms, DSP Applications, December 1993.

[Hor98] E. van der Horst, W. Kloosterhuis and J. van der Heyden – *A C Compiler for Embedded R.E.A.L. DSP Architecture*, Proc. ICSPAT Conference, vol. 1, pp. 464-468, Toronto, Canada, September 13-16, 1998.

[Kal93] A. Kalavade and E.A. Lee - *A Hardware/Software Codesign Methodology for DSP Applications*, IEEE Design and Test of Computers, vol. 10, no. 3, pp. 16-28, September 1993

[Kie98] P. Kievits, E. Lambers, C. Moerman and R. Woudsma – *R.E.A.L. DSP Technology for Telecom Basedband Processing*, Proc. ICSPAT Conference, vol. 1, pp. 327-331, Toronto, Canada, September 13-16, 1998.

[Kif96] A. Kifli - *Global Scheduling in high-level synthesis and code generation for embedded processors*, Rapport de Thèse, Université de Leuven, Belgique, Nov. 1996.

[Kre94] Benjamin Krepp, Intermetrics Inc. - *DSP-Oriented Extensions to ANSI C*, Proc. ICSPAT, pp. 695-704, October 1994.

[Kwi99] Laurent Kwiatkowski – *Utilisation des méthodes de résolution ILP pour la détermination de bornes de temps d'exécution d'une application sur un processeur cible*, Laboratoire Informatiques Signaux et Systèmes I3S/ CNRS, Rapport de Recherche n° 99-16, octobre 1999.

[Lam88] M. Lam - *Software Pipelining: An effective scheduling technique for VLIW Machines*, Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation, pp. 318-328, 1995.

[Lan80] D. Landskov, S. Davidson and B. Shriver - *Local Microcode Compaction Techniques*, Journal Computing Survey, vol. 12, No 3, 1980.

[Lan95] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen et G. Goossens - *CHESS: Retargetable Code Generation for Embedded DSP Processors*, Code Generation for Embedded Processors, Kluwer Academic Publishers, pp. 85-102, 1995.

[Lee88] Edward A. LEE – *Programmable DSP Architectures: Part 1*. IEEE ASSP Magazine, pp. 4-19, October 1988.

[Lee89] Edward A. LEE – *Programmable DSP Architectures: Part 2*. IEEE ASSP Magazine, pp. 4-14, 1989.

[Lee96] Dae-Hyun Lee and Seung Ho Hwang - *Abstract Simulator: A DSP Software Timing Analysis Tool*, pp. 890-894, ICSPAT'96.

[Lee97] Sang-Min Lee, Sangil Park and Youngbeom Jang – *Cost-Effective Implementation of ITU-T G.723.1 on a DSP Chip*, Proceedings of the 1998 IEEE International Symposium on Consumer ElectronicsProc (ISCE), pp. 31-34, 2-4 December 1997.

[Leu96] Rainer Leupers and Peter Marwedel – *Algorithms for Address Assignment in DSP Code Generation*, ICCAD96.

[Leu97] Rainer Leupers – *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997.

[Leu98a] Rainer Leupers and Peter Marwedel – *Retargetable Code Generation based on Structural Processor Description*, Desing Automation for Embedded Systems, vol. 3, no. 1, pp. 1-36, Jan 1998.

[Leu98b] Rainer Leupers – *Novel Code Optimization Techniques for DSPs*, 2nd European DSP Education and Research Conference, Paris, France, Sep. 1998.

[Lev97] Markus Levy, C Compilers for DSPs: Flex their Muscles, EDN June 1997, pp 93-102.

[Lia96] S. Liao – *Code Generation and Optimization for embedded digital signal processors*, Rapport de Thèse, MIT, Juin 1996.

[Lie95] C. Liem, P. Paulin, M. Cornero and A. Jerraya - *Industrial Experience using Rule-Driven Retargetable Code Generation for Multimedia Applications,* 8th Symposium on System Level Synthesis, September 1995.

[Lie96] C. Liem, P. Paulin and A. Jerraya - *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd DAC, Las Vegas, Nevada, June 3-7, 1996.

[Lie97] C. Liem – *Compilateurs Multicibles et Outils pour les Processeurs Embarqués dans le cadre d'Applications industrielles*, Rapport de thèse, Laboratoire TIMA-INPG, Grenoble, 1997.

[Lin94] Wen-Yen Lin, Corinna G. Lee and Paul Chow - *An Optimizing Compiler for the TMS320C25 DSP Chip*, Proc. ICSPAT, pp. 689-694, 1994.

[Mal95] Y-T. S. Li and S. Malik - *Performance Analysis of Embedded Software Using Implicit Path Enumeration*, *32nd DAC*, pp. 456-461, San Francisco, CA, June 1995.

[Mal96] Y-T. S. Li, S. Malik and A. Wolfe – *Cache modeling for Real-Time Software: Beyond Direct mapped instruction caches*, In Proceedings of the 17th IEEE Real-Time Systems Symposium, pp.254-263, December 1996.

[Mal97] S. Malik, M. Martonosi and Y-T.S. Li. - *Static Timing Analysis Of Embedded Software*, *34th DAC*, pp. 147-152, Anaheim, CA, 1997.

[Man99] NS. Manju Nath – *C Compilers and development tools Simplify DSP assembly-language programming*, EDN Magazine, pp. 103-110, January 21st 1999.

[Mar95a] Peter Marwedel and Gert Goossens, editors - *Code Generation for Embedded Processors : 1st International Workshop*. Kluwer Academic Publishers, Boston, Massachusetts, 1995. Proceedings of the 1994 Dagstuhl Workshop on Code Generation for Embedded Processors.

[Mar95b] Peter Marwedel and Gert Goossens - *Code Generation for embedded processors*, Kluwer Academic Publisher, 1995.

[Mar95c] Peter Marwedel - *Code Generation for embedded processors: An introduction*, pp. 14-31, In Code Generation for Embedded Processors, P. Marwedel and G. Goossens, Kluwer Academic Publisher, 1995.

[Mes99a] Vincent Messé et François Charot - *Définition interactive d'ASIP : utilisation d'une bibliothèque de modules flexibles de génération de code*. 5ème Symposium sur les architectures Nouvelles de Machines, Rennes, 8-11 juin 1999.

[Mes99b] Vincent Messé – *Production de compilateurs flexibles pour la conception de processeurs programmables spécialisés*. Rapport de Thèse, Mars 1999.

[Mot56k] Motorola Inc., DSP56000/DSP56001 User's Manual, 1990.

[Oak96] VVF3500 DSP core User Manual, VLSI Technology, Inc.

[Occ96] VVF3500 C-Compiler User Manual, VLSI Technology, Inc.

[Ova98] B-S. Ovadia, W. Gideon and B. Eran – *Multiple and Parallel Execution Units in Digital Signal Processors*, Proc. ICSPAT Conference, vol. 2, pp. 1491-1497, Toronto, Canada, September 13-16, 1998.

[Ott97] Greger Ottoson and Mikael Sjödin – *Worst Case Execution Time Analysis for Modern Hardware Architectures*, SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems, 1997.

[Par89] Chang Yun Park and Alan C. Shaw – *Experiments With A Program Timing Tool Based On Source-Level Timing Schema*, The Journal of Real-Time Systems, vol. 1, n°2, pp. 160-176, September 1989.

[Par90] Chang Yun Park and Alan C. Shaw – *Experiments With A Program Timing Tool Based On Source-Level Timing Schema*, Proc. 11th IEEE Real-Time Systems Symposium, pp. 72-81, 1990.

[Par93a] Matthew F. Parkinson, Paul M. Taylor and Sri Parameswaran – *A Profiler for Automated Translation of Signal Processing Algortihms into High Speed Hardware/Software Hybrid Architectures*, Microelectronics, Gold Coast Australia, October 5-8, 1993.

[Par93b] Matthew F. Parkinson, Paul M. Taylor and Sri Parameswaran – *An Automated Hardware/Software Codesign (HSC) using VHDL*, APCHOLSA, Brisbane, Australia, December 6-9, 1993.

[Pat82] D. Patterson and C. Séquin - *A VLSI RISC*, IEEE Computers, pp. 8-21, Sep. 1982.

[Pau95] P. Paulin et al. - *Flexware: A flexible firmware development environment for embedded systems*, In Code Generation for Embedded Processors, P. Marwedel and G. Goossens, Kluwer Academic Publisher, pp. 67-84, 1995.

[Pau97] P. Paulin, C. Liem, M. Cornero, F. Naçabal and G. Goossens - *Embedded Software in Real-Time Signal Processing Systems: Application and Architectural Trends*, Proceedings of the IEEE, Vol. 85, No. 3, March 1997.

[Pee99] S. Pees, A. Hoffmann, V. Zivojnovic and H. Meyr – *LISA : Machine Description Language for Cycle-Accurate Models of Programmablke DSP Architectures*. Proc. 36th DAC, June 21-24, New Orleans, LA, USA, 1999.

[Peg98a] A. Pegatoquet, M. Auguin, G. Gogniat and E. Gresset - *Software Estimations for DSPs – Improved Design Flow Shortens Time-to-Market.* Proc. ICSPAT, Vol. 1, pp. 805-808, Toronto, Canada, September 14-16, 1998.

[Peg98b] A. Pegatoquet, M. Auguin and E. Gresset - *Improving Performance VS Silicon size tradeoffs using coprocessors. A case study: G.721 on Oak and Pine DSP Cores*. Proc. EUSIPCO, Vol. 1, pp. 467-470, Isles of Rhodes, Greece, September 7-11, 1998.

[Peg99a] A. Pegatoquet, M. Auguin, L. Bianco and E. Gresset – *Rapid Development of Optimized DSP Code From a High Level Description Through Software Estimations*. 36th Design Automation Conference, June 19-24, New Orleans, Louisiana, USA, 1999.

[Peg99b] A. Pegatoquet, M. Auguin, L. Kwiatkowski, L. Bianco et E. Gresset – *VESTIM : Une méthode d'estimation de performances pour une implémentation optimisée d'applications sur processeurs de traitement du signal*. 17ème Colloque GRETSI sur le traitement du signal et des images, Vannes, 13-17 Septembre, 1999.

[Peg99c] A. Pegatoquet, M. Auguin and E. Gresset - *DSP Code Optimization Using Estimation Metrics – A Case Study: G.728 on the OakDSPCore.* International Conference on Signal Processing Applications and Technology (ICSPAT), Orlando, November 1-4, 1999.

[Pus89] P. Puschner and Ch. Koza - *Calculating the Maximum Execution Time of Real Time Programs*, The Journal Of Real Time Systems, Vol. 1, No. 2, pp. 159-176, Sept 1989.

[Rec84] ITU-T Recommendation G.721 - *32 kbit/s Adaptative Differential Pulse Code Modulation*, 1984.

[Rec94] ITU-T Recommendation G.728 - *Coding of Speech at 16 kbit/s using Low-Delay Code Excited Linear Prediction*, 1994.

[Rec95] ITU-T Recommendation G.729 - *Coding of Speech at 8 kbit/s using Conjugate-Structure Algebraic Code-Excited Linear Prediction*, 1995.

[Rec96] ITU-T Recommendation G.723.1 – *Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, Mars 1996.

[Sag94] Mazen A.R. Saghir, Paul Chow, and Corinna G.Lee - *Towards Better DSP Architectures and Compilers*. Proc. ICSPAT'94, pp. 658-664.

[Sak98] Rizos Sakellariou, Christine Eisenbeis and Peter Knijnenburg - *Efficient Implementation of the ROW-Column 8x8 IDCT on VLIW Architectures*, Proc. of EUSIPCO, vol. 2, pp. 869-872, Isles of Rhodes, Greece, September 7-11, 1998.

[Sam97] Rajeshkumar S. Sambandam and Xiabo Hu – *Predicting Timing Behavior in Architectural Design Exploration of Real-Time Embedded Systems*, Proc. 34th Design Automation Conference, Anaheim, California, USA, 1997.

[Sha89] Alan C. Shaw – *Reasoning about Time in Higher-Level Language Software*, IEEE Transactions on Software Engineering, vol. 15, n°7, pp.875-889, July 1989.

[Sol96] Marc SOLER et al. - *An Embedded DSP Platform for multi-standard ITU G.728, G.729 and G.723.1 audio compression*. Proc. ICSPAT, Boston, MA, October 7-10, 1996.

[Sta94] Richard Stallman – *Using and Porting GNU-CC Version 2.6*, Free Software Fundation, Inc.

[Ste91] Michel Stein – *Les Modems pour la Transmission de Données*, Collection Technique et Scientifique des télécommunications, Edition MASSON, 1991.

[Suc98] R. Sucher, R. Niggebaum, G. Fettweiss and A. Rom – *CARMEL : A New High Performance DSP Core Using CLIW*, Proc. ICSPAT Conference, vol. 1, pp. 499-504, Toronto, Canada, September 13-16, 1998.

[Tiw94] V. Tiwari, S. Malik and A. Wolfe – *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*, IEEE Transactions on VLSI Systems, December, 1994.

[Tur97] Jim Turley and Harri Hakkarainen - *TI's New'C6x DSP Screams at 1,600 MIPS*, Microprocessor Report, February 17, 1997.

[Wes95] B. Wess - *Code Generation Based on Trellis Diagrams*, In Code Generation for Embedded Processors, P. Marwedel and G. Goossens, Kluwer Academic Publisher, pp. 188-202, 1995.

[Wil94] Robert Wilson, Robert French, Christopher Wilson, Saman Amasinghe, Jennifer Anderson, Steve Tjiang, Shuh-Wei Liao, C.W Tseng, M. Hall, M. Lam and J. Hennesy - *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, Rapport Technique, Stanford University, mai 1994.

[Wil96] Markus Willems and Vojin Zivovjnovic - *DSP Compiler: Product Quality for Control Dominated Applications?*, Proc. ICSPAT, pp. 752-756, October 1996.

[Woo98] C. Woodthorpe and K. Stone – *Wireless DSP Development with C compilers*, Communication Systems Design, pp. 28-35, November 1998.

[Yan98] Jin-Hyuk Yang et al. - *MetaCore: An Application Specific DSP Development System*, 35th DAC, pp. 800-803, San Francisco, CA, 1998.

[Yos96] Kenichi Yoshi, Hiroaki Chiba, Hideki Fujita, Mototaka Sone and Koichi Nakamura - *The development of Extended C language and its compiler for DSP*, Proc. ICSPAT, pp. 679-683, October 1996.

[Ziv94] Vojin Zivovjnovic, Juan Martinez Velarde, Christian Schläger and Heinrich Meyr - *DSPSTONE: A DSP-Oriented Benchmarking Methodology*, Proc. ICSPAT, pp. 715-720, 1994.

[Ziv96] Vojin Zivovjnovic, Stefan Pees, Christian Schläger and Heinrich Meyr - *LISA - Machine Description Language and Generic Machine Model for HW/SW Codesign*, IEEE Workshop on VLSI Processing, San Francisco, October 1996.

[Ziv96a] Vojin Zivovjnovic and Heinrich Meyr - *Compiled HW/SW Co-Simulation*, 33rd Design Automation Conference, Las Vegas, NV, USA, June 1996.

[Ziv96b] Vojin Zivovjnovic, Stefan Pees, Christian Schlager, Markus Willems, Rainer Schoenen and Heinrich Meyr - *DSP Processor/Compiler Co-Design: A Quantitative Approach*, Proc. ICSPAT, pp. 679-683, October 1996.

Glossaire

AGU	:	**A**ddress **G**eneration **U**nit
ALU	:	**A**rithmetic and **L**ogic **U**nit
ASIC	:	**A**pplication **S**pecific **I**ntegrated Circuit
ASIP	:	**A**pplication **S**pecific **I**nstruction set **P**rocessor
BFO	:	**B**it **F**ield **O**perations
BMU	:	**B**it **M**anipulation **U**nit
CBU	:	**C**omputation and **B**it manipulation **U**nit
CFG	:	**C**ontrol **F**low **G**raph
CISC	:	**C**omplex **I**nstruction **S**et **C**omputer
CU	:	**C**omputation **U**nit
DAAU	:	**D**ata **A**ddressing **A**rithmetic **U**nit
DAG	:	**D**irected **A**cyclic **G**raph
DFG	:	**D**ata **F**low **G**raph
DIR	:	**D**SP **I**ntermediate **R**epresentation
DSP	:	**D**igital **S**ignal **P**rocessor
FFT	:	**F**ast **F**ourier **T**ransform
FIR	:	**F**inite **I**mpulse **R**esponse
GCC	:	**G**NU **C** **C**ompiler
GNU	:	**G**nu is **N**ot **U**nix
ILP	:	**I**nteger **L**inear **P**rogramming
ISA	:	**I**nstruction **S**et **A**rchitecture
ITU	:	**I**nternational **T**elecommunication **U**nion
MAC	:	**M**ultiply and **AC**cumulate
MIPS	:	**M**illion of **I**nstruction **P**er **S**econd
OCC	:	**O**akDSPCore **C** **C**ompiler
PC	:	**P**rogram **C**ounter
PCU	:	**P**rogram **C**ontrol **U**nit
RISC	:	**R**educed **I**nstruction **S**et **C**omputer
RTL	:	**R**egister **T**ransfer **L**anguage
VLIW	:	**V**ery **L**ong **I**nstruction **W**ord

Résumé

Les compilateurs C pour processeurs DSP actuellement disponibles sont généralement incapables de générer un code assembleur respectant les contraintes temps réel fortes des systèmes embarqués. Les coûts de développement élevés associés au codage manuel d'applications sur DSP et la pression sans cesse plus forte du « time-to-market » rendent cette situation de plus en plus inacceptable pour les entreprises et militent en faveur d'une approche de haut niveau basée sur l'utilisation de compilateurs. Or, si les compilateurs pour DSP sont globalement inefficaces, il est toutefois possible d'améliorer de manière significative les performances du code assembleur généré en modifiant le code C d'origine pour le compilateur cible (i.e. l'architecture cible) sur les parties de code critiques de l'application.

Ces problèmes ont motivé l'élaboration de nouveaux outils permettant d'accélérer ce processus. Nous proposons pour cela d'utiliser des méthodes d'estimations logicielles qui fournissent, à partir d'une description en C de l'application, d'une part les performances du code assembleur généré sans utiliser de simulateur de niveau instruction et d'autre part les performances d'un code assembleur optimisé. Ce dernier code correspond à une estimation d'un code écrit par un programmeur expérimenté. Par comparaison des deux performances il est aisé de localiser rapidement les parties à optimiser dans le code C de l'application. Par cette approche on limite ainsi aux parties réellement critiques, identifiées par la méthode, la nécessité de développer du code assembleur (si nécessaire). Le modèle d'estimation utilisé est multicible et se base sur une représentation intermédiaire orientée schéma de calcul DSP. De nombreuses expérimentations sur des applications industrielles illustrent l'intérêt de l'approche.

Mots-Clés

Processeur de traitement du signal (DSP), Estimation de performance logicielle, Développement de code assembleur, Compilation, Systèmes embarqués.

Abstract

Current DSP C compilers are generally unable to produce efficient assembly code. In order to respect tight real-time constraints of embedded systems, programmers commonly write DSP code by hand. However programming in assembly language becomes increasingly difficult since DSP applications are becoming larger and more complex. Programming DSP applications in high level language such as C is becoming more prevalent to reduce the development costs, thus the time-to-market. Moreover, it is well known that it is possible to improve the quality of generated assembly code by modifying the original C source code for the target compiler (i.e. target DSP).

In order to make more efficient use of DSP C compilers, and minimize the need to write assembly code by hand, we propose a methodology based on software estimations. A performance evaluation of the code generated by the C compiler is first provided. Then to evaluate the quality of this code, an estimation of an optimized assembly code is also computed from an intermediate representation of GNU-based C compiler. This metric represents the performance of the code as if it had been hand written by an experienced programmer. By comparing this estimation with the generated assembly code, it is easier to determine if the C compiler has produced efficient code and locate parts of the application that need to be optimized. Thus software estimations guide programmers for optimizing time-critical routines of the application. The estimation model is retargetable and based on a DSP intermediate representation. Experimentation with industrial applications has illustrated the interest of this approach.

Key words

Digital Signal Processors (DSP), Software Estimation, Assembly code development, Compilation, Embedded Systems.