Infrastructures et stratégies de compilation pour parallélisme à grain fin

Erven Rohou

N° d'ordre : 2023

THÈSE

Présentée devant

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

PAR

Erven Rohou

Équipe d'accueil : IRISA
École Doctorale : Sciences Pour l'Ingénieur
Composante universitaire : IFSIC

Titre de la thèse :

Infrastructures et stratégies de compilation pour parallélisme à grain fin

soutenue le 17 novembre 1998 devant la commission d'examen

COMPOSITION DU JURY :

M. :	Jean-Pierre	Banâtre	Président
MM. :	Christine	Eisenbeis	Rapporteurs
	Jesús	Labarta	
MM. :	François	Bodin	Examinateurs
	Philippe	Sarazin	
	André	Seznec	

Remerciements

Je remercie Jean-Pierre Banâtre qui m'a fait l'honneur de présider ce jury.

Je remercie Christine Eisenbeis et Jesús Labarta pour avoir accepté la lourde tâche de rapporteur de cette thèse, et Philippe Sarazin pour avoir accepté d'être membre du jury.

J'adresse mes plus chaleureux remerciements à François et André qui ont conjointement dirigé cette thèse. Leur soutien, leur enthousiasme et leur disponibilité ont grandement contribué à l'aboutissement de ces travaux.

Je remercie tous les membres du projet européen OCEANS (Bas Aarts, Michel Barreteau, François Bodin, Fabian Breg, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Toru Kisuki, Peter Knijnenburg, Paul van der Mark, Mike O'Boyle, Rizos Sakellariou, Henk Schepers, André Seznec, Elena Stöhr, Menno Treffers, Marco Verhoeven, Harry Wijshoff). Ils ont contribué à la réalisation concrète du prototype.

Je tiens aussi à remercier, dans un ordre aléatoire :
- Huguette pour son efficacité,
- Dany, Laurence et Pierrette pour le soutien logistique,
- l'atelier Irisa,
- Marie-Anne pour sa patience et aussi pour l'organisation du pot,
- Luc pour les «défis shell» et les explications sur le fonctionnement d'Excel,
- Pierre pour les discussions passionnantes sur l'entropie de l'univers, la vitesse des électrons dans un fil, ou le temps de vidange d'un évier,
- Seb pour ses explications sur l'architecture des processeurs et pour les gâteaux indiens,
- Zbigniew pour les «chrusts»,
- Yvon pour ses connaissances encyclopédiques et antédiluviennes,
- Dan pour son esprit critique,
- Yann pour son esprit taquin,
- Stéphane pour C/C++/Fortran/Lanfeust de Troy/Matlab/Motif,
- et tous les autres membres passés et présents, temporaires et permanents de l'équipe Caps.

Pour reprendre la formule de l'ADOC, «Merci à tous, c'est vraiment trop sympa !»

Table des matières

Table des figures

Introduction

Contexte

Il est aujourd'hui utopique de vouloir optimiser à la main l'ensemble d'une application. Sur certains systèmes, en particulier les systèmes enfouis, les noyaux de calculs, parties critiques des applications, sont encore écrits directement en assembleur parce qu'un programmeur parvient souvent à gagner un facteur d'accélération supérieur à dix par rapport à un compilateur. Cependant l'augmentation constante de la taille des applications [119] pousse à l'abandon de ces pratiques et à l'utilisation d'outils d'aide à l'analyse et à l'optimisation. Aujourd'hui un processeur exécute les instructions dans le désordre, dispose de plusieurs pipelines d'exécution et de nombreuses unités fonctionnelles, prédit plusieurs branchements en avance, dispose d'une hiérarchie mémoire complexe. Pour espérer écrire un code optimal, le développeur devrait comprendre l'ensemble de ces mécanismes et leurs interactions lors de l'exécution de son programme. Un compilateur — ou un outil d'optimisation — est plus à même de connaître les détails du fonctionnement du matériel et de les exploiter efficacement.

La recherche de performance est souvent un problème critique sur les systèmes enfouis. Les processeurs utilisés, souvent des DSP, ne sont pas réguliers, compliquent le processus et limitent sévèrement les performances des outils automatiques. Ici un manque d'outils se fait sentir et les besoins sont grands.

Pour exploiter pleinement les caractéristiques architecturales des processeurs, la communauté scientifique a développé un grand nombre de transformations de code appropriées : la hiérarchie mémoire a engendré le *padding*, le *skewing* et le placement de données, la prédiction des branchements a permis le raffinement du placement de code, les instructions conditionnelles ont conduit à l'analyse des prédicats, etc.

Cependant l'interaction des transformations entre elles est un phénomène complexe à deux égards :

- l'application successive de plusieurs transformations fait apparaître des conflits qui ne permettent pas d'exploiter la totalité de leur potentiel. Le gain qui résulte de l'enchaînement de plusieurs optimisations est fréquemment moindre que la somme des gains théoriques de chacune d'entre elles ;
- le choix de l'application d'une optimisation à un fragment de code a des répercussions sur les performances globales de l'application et donc sur les optimisations appliquées à d'autres fragments. Dans le cas du développement d'une application pour un système enfoui, par exemple, la place disponible pour le code est

constituée d'une mémoire de taille fixée. La décision de déplier une boucle n'est pas sans conséquence sur d'autres parties du programme, même indépendantes, puisque l'espace disponible est restreint. Il est possible qu'une autre boucle ne puisse pas être dépliée d'un facteur suffisant. Ainsi toute décision prise localement a un impact global.

Le premier de ces problèmes n'a été que peu abordé. Si l'interaction de deux ou trois techniques a fait l'objet de publications, peu de résultats traitent du choix des séquences d'optimisations dans sa globalité. Le deuxième problème, à notre connaissance, n'a actuellement pas été étudié.

Thèse

Nous sommes convaincus qu'une stratégie est nécessaire : le choix définitif d'une chaîne de transformations immuable, appliquée à tous les programmes a montré ses limites. Il est temps de considérer la compilation comme un processus itératif, dans lequel différentes optimisations s'échangent des informations, s'évaluent et peuvent se remettre en cause. La complexité des interactions rend hasardeuse l'estimation des performances dues à plusieurs actions. C'est pourquoi nous proposons de prendre des points de mesures, éventuellement en exécutant le programme, de façon à connaître certaines caractéristiques du code *a posteriori*. Les transformations agissent comme des entités autonomes et se communiquent des informations sur leurs actions. Nous avons montré que le coût de cette approche reste raisonnable malgré un algorithme de recherche relativement simple. Il fournit cependant des performances supérieures à celles obtenues par des techniques statiques.

L'analyse et l'optimisation ne doivent pas non plus se contenter d'une estimation de comportement locale à un fragment de code. Au contraire, le comportement global de l'application doit être considéré afin de prendre en compte les interactions entre les différents fragments de code.

Nous pensons qu'une véritable infrastructure s'impose pour aider le programmeur à analyser son programme, pour déterminer les sections critiques et les goulets d'étranglement, et pour améliorer les performances en prenant en compte toutes les caractéristiques matérielles (remplissage du pipeline, utilisation des registres, hiérarchies mémoire, ...). Cette infrastructure doit être entièrement paramétrable de façon à être rapidement adaptable à une nouvelle architecture matérielle.

Le développement d'une infrastructure complète de compilation et d'optimisation est hors de portée d'une seule équipe. Notre contribution dans ce domaine est le développement du système SALTO (*System for Assembly Language Tranformation and Optimization*). Cette infrastructure unique à notre connaissance permet à l'utilisateur d'analyser mais aussi de modifier un programme assembleur sans se préoccuper de l'inévitable analyse syntaxique ou de la construction du graphe de flot. Les structures de données sont construites automatiquement et sont accessibles grâce à une interface utilisateur orientée objet. L'architecture et le jeu d'instructions sont décrits dans un fichier externe à l'aide un langage proche de Lisp. Le niveau de détail est laissé à l'ap-

préciation de l'utilisateur. Le système SALTO a été utilisé pour des expérimentations dans des domaines divers en sus de celles présentées dans ce document : placement de code, instrumentation, simulation, ... Il a été distribué à plusieurs équipes en France et à l'étranger, et est en cours d'évaluation par des industriels.

La représentation du code de SALTO a ensuite été abstraite pour aboutir à une nouvelle infrastructure, appelée SEA (SALTO *Enhanced Abstraction*), plus adaptée à l'exploration des transformations de programme et à l'étude de leurs interactions. Son mérite est de cacher les détails non significatifs et présenter de permettre la manipulation de concepts de «plus haut niveau» tels que les nids de boucle. SEA dispose d'une représentation abstraite des fragments de code et d'un ensemble de transformations. Celles-ci possèdent un certain nombre de fonctionnalités comme tester leur légalité, s'appliquer, évaluer leur impact. Ceci permet au programmeur de se dégager d'un grand nombre de détails d'implémentation, le développement d'une stratégie est ainsi facilité.

Plan du document

Le premier chapitre de ce document présente un état de l'art dans le domaine de la compilation et des optimisations de code pour la performance. Après avoir abordé les principaux goulots d'étranglement des architectures actuelles, nous présentons les techniques classiques et récentes de transformations de code utilisées pour y remédier. Nous détaillons ensuite les techniques d'analyse de code dynamiques, qui permettent de déterminer les facteurs limitant les performances et de focaliser les optimisations sur les zones critiques. Nous montrons comment l'utilisateur peut intervenir pour signaler les propriétés qu'il connaît au compilateur puis nous passons en revue quelques outils d'optimisations. Nous décrivons alors plus précisément le contexte de nos travaux.

Dans le deuxième chapitre nous présentons deux infrastructures logicielles que nous avons développées dans le cadre de cette thèse. La première, appelée SALTO, est une bibliothèque de fonctions qui facilitent la manipulation de code assembleur. L'architecture et le jeu d'instructions sont décrits dans un fichier externe, ce qui permet de modifier rapidement les caractéristiques architecturales, voire de changer de processeur. Nous avons utilisé SALTO comme base de travail pour l'environnement SEA. SEA fournit une abstraction du programme assembleur. Il manipule des notions telles que les boucles ou les superblocs, plus adaptées à l'optimisation de code que les listes de mnémoniques. SALTO et SEA sont été utilisés pour mener à bien les études de cette thèse.

Le troisième chapitre décrit le problème de l'interaction entre les optimisations appliquées à différents fragments de code. En particulier nous présentons la difficulté de trouver un compromis entre la taille du code final et ses performances, critique dans le domaine des systèmes enfouis. L'approche GCDS (*Global Constraints-Driven Strategy*) que nous présentons permet la recherche globale d'une solution satisfaisant à un ensemble des contraintes.

Dans le dernier chapitre, nous généralisons le problème de l'interaction entre trans-

formations de code et nous montrons l'intérêt de la compilation itérative : dès lors que le temps de compilation n'est pas une contrainte, un système d'optimisation peut explorer l'espace des transformations de code de façon à évaluer l'intérêt des transformations et se déplacer vers celles qui offrent la meilleure performance. Nous avons développé un prototype pour valider notre approche et nous montrons que cette stratégie permet de mieux contrôler les propriétés du code produit que ne le font aujourd'hui les compilateurs classiques.

Chapitre 1

Optimisation de code

Les performances d'une application dépendent de trois paramètres : l'algorithme utilisé par le programmeur pour résoudre son problème, l'architecture matérielle utilisée pour exécuter le programme et l'efficacité du compilateur.

Le travail accompli par le compilateur consiste à adapter l'organisation des calculs et l'utilisation des données de l'application pour exploiter au mieux une architecture. Pour cela il dispose de transformations de code. La difficulté majeure de la conception d'un compilateur consiste à définir une stratégie capable de déterminer quelles transformations appliquer et avec quels paramètres. Les techniques actuelles appliquent généralement une suite de transformations dans un ordre prédéfini.

La recherche de performance est un point critique pour les systèmes enfouis. La problématique de la compilation est légèrement différente de la compilation pour processeur à usage général et nous en donnons les caractéristiques essentielles. Nous illustrons nos propos avec l'exemple du processeur VLIW TriMedia de Philips.

L'optimisation est précédée d'une phase d'analyse qui permet de localiser les sections d'une application qui n'exploitent pas au mieux l'architecture matérielle. L'analyse dynamique est une technique importante pour l'extraction d'informations précises sur le déroulement d'un programme.

Dans ce chapitre nous commençons par donner une vue générale du processus de compilation classique et nous illustrons la structure des compilateurs. Dans la deuxième section nous présentons les caractéristiques architecturales des processeurs actuels qui peuvent constituer un goulot d'étranglement. Nous détaillons alors des techniques d'optimisations utilisées pour adapter l'organisation des calculs à une architecture particulière. Dans la quatrième section nous développons le *profiling* [1], une technique d'analyse indispensable à la détermination des facteurs limitant la performance. Nous abordons alors la possibilité d'une interaction avec le programmeur dans la section 5. La section 6 présente quelques outils existants. La dernière section précise le contexte de nos travaux.

1. Un certain nombre de termes anglais sont couramment utilisés tel quel et il n'existe pas de consensus pour leur traduction. Nous avons préféré les franciser pour éviter des lourdeurs inutiles ou des traductions hasardeuses.

1.1 Compilation

Le fonctionnement classique d'un compilateur fait appel à un certain nombre de phases qui transforment le code source écrit par le programmeur en un fichier exécutable par étapes successives. Nous appelons «haut niveau» les fichiers écrits dans un langage de programmation évolué, comme le C ou le Fortran, et les transformations qui les concernent. Le terme «bas niveau» réfère aux programmes écrits en langage assembleur ou en code machine.

La chaîne de transformations qui permet le passage du haut niveau au bas niveau est illustrée sur la figure 1.1. Les principales phases en sont les suivantes :

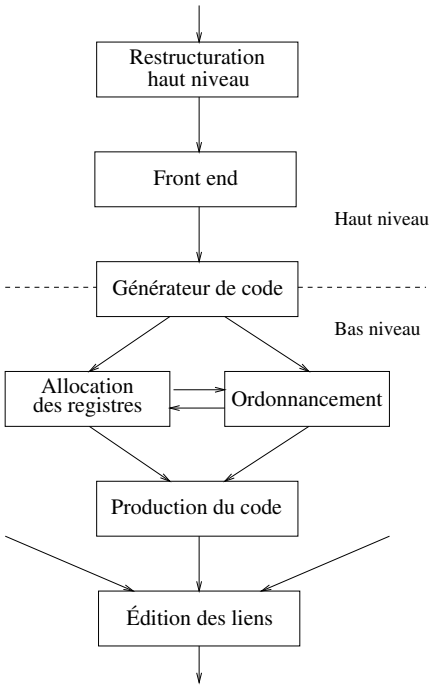


FIG. 1.1 – *Enchaînement des phases d'un compilateur*

Restructurations de haut niveau : elles permettent d'améliorer le placement des données (*padding* par exemple) et de restructurer le programme par dépliage, *inlining*, parallélisation ou vectorisation, etc. Les élémentaires telles que la propagation des constantes ou le calcul des invariants de boucle ont aussi lieu à ce moment.

Front end : il transforme le langage de haut niveau en une représentation intermédiaire, indépendante de ce langage. Les structures de données et de contrôle sont visibles.

Génération du code : elle transforme la représentation intermédiaire en langage assembleur. La phases de sélection des instructions choisit les mnémoniques utilisés

pour réaliser les actions du programme avec le jeu d'instructions disponible. C'est une partie critique car ce choix conditionne toutes les transformations ultérieures.

Optimisation de bas niveau : elles incluent le dépliage de boucle, le pipeline logiciel, construction de superblocs, insertion de gardes, etc. Ces optimisations ont connaissance de l'utilisation des ressources matérielles par le programme et elles tentent d'exploiter les unités fonctionnelles au maximum.

Ordonnancement : la génération de code produit des instructions dans un ordre qui vérifie simplement la sémantique du programme. L'ordonnancement consiste à modifier l'ordre des calculs effectués et à placer les instructions dans un ordre qui exploite mieux le matériel disponible. Ainsi les instructions indépendantes peuvent être permutées ou exécutées simultanément.

Allocation des registres : jusqu'à ce point, les noms des registres utilisés sont virtuels, ils désignent simplement des valeurs calculées par le programme. Seuls quelques registres réservés par convention par l'architecture, le système ou le compilateur (adresse de retour de fonction, paramètres d'une fonction) ont leur nom définitif. Cette phase détermine les durées de vie de tous les registres utilisés, les compare pour détecter les interférences et affecte chacun à un registre physique. Si le nombre de valeurs vivantes est supérieur au nombre de registres disponibles, le compilateur insère du *spill code* : il stocke une valeur en mémoire pour libérer un registre, puis la relit plus tard. Les performances sont évidemment dégradées.

Production du code : le code assembleur est une représentation fidèle de ce que doit être le code final. L'assemblage consiste à traduire les mnémoniques en codes machine pour obtenir un code «objet».

Édition des liens : elle réunit les différents codes objets qui proviennent de fichiers distincts de façon à connecter les appels de procédures entre modules différents et les appels aux procédures contenues dans des bibliothèques. Le code final est produit au terme de cette transformation.

Ces phases sont dépendantes les unes des autres, aucune ne remet en cause le travail de la précédente. Si l'ordonnancement a lieu avant l'allocation des registres, il impose un ordre aux calculs qui tend à augmenter la pression sur les registres. Au contraire, si l'allocation des registres a lieu avant, elle impose des contraintes supplémentaires à l'ordonnanceur en termes de dépendances de données[2].

Au cours de la thèse nous avons développé des outils capables de travailler sur du code assembleur, donc après la phase de génération de code : optimisation de bas niveau, ordonnancement et allocation des registres. Ceci a été réalisé dans le contexte d'un projet Esprit LTR [1, 14]. Le développement des restructurations de haut niveau et la génération de code n'étaient pas de notre responsabilité et n'avons pas pu modifier leur fonctionnement. Par contre nous nous sommes attachés à établir une communication entre toutes ces phases, de façon à pouvoir contrôler leurs actions et avoir une rétroaction.

2. Le problème des dépendances de données en relation avec la compilation et l'optimisation est abordé dans [11, 74].

1.2 Architecture des processeurs et performances

L'augmentation rapide des fréquences des microprocesseurs a nécessité l'invention de mécanismes capables d'alimenter les unités fonctionnelles en opérations et en données à un rythme suffisant. Ces mécanismes, qui permettent des fréquences d'exécution élevées, peuvent se comporter en véritables goulots d'étranglement quand ils n'ont pas un fonctionnement idéal et limitent alors sévèrement les performances d'un programme [105]. Nous illustrons ces phénomènes à l'aide du processeur TM1000 de Philips.

1.2.1 Implications du matériel

Les caractéristiques architecturales des processeurs ont un impact sur les performances des applications qui les utilisent. Nous présentons ici l'influence de la hiérarchie mémoire, de la politique de séquencement des instructions, des branchements et des gardes.

1.2.1.1 Hiérarchie mémoire

La différence croissante entre le temps de cycles d'un processeur et le temps d'accès à la mémoire (plusieurs dizaines de cycles) a fait naître le concept de hiérarchie mémoire. La mémoire est organisée en niveaux, le plus petit et le plus rapide étant le plus proche du processeur. Le but est de placer les données qui ont le plus de chances d'être réutilisées dans les niveaux les plus proches du CPU, de façon à exploiter le plus efficacement possible la «localité» des accès à la mémoire. On distingue deux types de localité :

la localité temporelle : quand une donnée est utilisée, elle a de grandes chances d'être réutilisée dans le futur ;

la localité spatiale : quand une donnée est utilisée, ses voisines ont une forte probabilité d'être utilisées prochainement.

Idéalement toutes les données utilisées par un programme résideraient dans le cache de premier niveau. Le volume des données traitées rend ceci généralement impossible. Le problème est alors de placer dans le cache les données les plus utiles. Le comportement des mémoires caches est déterminant pour les performances d'un programme. Notons m le taux de défaut de cache, N le nombre d'accès à la mémoire et p la pénalité encourue à chaque défaut. Supposons qu'une donnée lue dans le cache se fait en un cycle. Le temps idéal est $t_{\text{idéal}} = N \times 1$, le temps réel est $t_{\text{réel}} + t_{\text{penalité}}$ avec $t_{\text{penalité}} = N \times m \times p$. Le ralentissement dû au mauvais fonctionnement du cache est :

$$r = \frac{t_{\text{réel}}}{t_{\text{idéal}}} = \frac{N \times 1 + N \times m \times p}{N \times 1} = 1 + mp$$

Si la pénalité est de 20 cycles, un taux de succès du cache de 95 % implique une dégradation des performances d'un facteur 2.

1.2.1.2 Exécution des instructions

Les dépendances de données imposent un ordre entre les instructions qui composent un programme. Cet ordre garantit la correction du programme. Lorsqu'il s'agit d'exécuter ce programme sur un processeur, la limitation du nombre de ressources introduit de nouvelles contraintes.

L'ordonnancement consiste à ordonner totalement les instructions en respectant l'ordre partiel imposé par le graphe de dépendances et en évitant les conflits de ressources. L'ordonnancement produit est différent selon que l'architecture est RISC ou VLIW :

RISC : Un processeur superscalaire RISC dispose de toute la logique nécessaire pour détecter les conflits de ressources. Une instruction qui nécessite une unité fonctionnelle en cours d'utilisation est simplement retardée. Les délais induits par les latences des instructions sont aussi gérés par le matériel. Certains processeurs récents (MIPS R10000, DEC 21264, Pentium Pro, ...) sont capables d'exécuter les instructions en désordre pour limiter le nombre de bulles dans le pipeline. L'ordonnancement permet alors d'améliorer la répartition des instructions sur les unités fonctionnelles.

VLIW : La logique d'un processeur VLIW, au contraire, est beaucoup plus simple. Le compilateur est responsable du calcul des délais entre instructions, de l'affectation des instructions à un *slot* et de l'absence de conflits de ressources. En cas de besoin, il insère des instructions vides (*nop*) :

slot 1	*slot* 2	*slot* 3	*slot* 4	*slot* 5
asri (1) r55 → r50	ijmpi (MXM.DT9)	bitand r55 r47 → r48	nop	nop
ugtr r48 r53 → r47	nop	nop	nop	nop
iadd r50 r47 → r44	nop	nop	nop	nop
nop	nop	nop	nop	nop

L'ordonnancement influe directement sur le nombre de cycles nécessaires à l'exécution d'un programme. Si cet aspect est relativement moins important pour les processeurs RISC capables d'exécuter les instructions dans le désordre, il est critique pour les processeurs VLIW.

1.2.1.3 Branchements

Un programme contient nécessairement des boucles et des conditionnelles. Ces structures de contrôle sont traduites en langage machine par des instructions de saut. Les ruptures dans le flot de contrôle posent plusieurs problèmes en termes de performance :

- quand un branchement est pris, le destination risque de ne pas profiter de la localité spatiale des accès au cache instructions et de provoquer un défaut de cache ou de TLB ;
- la majorité des processeurs actuels étant pipelinés, un branchement introduit l'insertion de «bulles». En effet tant que la destination du saut n'est pas connue, c'est-à-dire pas avant un des derniers étages du pipeline, le processeur ne peut lancer de nouvelle instruction.

Pour remédier à ce dernier point, les processeur prédisent la destination du branchement et exécutent spéculativement des instructions, quitte à les annuler en cas de mauvaise prédiction. Le prédicteur doit déterminer la direction et, le cas échéant, l'adresse du branchement. De nombreux type de prédicteurs ont été proposés : prédicteur à un bit, à deux bits [106], à deux niveaux d'historique [120], combinés [67, 83], etc.

Le taux de bonnes prédictions atteint 90 à 95 % sur les codes «entier» et près de 99 % sur les codes «flottant».

Le branchement retardé est un mécanisme matériel qui retarde l'exécution d'un branchement d'un ou plusieurs cycles. La (ou les) instruction(s) qui sui(ven)t le branchement sont exécutées. Ceci permet d'éviter quelques bulles dans le pipeline en attribuant au compilateur la charge de trouver des instructions utiles. Ce mécanisme est notamment mis en œuvre dans le processeur Sparc [38].

1.2.1.4 Gardes

L'insertion de gardes permet d'éviter l'ajout de branchements pour traiter les conditionnelles et d'exploiter davantage de parallélisme d'instructions. En revanche, dans certains cas, l'allocation de registres est plus complexe : en effet la durée de vie des registres de destination d'une instruction gardée ne peut plus toujours être calculée statiquement, car elle dépend d'une valeur booléenne calculée pendant l'exécution. En l'absence de certitude, l'allocateur doit supposer que de nombreux registres interfèrent, ce qui accroît la pression. En définitive, l'allocation de registres échoue plus souvent, ou insère plus de *spill code*.

Nous utilisons les gardes dans les expériences présentées au chapitre 3.

1.2.2 L'exemple du processeur TM1000

Le processeur TM1000 est le premier de la gamme TriMedia de Philips [30, 52] (cf. figure 1.2). Ces processeurs sont destinés à servir des applications multimédia et intègrent des mécanismes d'acquisition vidéo et audio, un bus PCI et un support pour les algorithmes de décompression.

Présentation générale Le TM1000 [30] est un processeur VLIW cadencé à 100 MHz. Il est capable d'exécuter cinq opérations par cycle et dispose d'un cache instructions et d'un cache données séparés. Le fichier de registres contient 128 registres de 32 bits à usage général, il est divisé en deux parties : les registres r2 à r63, dit globaux, sont sauvegardés pendant le traitement d'une interruption, les registres locaux r64 à r127 ne le sont pas. Le jeu d'instructions inclut des instructions pour le multimédia (r0 et r1 sont constants et respectivement égaux à 0 et 1).

Spécificités VLIW Une architecture VLIW se distingue essentiellement des RISC par l'absence de mécanisme matériel pour contrôler l'affectation des opérations aux unités fonctionnelles et le respect des dépendances de données. L'ordonnancement des instructions est entièrement à la charge du compilateur.

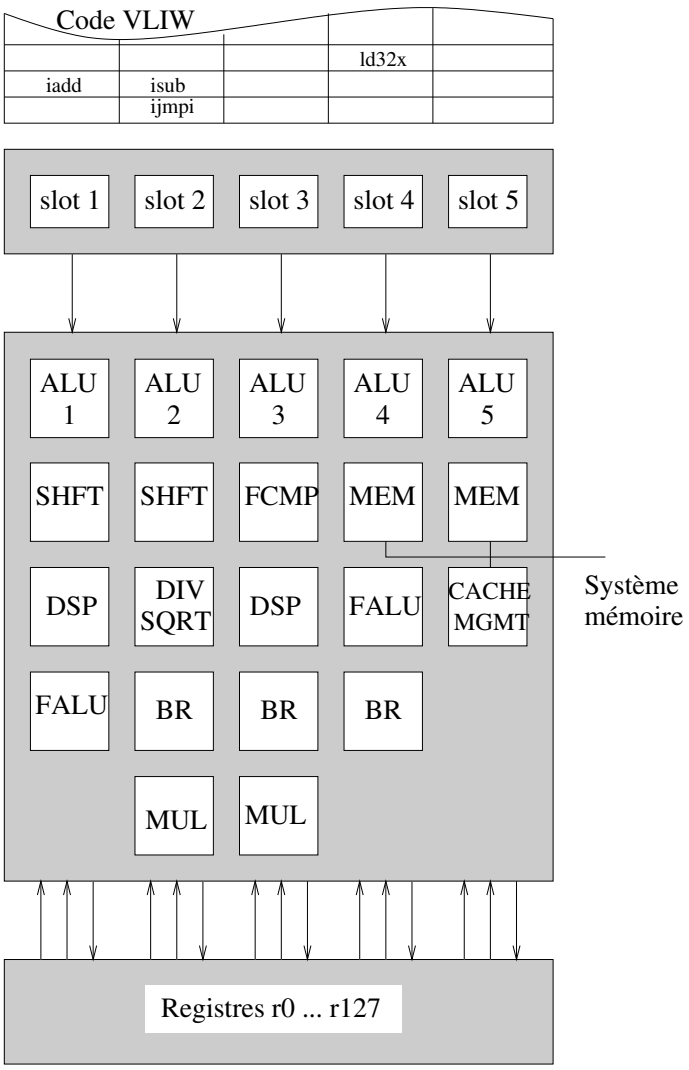


FIG. 1.2 – *Le processeur TM1000*

Tous les groupes de cinq opérations ne forment pas une instruction valide. Chaque opération ne peut être placée que dans certains *slots*. Par exemple une addition entière peut être attribuée à n'importe quel *slot*, par contre une division flottante n'est possible que dans le *slot* n° 2.

Mémoires cache Deux mémoires cache distinctes pour les instructions et les données permettent de masquer la latence de la mémoire. Les deux sont associatifs à huit voies. Le cache données a une capacité de 16 Ko, le cache instructions dispose de 32 Ko et contient des informations comprimées. La décompression est assurée par du matériel entre le cache et le CPU.

Branchements Sur le TM1000 les branchements sont retardés de trois cycles. Cela signifie que les $3 \times 5 = 15$ opérations qui suivent un branchement sont toujours exécutées. En outre, le mécanisme de compression du code impose que tout bloc de base commençant par un label soit atteint par un branchement et non pas par une exécution en séquence (*fallthrough*), ce qui augmente encore le nombre de branchements.

Si le compilateur ne réussit pas à placer dans ce délai des instructions utiles, le parallélisme d'instructions sera dégradé d'autant. Nous présentons plus loin dans ce chapitre des techniques qui visent à augmenter la taille des blocs de base de façon à diminuer la fréquence des branchements.

Gardes Toutes les opérations peuvent être gardées (c'est-à-dire voir leur exécution conditionnée) par n'importe quel registre. Une opération gardée s'exécute normalement mais son effet (écriture en mémoire ou dans un registre) n'est validé que si la condition de la garde est satisfaite. Chacune des cinq opérations d'une instruction peut être gardée par un registre différent.

1.3 Optimisations dépendantes de l'architecture

Les optimisations dépendantes de l'architecture sont celles qui ont pour but d'adapter l'ordre des calculs et le rangement des données à la structure de l'architecture. Schématiquement, les transformations classiques s'appliquent au code ou aux données. Nous nous intéressons surtout à celles qui modifient le code. Pour une étude détaillée des transformations de programme, le lecteur peut se référer au rapport de Bacon, Graham et Sharp [7]. Ebcioğlu *et al.* présentent des techniques de compilation adaptées au processeurs VLIW dans [43].

Nous présentons trois optimisations de haut niveau qui nous sont utiles pour les prochains chapitres dans la première partie de cette section, puis plusieurs optimisations de bas niveau. Nous citons ensuite des travaux relatifs aux séquences de transformations et nous abordons l'interaction entre logiciel et matériel au cours du processus d'optimisation.

1.3.1 Optimisations de haut niveau

La loi d'Amdahl [67] nous apprend que les zones les plus intéressantes à optimiser sont celles qui consomment le plus de temps. Les boucles ont donc naturellement fait l'objet de nombreuses études et un grand nombre de transformations ont été proposées : certaines modifient l'ordre des itérations (échange, blocage, *skewing*), d'autres affectent la structure du code en respectant l'ordre des itérations (dépliage, pipeline logiciel), d'autres encore modifient les accès à la mémoire (*padding*, *scalar expansion*). Elle visent à améliorer le parallélisme d'instructions ou l'utilisation des registres, diminuer la taille du code, augmenter la localité des accès, etc. De nombreux aspects des transformations de boucles sont décrits dans [7]. Nous décrivons ici deux optimisations de boucles qui nous sont utiles dans ce document : le dépliage et le blocage. L'*inlining* nous est utile pour supprimer les appels de fonctions qui interdisent certaines transformations et pour augmenter le parallélisme potentiel. Il est présenté à la fin de cette partie.

1.3.1.1 Dépliage de boucles

Déplier une boucle consiste à répliquer son corps un certain nombre de fois, appelé le facteur de dépliage u, et à parcourir le nouvel espace d'itération par pas de u. Ainsi une seule itération de la nouvelle boucle réalise le travail effectué par u itérations de la boucle originale. Le programme ci-dessous donne un exemple d'une boucle simple dépliée trois fois. Un épilogue est nécessaire pour le cas où le nombre d'itérations de la boucle n'est pas multiple de u.

```
for(i=0; i < n; i++) {
   a[i] = a[i] + b[i];
}
```

```
for(i=0; i < n-3; i+=3) {
   a[i] = a[i] + b[i];
   a[i+1] = a[i+1] + b[i+1];
   a[i+2] = a[i+2] + b[i+2];
}
for( ; i < n; i++) {
   a[i] = a[i] + b[i];
}
```

Le dépliage présente de nombreux avantages en termes de parallélisme d'instructions :

- le coût de gestion de la boucle (tests, branchements) est réduit d'un facteur u, puisque le coût fixe correspond à u itérations de la boucle originale ;
- les blocs sont plus longs, ce qui permet à l'ordonnanceur d'exploiter plus de parallélisme. Nous prenons ici les caractéristiques du TM1000 : nous supposons que la latence d'une lecture en mémoire est de trois cycles et que celle d'une addition est d'un cycle. Au maximum deux accès à la mémoire sont possibles à chaque cycle. L'ordonnancement optimal du corps de cette boucle nécessite cinq

cycles :

cycle 1	lire a[i], lire b[i]
2	nop
3	nop
4	addition
5	écrire a[i]

La boucle dépliée trois fois nécessite sept cycles :

cycle 1	lire a[i], lire b[i]
2	lire a[i+1], lire b[i+1]
3	lire a[i+2], lire b[i+2]
4	addition (i)
5	écrire a[i], addition (i+1)
6	écrire a[i+1], addition (i+2)
7	écrire a[i+2]

Le nombre de cycles nécessaires à l'exécution d'une itération est passé de 5 à $7/3 \simeq 2,3$ soit une accélération d'un facteur supérieur à 2.

– Pour certaines boucles, il permet de réduire le nombre d'accès à la mémoire en réutilisant les données.

En contrepartie le dépliage accroît la taille du code et risque de dégrader le comportement du cache instructions.

Le dépliage peut être appliqué à toutes les boucles, tant à haut niveau qu'à bas niveau. Nous revenons sur le dépliage de boucle à bas niveau et sur les conséquences de l'épilogue au chapitre 3. Le problème de la détermination d'un facteur de dépliage optimal est abordé au chapitre 4.

1.3.1.2 Blocage

Les boucles imbriquées sont typiquement utilisées pour parcourir des tableaux à plusieurs dimensions. Dès lors que la taille des tableaux utilisées dépasse la taille du cache données, la localité des accès va avoir un impact significatif sur les performances. Considérons le produit matriciel (partie gauche de la figure 1.3) : les éléments de la matrice B sont parcourus l'un après l'autre, en colonnes. L'ensemble de la matrice est parcouru n fois. Cette implémentation n'exploite ni localité spatiale car le langage C range les matrices en lignes, ni localité temporelle car l'ensemble de la matrice est parcouru avant qu'un élément ne soit réutilisé (cf. figure 1.4). La matrice A ne profite de localité temporelle que si une ligne entière peut résider dans le cache.

Le principe du blocage est de diviser l'espace d'itérations en pavés de façon à maximiser la réutilisation des données au sein de chaque bloc, au prix de perte de localité entre les pavés. La partie droite de la figure 1.4 illustre la transformation de l'espace d'itérations, la code correspondant est en partie droite de la figure 1.3. Cette optimisation dépend de trois paramètres qui sont les dimensions des pavés. Une approche analytique pour les déterminer est présentée dans [22]. Dans [37] Coleman et McKinley proposent un algorithme qui calcule une taille de pavé qui minimise les interférences et

exploite la localité en fonction de la taille de l'espace d'itérations et de l'organisation du cache.

Notons que le blocage change l'ordre des accès aux éléments des tableaux, et n'est pas toujours une transformation légale.

```
for(i=0; i < n; i++) {
  for(j=0; j < n; j++) {
    for(k=0; k < n; k++) {
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
  }
}
```

Code original

```
for(ii = 0, ii < n; ii+=b1) {
  for(jj = 0; jj < n; jj+=b2) {
    for(kk = 0; kk < n; kk+=b3) {
      for(i=ii; i < min(ii+b1-1, n); i++) {
        for(j=jj; j < min(jj+b2-1, n); j++) {
          for(k=kk; k < min(kk+b3-1, n); k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
          }
        }
      }
    }
  }
}
```

Code transformé

FIG. 1.3 – *Blocage d'un produit matriciel*

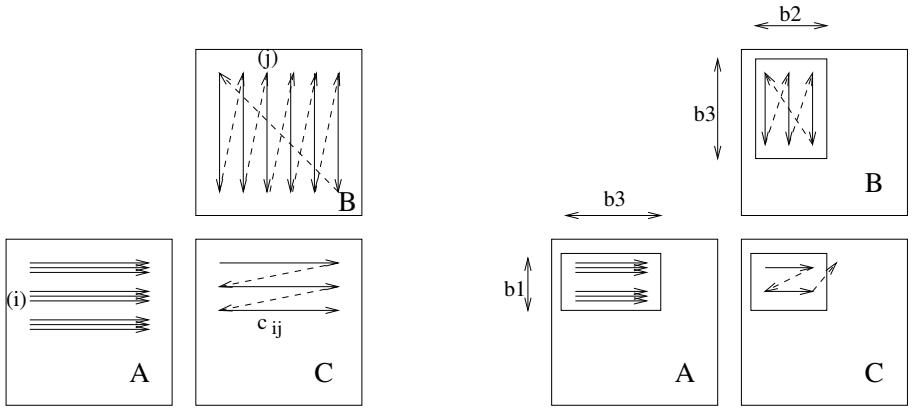


FIG. 1.4 – *Espaces d'itérations avant et après blocage*

Le blocage et le dépliage peuvent être associés pour améliorer les performances d'un programme. Nous abordons au chapitre 4 le problème de la détermination de la taille de bloc et du facteur de dépliage.

1.3.1.3 Inlining

L'*inlining*, c'est le remplacement d'un appel de fonction par une copie de son corps [62]. Les avantages sont nombreux : la propagation de constantes est plus efficace,

le passage de paramètres est inutile, un certain nombre de sauvegardes de registres n'a plus lieu d'être et la nouvelle région est plus grande et possède probablement plus de parallélisme d'instructions. La forte expansion du code constitue l'envers de la médaille : si elle n'est pas contrôlée la dégradation des performances de la hiérarchie mémoire risque de contrebalancer les gains espérés.

Dans [39], Davidson et Holler décomposent le coût d'un appel de fonction en éléments simples (comme le temps d'exécution de l'instruction de saut, le temps de sauvegarde des registres, ...) et obtiennent une expression détaillée qu'ils ont vérifié sur plusieurs architectures RISC et CISC. Contrairement à beaucoup d'articles sur ce sujet, leurs résultats montrent que l'augmentation de la taille du code n'affecte pas les performances du programme : si le taux de défauts de cache est supérieur, le nombre total d'instructions diminue, ce qui résulte en temps d'exécution plus faible.

Hwu et Chang [68] utilisent un *profile* pour connaître le graphe d'appel pondéré d'une application. Ils utilisent une fonction de coût qui estime l'augmentation de la taille du code à partir d'une représentation intermédiaire. Le gain, supposé constant pour toutes les fonctions, n'est pas pris en compte.

McFarling présente un modèle plus précis pour déterminer quelles sont les procédures à *inliner* [82]. Il remarque que si une procédure est typiquement appelée à partir d'un seul endroit du programme, elle doit être incluse. De même si la plupart des procédures appelées sont petites. Il constate que la majorité des procédures ne rentre dans aucun de ces deux cas extrêmes. Le modèle utilise un *profile* pour la fréquence d'utilisation des blocs de base, et la structure des boucles. Un ratio est calculé pour chaque appel de fonction dans une boucle : $r = \frac{\text{nombre d'appels}}{\text{taille}}$. McFarling favorise les fonctions fréquemment appelées ou de petite taille. L'algorithme évalue ce compromis entre coût et le gain et décide s'il faut *inliner*. Les résultats sont meilleurs que tous les algorithmes qui n'utilisent que la taille de la procédure.

L'*inlining* est une approche intéressante dans le cas où les branchements sont coûteux (trois cycles de cinq opérations sur le TM1000), mais il est essentiel de contrôler la taille globale du code : le problème est de déterminer quelles procédures doivent bénéficier le plus de la place disponible. Nous abordons cette approche dans le chapitre 3.

1.3.2 Optimisations de bas niveau

Les optimisations de bas niveau ont une connaissance précise de l'utilisation des unités fonctionnelles du processeur par le programme. C'est à ce niveau que le parallélisme d'instructions est réellement exploité.

1.3.2.1 Ordonnancement des instructions

L'ordonnancement des instructions par liste est la technique de placement des instructions la plus simple. Rau [99] et Gasperoni [49] se sont intéressés à l'ordonnancement pour processeur VLIW. Fisher a inventé le *trace scheduling* [46]. Bernstein et Rodeh [15] optimisent aussi un programme au delà des blocs de base : ils construisent le graphe de dépendance du programme, modélisent la machine cible et déplacent des instructions

utiles ou spéculatives. Des heuristiques utilisant les délais entre instructions et le chemin critique sont utilisées pour guider l'algorithme. Kerns et Eggers [72] ont introduit une variante de l'ordonnancement par liste en modifiant le calcul de la priorité associée aux opérations de chargement. La notion de *load level parallelism* est introduite. Elle correspond au nombre d'instructions qui peuvent être séquencées en parallèle avec le *load*. C'est cette valeur qui est utilisée par l'algorithme pour déterminer la prochaine instruction à séquencer. Ainsi le parallélisme potentiel d'un bloc de base est uniformément réparti entre les différentes instructions de chargement, même si la latence est plus longue en cas de défaut de cache (pour les caches non-bloquants).

En analysant des traces d'exécution de programmes, Cohn et Lowney [36] se sont aperçu que des programmes apparemment de bonne qualité semblent ne pas avoir été optimisés. Le chemin le plus fréquent dans une trace contient un grand nombre d'instructions utiles uniquement dans le cas défavorable. À partir de cette constatation ils copient des routines et les optimisent pour le meilleur cas en supprimant les instructions qui ont peu de chances d'être exécutées. Ainsi la longueur du chemin le plus fréquent est réduite. Du code de compensation est ajouté pour retourner à la routine originale si besoin est. La copie peut être fortement optimisée puisqu'elle peut généralement éviter de sauvegarder des registres, et contient du code mort (utilisé par la version originale). Dans la mesure où cette technique, baptisée *Hot-Cold Optimization*, repose sur des copies, il est important de trouver un bon compromis du nombre de routines à optimiser pour éviter de dégrader le comportement du cache instructions. Les résultats montrent une réduction du nombre d'instructions exécutées entre 3 et 8 %.

Récemment des travaux se sont intéressés à des techniques d'ordonnancement qui limitent la puissance consommée par le processeur [112].

1.3.2.2 Pipeline logiciel

Le pipeline logiciel [76] est une des techniques les plus sophistiquées pour exploiter le parallélisme d'instructions. De nombreux aspects en sont décrits dans [4]. Le principe est d'exploiter le parallélisme qui existe entre des instructions qui proviennent d'itérations différentes de la boucle.

Reprenons l'exemple que nous avons utilisé pour le dépliage de boucle :

```
for(i=0; i < n; i++) {
   a[i] = a[i] + b[i];
}
```

Nous avons vu que cette boucle nécessite cinq cycles par itération.

L'approche intuitive de construction du pipeline logiciel consiste à répliquer le corps des itérations successives de la boucle tous les II cycles jusqu'à apparition d'un motif régulier. La répétition de ce motif est alors remplacée par un branchement autour de ce nouveau corps de boucle. Lancer une itération par cycle impliquerait trois accès à la mémoire simultané, ce qui est impossible sur le TM1000. Nous commençons donc à

$II = 2$ et obtenons l'ordonnancement suivant. Le motif est indiqué en caractères gras.

	1	2	3	4
cycle 1	lire a[i]			
2	lire b[i]			
3		lire a[i+1]		
4		lire b[i+1]		
5	**add(i)**		**lire a[i+2]**	
6	**écrire a[i]**		**lire b[i+2]**	
7		add(i+1)		lire a[i+3]
8		écrire a[i+1]		lire b[i+3]
9			add(i+2)	
10			écrire a[i+2]	
11				add(i+3)

Cette technique permet de lancer une itération tous les deux cycles. Pourtant les contraintes matérielles nous autorisent à lancer jusqu'à deux itérations tous les trois cycles, ce qui sature les unités de chargement/déchargement.

Considérons le corps de la boucle dépliée deux fois et ordonnancé comme indiqué dans la première colonne du tableau ci-dessous. Si nous répétons ce code tous les trois cycles (cf. colonnes 2 à 4), nous observons un motif répétitif (en gras) qui nous permet de reconstruire une boucle. Nous avons obtenu une boucle dont la vitesse asymptotique est de 1,5 cycle par itération, c'est-à-dire le maximum possible selon les contraintes matérielles.

	1	2	3	4
cycle 1	lire a[i],a[i+1]			
2	lire b[i],b[i+1]			
3				
4		**lire a[i+2],a[i+3]**		
5	**add(i),add(i+1)**	**lire b[i+2],b[i+3]**		
6	**écrire a[i],a[i+1]**			
7			lire a[i+4],a[i+5]	
8		add(i+2),add(i+3)	lire b[i+4],b[i+5]	
9		écrire a[i+2],a[i+2]		
10				lire a[i+6],a[i+7]
11			add(i+4),add(i+5)	lire b[i+6],b[i+7]
12			écrire a[i+4],a[i+5]	

Quand il peut s'appliquer, le pipeline logiciel parvient généralement à réaliser un compactage efficace d'un corps de boucle, pour une taille de code inférieure à celle obtenue par le dépliage.

Les techniques de pipeline logiciel ont fait l'objet de nombreuses publications, par exemple [12, 20, 55, 76, 86, 96, 97, 104, 117]. Nous revenons sur le pipeline logiciel dans les chapitres 2 et 3.

1.3.2.3 Superbloc et hyperbloc

La majorité des optimisations appliquées au sein d'un bloc de base s'effectuent sans difficulté. La raison en est que le flot de contrôle est rectiligne, le déplacement d'une instruction n'aura pas d'effet de bord en dehors du bloc. Malheureusement les blocs de base sont généralement de petite taille, typiquement cinq instructions pour des applications classiques (i.e. qui ne sont pas des codes numériques), et le parallélisme potentiel est faible.

Le déplacement d'instructions entre blocs de base est nécessaire à l'obtention de performances élevées. Deux cas se présentent : une instruction peut franchir un point d'entrée ou un point de sortie du flot de contrôle.

- Le deuxième cas est relativement simple à traiter : si l'instruction est déplacée après une instruction de saut et qu'elle calcule un résultat utile dans l'autre branche, il suffit de placer une copie de l'instruction à la destination du branchement. Si l'instruction est déplacée avant le point de sortie, elle risque d'être exécutée par erreur. Il faut s'assurer qu'elle n'écrase pas une valeur utilisée dans l'autre branche du programme et qu'elle ne provoque pas d'exception (division par zéro, chargement à une adresse invalide, etc.). Certains processeurs (Sparc v9 [118], IA-64 [59]) disposent de version spéciales de ces instructions qui ne lèvent pas d'exceptions et assurent donc ce dernier point par matériel [31].
- Le déplacement d'une instruction au delà d'un point d'entrée est plus complexe. La figure 1.5 illustre le problème. Si l'instruction est déplacée vers le bas (schéma de gauche), le point d'entrée doit être modifié et les instructions intermédiaires doivent être copiées dans le bloc prédécesseur. Si l'instruction est déplacée vers le haut, elle doit aussi être copiée.

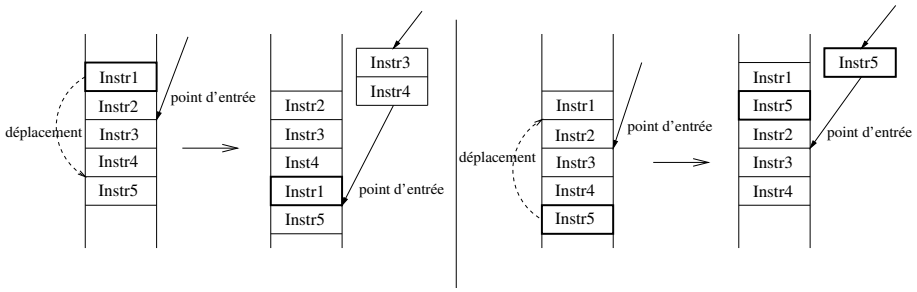


FIG. 1.5 – *Déplacement d'une instruction au delà d'un point d'entrée*

La notion de *superbloc* étend celle de bloc de base en relâchant une hypothèse : le superbloc est une section de code dont l'exécution ne peut commencer qu'à la première instruction. Contrairement au bloc de base, elle peut se terminer avant la dernière instruction. Cette unique propriété du superbloc simplifie le déplacement spéculatif d'instructions.

La construction d'un superbloc comprend deux étapes [69] : la première identifie des suites de blocs fréquemment exécutés en séquence, la seconde duplique les blocs situés

après un point d'entrée et redirige tous les branchements vers ce point sur la copie (voir illustration figure 1.6). L'identification des séquences de blocs fait généralement appel à des traces d'exécution. Pour s'affranchir de ce coût élevé, Hank *et al.* ont inventé une

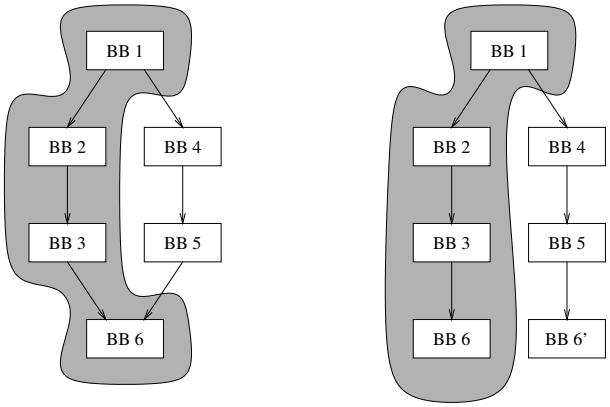


FIG. 1.6 – *Construction d'un superbloc*

méthode statique fondée sur plusieurs heuristiques triées par ordre de priorité [64]. L'article montre que les chemins fréquemment exécutés sont ceux qui contiennent le moins d'événements imprévisibles au moment de la compilation : entrées-sorties, synchronisations, sauts indirects, etc. Les superblocs sont formés de façon à éviter d'introduire des blocs qui contiennent ces aléas (qui empêcheraient toute optimisation). Les résultats montrent que les chemins choisis sont fréquemment exécutés, et que la précision de la prédiction des branchements n'est pas forcément un bon indicateur de la qualité des optimisations. Les résultats par rapport aux superblocs construits avec des informations de *profile* dépendent du degré de parallélisme de la machine cible.

À l'inverse, les auteurs de [33] obtiennent des résultats intéressants en utilisant l'information dynamique : les probabilités que le flot de contrôle quitte le superbloc guident l'ordonnancement de façon à ce que les branchements de sortie du superbloc les plus fréquentes soient placées le plus tôt possible.

Un superbloc correspond à un chemin d'exécution dans une application. À l'université de l'Illinois, Mahlke *et al.* ont généralisé la notion et proposent l'hyperbloc : il s'agit d'un ensemble d'instructions qui reprend les propriétés du superbloc, mais permet d'avoir des instructions gardées. Un hyperbloc peut donc correspondre à plusieurs chemins. La construction et un ensemble d'optimisations spécifiques sont développés dans [80].

L'allongement des blocs est intéressant pour augmenter le parallélisme d'instructions, particulièrement dans le contexte de processeurs VLIW. Nous décrivons au chapitre 3 comment cette technique, associée à l'insertion de gardes, nous permet d'améliorer les performances.

1.3.2.4 Exploitation de la prédiction de branchement

La prédiction de branchement est un mécanisme matériel qui permet au processeur de maintenir un flot d'exécution soutenu malgré les ruptures de séquence qu'occasionnent les branchements. Le processeur prédit l'adresse de la cible et continue à exécuter spéculativement des instructions. La connaissance du fonctionnement de ce mécanisme permet d'organiser le code de façon à en tirer le meilleur parti.

Pour améliorer l'exploitation de la corrélation des branchements pour la prédiction, Young et Smith [121] proposent de coder l'information d'historique dans le compteur de programme. En effet le prédicteur ne fonde sa décision que sur un schéma pris/pas-pris, pas sur le chemin suivi. Un *profile* est utilisé pour recenser tous les chemins conduisant à un branchement. Si deux chemins différents qui ont le même schéma conduisent à un comportement différent du branchement, un bout de code va être dupliqué pour résoudre le conflit. Le nombre d'instructions exécutées n'est pas modifié (notamment il n'y a pas d'ajout de branchements). L'augmentation de la taille du code n'excède jamais 30 % et cette technique améliore la prédiction de presque 15 % par rapport à d'autres techniques fondées sur les *profiles*. Le gain en performance n'est pas donné.

Dans leur article [28] Calder et Grunwald présentent quelques techniques pour réduire les pénalités de branchement : après la compilation, les blocs de base sont placés de façon à améliorer les prédictions en fonction de l'architecture (branchement en arrière pris, en avant non-pris, etc.). Trois algorithmes sont utilisés: *Greedy* qui construit une chaîne des arcs les plus utilisés, *Cost* fait de même en incluant le coût des différentes pénalités et *Try15* qui essaie toutes les combinaisons pour les 15 blocs les plus fréquents. Un gain de performance de 16 % en moyenne est constaté sur un processeur DEC Alpha 21064.

Mueller et Whalley [87], constatant l'augmentation de la taille des caches, décident d'appliquer des optimisations qui ont pour conséquence d'augmenter la taille du code. Ils essaient de déterminer si la direction de certains branchements peut être connue à l'avance sur certains chemins. Si une condition se déduit d'une autre, il est intéressant de répliquer le code de manière à supprimer le branchement. L'optimisation se fait en trois phases : détermination des branchements qui peuvent être évités, restructuration du flot de contrôle en dupliquant certains blocs et positionnement des blocs nouvellement créés.

1.3.2.5 Placement de code

Le comportement de la hiérarchie mémoire a un impact non négligeable sur les performances d'un programme. Pour réduire le nombre de fautes de pages et/ou de défauts de cache, il faut essayer de minimiser le nombre de conflits. Différentes approches tentent de résoudre le problème en manipulant le code à différents niveaux de granularités : modules, procédures, blocs de base ou instructions. D'autres études franchissent volontairement ces frontières pour augmenter encore le parallélisme.

Modules Dans leur article [57], Gösmann *et al.* modifient l'ordre des modules qui composent un programme lors de l'édition des liens. L'algorithme utilise une fonction

de coût, une fonction de sélection et une heuristique de recherche. À partir d'un ordre initial aléatoire ils calculent son voisinage, obtenu en effectuant toutes les permutations possibles de deux modules. Un élément du voisinage est aléatoirement choisi, évalué et comparé à l'ordre actuel. S'il est meilleur il est sélectionné, sinon il est ignoré. La fonction de sélection permet d'éviter de rester bloqué dans un minimum local en rejetant aléatoirement des bons éléments. Les résultats avec cinq programmes SPEC'92 montrent une amélioration de 20 % des performances.

Procédures `cord` est un utilitaire livré avec le système Irix des stations de travail Silicon Graphics. Il réorganise les procédures dans un programme exécutable selon un ordre spécifié dans un fichier. Le but est de réduire le nombre de défauts de pages en groupant les fonctions appelante et appelée dans la même page. L'ordre des procédures est généralement calculé à l'aide d'un *profile* obtenu par `pixie` et `prof`. L'algorithme utilisé par `cord` et `pixie` n'est pas détaillé.

Pettis et Hansen [93] ordonnent les procédures de telle sorte qu'une procédure fréquemment appelée soit à proximité de la procédure appelante. Ainsi ils augmentent la probabilité que ces procédures résident dans la même page et minimisent les risques de conflit dans le cache instructions. `gprof` [56] est utilisé pour obtenir un graphe d'appel pondéré. L'algorithme considère alors répétitivement l'arc de poids le plus élevé et regroupe les deux nœuds en un seul. Les procédures correspondantes sont placées côte à côte. L'ordre ainsi déterminé est utilisé par l'éditeur de liens pour produire le programme final.

Hashemi, Kaeli et Calder modifient aussi l'ordre des procédures au moment de l'édition des liens [66]. Ils utilisent comme données le graphe d'appel ainsi que la taille du cache, des lignes et des procédures. L'algorithme de placement procède par coloriage des lignes de cache : une procédure qui pourrait entrer en conflit dans le cache avec une autre se voit attribuer un ensemble de couleurs interdites pour les lignes qu'elle occupe. Les arcs sont traités par ordre décroissant de poids. Lorsqu'une procédure est déplacée, le vide est comblé par des procédures qui ne sont exécutées que peu fréquemment. Selon les auteurs, l'algorithme améliore les performances de 17 % par rapport au placement de Pettis et Hansen.

Toutes les méthodes basées sur le graphe d'appel ont un défaut : elle ne prennent pas en compte les conflits en «frères» et «sœurs». Supposons que la fonction A soit exécutée 100 fois et qu'elle appelle les fonctions B et C 50 fois chacune. Un *profile* ne donne aucune indication sur l'interaction entre B et C. Si le déroulement du programme est de la forme $AB^{50}C^{50}$, les deux fonctions ne sont pas en conflit et peuvent occuper le même emplacement dans le cache. Par contre si le motif $A(BC)^{50}$ prévaut, il est essentiel de s'assurer que B et C n'entrent pas en conflit. Dans [53] les auteurs prennent cette considération en compte : ils extraient l'ordonnancement temporel des procédures d'une trace d'exécution et l'intègrent à la technique de Pettis et Hansen [93].

L'*inlining* est aussi utilisé pour diminuer le nombre de défauts de cache. Ce point est développé dans le paragraphe 1.3.1.3.

Au niveau du bloc Le placement de blocs de base au sein d'une procédure est une technique plus fine mais plus complexe à mettre en œuvre. Supposons en effet que le bloc A ait pour successeur les blocs B et C correspondant respectivement au branchement pris et non pris (cf. figure 1.7). Si C est déplacé, il est nécessaire d'ajouter un branchement inconditionnel vers C à la fin de A pour conserver un code correct. Sans ce code de compensation, le successeur de A deviendrait le bloc D. De même un branchement est nécessaire à la fin de B pour «éviter» C.

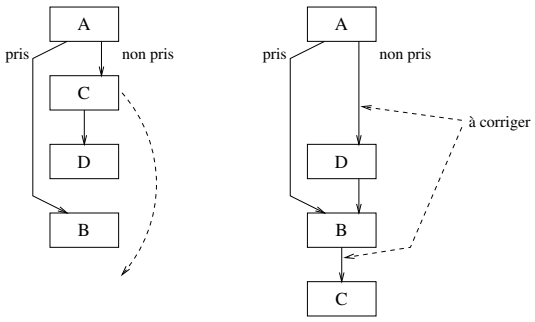


FIG. 1.7 – *Nécessité du code de compensation*

McFarling [81] utilise une représentation du programme qui contient le graphe d'appel, les boucles et les conditionnelles. Il suppose qu'il est possible de préciser quelles instructions ne doivent pas être stockées dans le cache. L'algorithme parcourt le graphe en profondeur d'abord en se concentrant sur les boucles. S'il reste de la place dans le cache quand la boucle la plus imbriquée est placée, il commence à placer des instructions de la boucle de niveau immédiatement inférieur, et ainsi de suite. McFarling obtient les performances d'un cache associatif avec un cache à correspondance directe.

Après avoir placé les procédures, Pettis et Hansen s'intéressent au placement des blocs dans chaque procédure et présentent deux algorithmes [93]. Le premier construit des chaînes de blocs de plus en plus longues : de tous les arcs, il sélectionne celui de poids le plus élevé, forme une chaîne avec les deux blocs et recommence tant que cela est possible. Si l'un des deux blocs est déjà au milieu d'une chaîne, l'arc est ignoré. Si l'un des blocs est en tête d'une chaîne et l'autre en queue, les deux chaînes sont regroupées. Les chaînes sont alors ordonnées de façon à minimiser le nombre de branchements pris. Le deuxième algorithme commence par le premier bloc de la procédure et place systématiquement le successeur le plus fréquent du dernier bloc placé. Si celui-ci est déjà placé, l'algorithme choisit le bloc relié par l'arc de poids maximum à un des blocs placés. Les deux algorithmes produisent un code dans lequel les blocs les plus fréquemment exécutés se trouvent au début de la procédure. Pour augmenter encore la localité il est possible de scinder les procédures en deux et de rejeter toutes les zones les moins fréquentes à la fin du code.

Dans [113] Torrelas, Xia et Daigle optimisent le système d'exploitation entier. À l'aide de *profiles* ils commencent par déterminer des séquences de blocs en partant de germes et en suivant les successeurs les plus probables tant que la probabilité de

branchement est supérieure à un seuil fixé. Le cache est divisé en deux zones, la plus petite étant réservée aux séquences extrêmement fréquentes (cf. figure 1.8), l'autre contient les séquences restantes placées par ordre de fréquences décroissantes, et le corps des boucles importantes. Le reste du cache est rempli par le code peu ou jamais exécuté. Réserver une zone du cache permet de faire bénéficier le code qui y est placé de localité temporelle en plus de la localité spatiale. Cet algorithme permet de réduire les défauts de cache de 31 à 86 %. Les auteurs envisagent des variantes de leur approche,

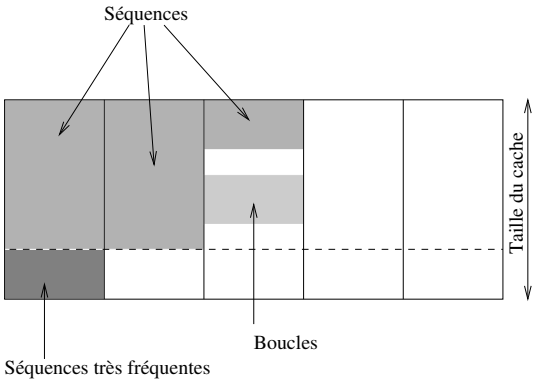


FIG. 1.8 – *Placement de code de Torrelas* et al.

mais aucune n'est satisfaisante : le partionnement du cache en deux zones réservées l'une au système et l'autre aux applications n'apporte pas de gain. L'ajout d'un petit cache, de la taille de la zone réservée, dédié aux séquences fréquentes n'apporte pas non plus de gain par rapport à un bon placement du code. L'alignement des boucles et de leurs descendants dans des caches logiques fait perdre la localité spatiale et n'est pas non plus une bonne solution.

1.3.2.6 Utilisation de gardes

Certains jeux d'instructions permettent de mettre en œuvre des gardes. Une instruction gardée voit son effet conditionné par une valeur booléenne. Certains processeurs fournissent quelques registres de garde spécifiques alors que d'autres permettent d'utiliser n'importe quel registre.

Les gardes permettent d'éliminer des branchements, qui nuisent toujours aux performances du pipeline. Sans garde, la ligne C :

if (x==3) y = y1+y2;

s'écrit comme sur la partie gauche de la figure 1.9. Avec garde, le branchement n'est plus nécessaire (partie droite de la figure).

Le revers de la médaille apparaît au moment de l'allocation des registres. En effet un registre est considéré comme «vivant» entre son écriture et sa dernière lecture. En présence de gardes, il est difficile de déterminer si un registre est effectivement lu ou écrit. En l'absence de certitude, on suppose — souvent inutilement — qu'un grand nombre de registres sont en conflit, et la pression de registres risque de devenir ingérable.

```
        cmp r11,3                  |
        bne suite                  |        iseq r11,3 → r10
        add r2,r3 → r4             |  IF r10 add r2,r3 → r4
suite:                             |
                                   |
        Sans garde                 |        Avec garde
```

FIG. 1.9 – *Utilisation des gardes pour éviter un branchement*

Les auteurs de [51] ont proposé une technique qui prend en compte les relations entre les prédicats et améliore la précision de l'analyse de la durée de vie des registres. L'allocation de registres globale bénéficie de cette précision et nécessite un nombre de registres en moyenne 20 % inférieur à la technique classique. Johnson et Schlansker [71] ont proposé une technique d'analyse des prédicats. Ils utilisent un graphe qui représente les relations entre les différentes valeurs que peuvent prendre les prédicats. Un système de requêtes permet à l'optimiseur d'obtenir les informations dont il a besoin sur les valeurs respectives des prédicats.

Quand les gardes sont utilisées pour conditionner l'exécution d'un bloc de plusieurs instructions, cette technique peut s'avérer inefficace. En effet les instructions gardées s'exécutent, seul leur résultat est inutilisé le cas échéant. Le coût de l'exécution d'un bloc entier est à comparer avec le coût d'une instruction de branchement. Ce phénomène est modélisé en détail dans le chapitre 3.

1.3.3 Séquences de transformations

L'optimisation d'un programme met en jeu un certain nombre de transformations de code, tant à haut niveau qu'à bas niveau. La question est de savoir dans quel ordre ces tâches doivent être exécutées de façon à obtenir la meilleure performance. Il semble difficile de déterminer une séquence de transformations de code optimale, indépendante de l'application à compiler et de l'architecture matérielle. L'intégration de plusieurs techniques a déjà fait l'objet d'expérimentations.

L'intégration de l'allocation de registres et de l'ordonnancement des instructions a été étudiée par de nombreuses équipes [16, 26, 90, 94].

Dans [54] Goodman et Hsu présentent deux solutions : dans la première le compilateur préfère générer le code en prenant en compte seulement les interblocages du pipeline mais mémorise le nombre de registres utilisés. Quand ce nombre devient élevé il change de stratégie et essaie de séquencer des instructions qui libèrent des registres. La deuxième approche utilise le graphe de dépendance et cherche un compromis entre hauteur et largeur, c'est-à-dire tente de minimiser le chemin critique tout en limitant le nombre de valeurs vivantes au nombre de registres disponibles.

Bradlee, Eggers et Henry, de l'université de Washington ont développé le système Marion [25] : un générateur de générateur de code pour processeurs RISC. Le système, indépendant de l'architecture, lit un fichier de description rédigé en langage Maril. La modularité de Marion permet de changer les politiques d'ordonnancement, d'allocation

des registres ainsi que leur stratégie de communication.

Carr s'intéresse aux nids de boucles et cherche un compromis entre le parallélisme d'instructions et la localité des données [29]. Pour cela il modélise les caractéristiques du processeur et du nid de boucles, et aboutit à un problème d'optimisation en nombre entiers. Les principaux paramètres du modèle sont :

- pour le processeur, la vitesse à laquelle il peut exécuter une instruction flottante et la vitesse à laquelle il peut lire une donnée en mémoire ;
- pour le nid de boucles, le nombre d'instructions de chargement et le nombre d'instructions flottantes.

Les effets de recouvrement des latences et des défauts de caches sont modélisés. L'algorithme est implémenté dans un transformateur Fortran-Fortran. Les transformations doublent les performances des boucles considérées.

Dans [17], les auteurs notent qu'un certain nombre d'optimisations sont appliquées sans savoir si leur effet sera bénéfique au moment de l'ordonnancement parce qu'elles manquent d'informations, notamment sur la disponibilité des registres ou des unités fonctionnelles, la qualité du parallélisme atteint et donc la possibilité d'insérer des instructions à moindre coût dans des *slots* vides. Pour valider leur approche, les auteurs ont intégré la construction de superblocs et l'ordonnancement. La construction de superblocs se poursuit tant que le parallélisme d'instructions est insuffisant. Quand il se trouve en excès, il peut être redistribué vers d'autres zones de code.

Le générateur de code Aviv [65] propose une approche originale pour intégrer la sélection des instructions, l'allocation des ressources et l'ordonnancement. Pour cela il utilise un graphe qui représente toutes les allocations possibles d'opérations à des unités fonctionnelles ainsi que les transferts entre bancs de registres qu'elles impliquent. Un parcours du graphe avec par un algorithme de *branch-and-bound* et quelques heuristiques permettent de choisir les meilleures allocations en maximisant le parallélisme potentiel et en minimisant les transferts entre bancs de registres. Une recherche de cliques maximales permet ensuite de construire les instructions VLIW. L'architecture cible est décrite grâce au langage ISDL [61] et Aviv utilise la technologie Suif et SPAM. Dans [98], Rau propose une approche de génération code similaire, à l'aide de graphes.

L'affectation des instructions aux unités fonctionnelles a lieu au moment de la sélection du code. Dans une architecture irrégulière de type VLIW, cette affectation ajoute des contraintes qui risquent de nuire aux phases ultérieures. Pour éviter cela Rau a proposé de décomposer la prise de décision pour pouvoir la retarder au maximum [98]. Le nom d'un registre virtuel est concrétisé par étapes : quel type, dans quel banc, enfin quel numéro. Parallèlement le code opératoire devient de plus en plus précis, du simple `add` au choix d'une unité fonctionnelle précise.

Ces approches statiques permettent d'améliorer les performances en intégrant plusieurs techniques. Pourtant elles ne résolvent pas le problème, qui est plus général : en cas de mauvaise décision de la part d'une transformation de code, il est impossible de revenir en arrière. Nous montrons au chapitre 4 que la modification de la structure d'un compilateur permet d'ajouter des mécanismes de communications entre différents modules d'optimisations et qu'un mauvais choix peut être remis en cause par un autre

module de façon à accroître les performances.

1.3.4 Interaction logiciel-matériel

Nous assistons aussi à une répartition des tâches entre le logiciel et le matériel. Des optimisations qui étaient exclusivement l'apanage du processeur ou du compilateur sont maintenant réalisées par les deux. La frontière entre les optimisations qui relèvent de la compilation et celles qui relèvent de l'exécution devient de plus en plus floue : les compilateurs connaissent de nombreux détails architecturaux et sont capables d'ordonnancer des instructions de façon à diminuer l'impact des latences ou des défauts de cache. Inversement les processeurs peuvent par exemple renommer les registres pendant l'exécution et changer l'ordre des instructions exécutées.

Les auteurs de [2] considèrent l'espace des optimisations comme un *continuum* qui commence au moment de la compilation et s'étend jusqu'à l'analyse à la fin de l'exécution. Les étapes intermédiaires sont l'édition des liens, le chargement et l'exécution. Plus une optimisation est repoussée vers le moment de l'exécution, plus elle a de chances d'être efficace car elle dispose de plus d'informations (à supposer qu'elle n'en perde pas à chaque étape), mais plus elle risque d'être pénalisante. Idéalement toute information concernant l'application est stockée dans une base de données et accessible à tout moment. La conception de cette base reste évidemment problématique.

1.4 Profiling

Les transformations de code ont pour but d'améliorer le comportement d'une application relativement à l'architecture matérielle. Avant de commencer à optimiser, le programmeur doit identifier les facteurs limitant la performance et faire appel à des techniques d'analyse.

Les techniques d'analyse de code se divisent en deux catégories : les techniques statiques qui extraient de l'information de l'étude du programme (code source, assembleur ou exécutable), et les techniques dynamiques qui nécessitent l'exécution du programme, qu'elle soit réelle ou simulée.

Le *profiling* permet de recueillir des informations qui ne peuvent pas être déduites de l'analyse statique du programme. Elles sont mesurables uniquement pendant l'exécution. Les techniques qui permettent de les obtenir sont classées en plusieurs catégories : instrumentation, échantillonnage, émulation et utilisation des dispositifs matériels.

1.4.1 Instrumentation du code

La technique appelée *profiling* consiste à caractériser le comportement d'un programme grâce à un ensemble de valeurs collectées pendant l'exécution. Il s'agit généralement de la fréquence d'exécution de certaines sections de codes (instructions ou blocs de base), mais des informations sur le taux de défauts de cache ou la précision de la prédiction de branchement par exemple sont aussi souvent utilisées.

L'approche classique pour obtenir un *profile* est la suivante : on prend le programme original et on y ajoute des instructions qui vont servir à compter les événements que l'on souhaite mesurer. L'insertion des ces instructions se fait soit dans le code source (C, Fortran, ...), soit dans le code assembleur, soit dans l'exécutable.

Classiquement, le *profiling* est utilisé pour mesurer soit la fréquence des blocs de base, soit celles des arcs entre les blocs, soit celle de chemins qui regroupent plusieurs blocs de base [10].

1.4.1.1 Outils classiques

Le besoin d'information dynamique n'est pas récent, et un certain nombre d'outils permettent d'obtenir rapidement des résultats.

tcov réalise une analyse de couverture (*test coverage*). L'application doit être compilée avec des options spécifiques qui indiquent au compilateur d'insérer des instructions de comptage. À chaque exécution, les valeurs des compteurs sont accumulées dans un fichier. La commande `tcov` relit alors les fichiers source et le fichier de compteurs et produit des fichiers sources dans lesquels chaque ligne est annotée avec sa fréquence d'exécution (cf. figure 3.5).

pixie instrumente directement des exécutables et des bibliothèques [107]. Il réécrit le programme pour libérer trois registres dans chaque bloc de base et ajoute ensuite un compteur. Il détermine ainsi la fréquence d'exécution de chaque bloc. `pixie` est aussi capable de compter les branchements pris et non-pris et les appels de fonction. Ces résultats servent à des outils d'optimisation comme `cord` ou à d'autres outils d'analyse comme `prof`.

qpt est un outil développé par Ball et Larus [9]. L'approche est la même que celle de `pixie`, mais le nombre de compteurs insérés est minimal. En effet, les compteurs insérés par `pixie` sont souvent redondants, et donc coûteux en temps d'exécution. Que l'on choisisse d'instrumenter les nœuds ou les arcs, il est clair qu'il est inutile de les instrumenter tous. Knuth [73] a proposé des algorithmes pour déterminer le nombre minimal de compteurs nécessaires.

EEL est un environnement développé par Larus et Schnarr [77] qui permet la manipulation d'exécutables pour l'instrumentation ou l'optimisation. La complexité du processeur et du jeu d'instruction sont dissimulés grâce à une interface en C++ et l'utilisateur ne manipule que des abstractions : exécutable, routine, CFG et instruction. L'exécutable contient des routines et des données. Une routine est constituée d'un CFG et d'instructions. Le code spécifique à une architecture particulière est encapsulé pour pouvoir être intégré dans l'abstraction. La difficulté majeure à laquelle se heurte `EEL` est la construction du flot de contrôle : il doit être déduit de la table des symboles de l'exécutable, qui ne contient pas toujours les noms de toutes les fonctions (voire aucun si l'exécutable est «réduit»). Des zones de données sont souvent intercalées au milieu de zones de code, sans que les noms des labels soient significatifs.

`EEL` est conçu pour être adaptable à différentes architectures. Pourtant, à notre connaissance, seule l'architecture Sparc est supportée.

ATOM est une bibliothèque d'aide à l'instrumentation pour architecture DEC Alpha [42, 108]. Elle permet de manipuler procédures, blocs de base et instructions et d'obtenir des informations sur la taille de la pile, du segment de code, le type d'une instruction, ... Il est possible d'insérer des appels à des fonctions externes en certains points d'un programme : début ou fin de procédure, début de chaque bloc de base, etc. Des outils prêts à l'utilisation permettent de réaliser rapidement les analyses classiques : comptage du nombre d'instructions, de blocs ou d'appels système exécutés, temps passé dans chaque procédure, détail des entrées-sorties réalisées par le programme, taux de remplissage du pipeline, ...

Etch manipule des exécutables Win32/Intel dans un but d'instrumentation ou d'optimisation [103]. Le code source de l'application n'est pas nécessaire. Comme EEL, Etch doit distinguer le programme des données au sein de l'exécutable. En outre Etch doit déterminer les modules (DLL) auxquels l'application fait appel. L'interface est similaire à celle d'ATOM : il est possible d'insérer des appels de fonction avant ou après certains points. Etch a permis d'instrumenter des programmes en vue d'obtenir des simulations de cache, et d'optimiser la localité en réarrangeant les blocs de base.

Spike est un système d'optimisation d'exécutables pour Windows NT sur processeur DEC Alpha spécialisé dans le placement de code [35]. Deux composants interagissent : l'optimiseur et l'analyseur. L'analyseur instrumente directement un exécutable et toutes les bibliothèques auxquelles il fait appel en insérant des compteurs selon la technique de l'arbre couvrant de Knuth [73]. Il se charge d'exécuter automatiquement la version instrumentée ou optimisée d'une application et de la gestion des *profiles* sans que l'utilisateur n'ait à s'en soucier. L'optimisation se fait en trois étapes : placement des blocs dans chaque procédure de façon à rapprocher les blocs qui s'exécutent fréquemment en séquence, placement des procédures, et enfin *Hot-Cold optimisation* (cf. paragraphe 1.3.2.1).

Parmi ces outils, la plupart sont spécialisés dans un type d'analyse précis, et ne permettent pas à l'utilisateur de tirer parti de l'analyse qu'ils ont faite sur un programme pour obtenir d'autres informations. Les outils «ouverts» qui permettent à l'utilisateur de programmer une analyse nouvelle sont dédiés à une architecture et ne permettent pas de réutiliser un algorithme efficace avec une autre machine. Cette constatation nous a amené à développer le système SALTO [23] que nous présentons dans le chapitre suivant.

1.4.1.2 Tendance à l'intégration dans le système d'exploitation

Les outils présentés plus haut apportent une aide non négligeable au programmeur. Ils ont toutefois l'inconvénient de nécessiter une certaine expertise et sont hors de portée de l'utilisateur moyen. D'autre part les informations recueillies concernent le processus courant et ignorent les interactions avec le système et les autres processus. Pour contourner ces problèmes la tendance actuelle est d'intégrer des outils de recueil

d'information dans le système d'exploitation.

Morph est un système développé pour des stations de travail DEC Alpha sous UNIX qui optimise des applications de façon totalement transparente pour l'utilisateur [2, 122]. Le système est modifié pour échantillonner automatiquement les applications sans qu'elles nécessitent de traitement particulier. Lorsque la charge de la machine est faible, le *profile* obtenu permet de guider différentes optimisations : placement de procédure, placement de blocs de base et élimination de code mort. Le compilateur utilise SUIF [111]. Cette approche suppose que les stations de travail fonctionnent en mode mono-utilisateur et que chaque utilisateur dispose de sa propre copie des applications qu'il utilise.

DCPI (*DIGITAL Continuous Profiling Infrastructure)* [6] est semblable à Morph. Il échantillonne périodiquement les applications qui fonctionnent sur un processeur DEC Alpha grâce aux compteurs matériels. DCPI comporte trois composants : un pilote dans le noyau du système d'exploitation qui contrôle les compteurs, un démon chargé de l'extraction des échantillons et de leur écriture sur disque et un mécanisme de chargement en mémoire modifié qui mémorise les adresses des applications de façon à pouvoir associer chaque événement à un processus. Le ralentissement engendré par DCPI est inférieur à 3 %.

calvin est une approche novatrice [75] : l'ensemble du système d'exploitation d'une station de travail est instrumenté. Deux copies du système coexistent, l'une est instrumentée très légèrement, l'autre beaucoup plus lourdement. La première s'exécute presque à vitesse normale, mais ne collecte pas d'information autre que la position courante dans le code. La seconde est beaucoup plus lente, mais effectue toutes les mesures. Le basculement entre ces deux versions se fait à l'occasion d'événements prédéfinis. Cette approche permet d'utiliser la même machine pour le développement et les tests. Durant la phase de tests, la machine subit naturellement la charge de travail normale qu'est l'environnement de travail de l'utilisateur. Second avantage : le système, en charge des mesures, a accès à des ressources de la machine inaccessibles à un processus normal. Il peut aussi discriminer les événements entre les différents processus.

1.4.1.3 Gestion des traces

La récolte de traces d'exécution est très semblable au *profiling*. La différence réside dans le traitement. Une trace conserve l'ordre des événements mesurés et représente donc un volume d'information considérable. Imaginons que nous soyons intéressés par l'ordre d'exécution des blocs de base, et admettons la valeur empirique de cinq instructions par bloc. En l'espace de dix secondes sur un processeur à 200 MHz, deux milliards d'instructions sont exécutées, soit environ 400 millions de blocs. Un minimum de deux octets sont nécessaires pour coder le numéro d'un bloc, la taille de la trace approche le gigaoctet.

La manipulation de fichiers de cette taille pose des problèmes techniques de stockage et de temps d'accès, des techniques de compression s'imposent. [114] donne un aperçu de différentes techniques relatives à la manipulation de traces.

Pour éviter ces problèmes, les outils d'analyse peuvent traiter les données au vol. Si les données ne transitent pas par un disque, le traitement est d'autant plus rapide. Le passage d'information peut se faire par *socket*, c'est le cas pour `spy` de Irlam [70], par un segment de mémoire partagée, par appel direct à une procédure de l'analyseur, ...

Dans ce cas la trace n'existe jamais sous forme de fichier, et l'obtention d'une nouvelle statistique nécessite de réexécuter la totalité du programme.

1.4.2 Échantillonnage

Le surcoût induit par l'insertion de compteurs tout au long d'un programme a conduit à des approches moins pénalisantes. L'échantillonnage ne fournit pas un résultat exact au sens strict du terme. Toutefois, s'il est bien utilisé, les résultats sont parfaitement exploitables dans la majorité des cas. Plutôt que de mesurer constamment l'activité d'un programme, l'échantillonnage prélève fréquemment un point de mesure pendant un court intervalle de temps. Le choix du rapport entre le fréquence des mesures et leur durée fait la validité des résultats. Les outils classiques d'Unix `prof` et `gprof` [56] font appel à cette technique.

1.4.3 Émulation

L'émulation permet d'exécuter un programme compilé pour une architecture sur une architecture ou un jeu d'instructions différent. Chaque instruction est lue, décodée et simulée. Éventuellement toute information disponible peut être récoltée à chaque instruction. Cette approche est utilisée par `Shade` [34] pour les jeux d'instructions Sparc versions 8 et 9 et MIPS I. Le cœur de `Shade` est une boucle qui lit les instructions du programme cible une à une et les traduit en un fragment de code pour la machine hôte, à la manière d'un compilateur croisé. Puis ce fragment est exécuté. Les valeurs des registres, du compteur ordinal, de la mémoire sont maintenues à jour. Le fragment de code est conservé de façon à pouvoir être utilisé si la même instruction apparaît plus tard.

1.4.4 Mécanismes matériels

Matériel de mesure externe La collecte d'information peut être faite entièrement par matériel, à l'aide de sondes branchées directement sur certaines pattes du microprocesseur. Les signaux prélevés sont stockés dans une zone tampon. Quand le tampon est plein, une interruption stoppe l'exécution du processeur pendant que les données sont écrites sur le disque d'une autre machine. L'activité normale reprend dès que le tampon est vide.

La différence de vitesse entre la collecte et le stockage nécessite d'interrompre l'activité du microprocesseur chaque fois que le tampon intermédiaire est plein, ce qui a pour conséquence de perturber la fréquence des événements extérieurs au processus. Ce phénomène, appelé dilatation temporelle, est exposé dans [114].

Cette technique reste néanmoins la méthode la plus précise pour le comptage des événements visibles de l'extérieur du processeur. Elle nécessite par contre une mise en œuvre de moyens qui la met hors de portée de la majorité.

Compteurs Les générations récentes de processeurs intègrent des compteurs matériels accessibles par logiciel. Ils allient précision de la mesure, faible surcoût du temps d'exécution et utilisation relativement simple: les événements à recenser sont spécifiés par l'écriture d'une valeur dans un registre spécial du processeur. La fréquence de ces événements est alors accumulée dans des compteurs internes. Lorsque l'un des compteurs atteint la valeur maximale, une interruption est levée pour permettre au système de mémoriser cette valeur. C'est le seul surcoût lié à l'utilisation des compteurs matériels.

Les microprocesseurs Ultrasparc, MIPS R10000 et Pentium Pro disposent de seulement deux compteurs matériels, le DEC 21164 en a trois. En conséquence seulement deux ou trois types d'événements peuvent être collectés à un instant donné. Pour recueillir plus d'information, il est nécessaire soit d'exécuter plusieurs fois l'application, soit d'échantillonner les différents événements en entrelaçant les mesures. Dans les deux cas, la précision est moindre. À l'inverse le Power2 dispose de 22 compteurs de performance et permet une bonne compréhension des événements internes au processeur.

L'incrémentation des compteurs est faite par le processeur lui-même, indépendamment du processus à l'origine de l'événement. Ce mécanisme a l'avantage de prendre en compte l'ensemble de la charge de la machine. Par contre il rend difficile la reproduction des expériences et donc la mise au point. La discrimination des événements entre processus nécessite une modification du système d'exploitation, notamment l'ajout d'informations à l'environnement du processus et leur mise à jour à chaque changement de contexte. `perfex`, disponible sous Irix, permet d'attribuer les événements au processus qui en est responsable. Il reste que les caches données et instructions sont partagés entre tous les processus de la machine, et qu'il est impossible de reproduire exactement une expérience.

Précise et performante, cette approche souffre d'un manque de compteurs sur beaucoup de processeurs. Leur utilisation reste ardue sur l'Ultrasparc et n'est pas documentée sur le Pentium. Toutefois, quand ils sont disponibles, les compteurs donnent accès à des événements internes au processeur et remplacent avantageusement la simulation.

1.4.5 Et donc...

L'obtention d'informations dynamiques sur le comportement d'un programme est relativement complexe, même si un support du système d'exploitation ou du matériel et une infrastructure adaptée facilitent le travail. En outre la validité de ces informations est sujette à caution dès que les paramètres d'entrée du programme changent et nous prenons soin de systématiquement vérifier leur validité.

Toutefois certaines informations ne peuvent être obtenues que dynamiquement et elles ont un impact certain sur l'efficacité des optimisations. Nous y avons recours dans

le chapitre 4 pour guider un algorithme d'optimisation, mais nous nous contentons d'informations statiques chaque fois que cela est possible.

1.5 Interaction avec le programmeur

Dans certains cas l'analyse du code source par le compilateur n'obtient pas suffisamment d'information pour autoriser une transformation. Si le programmeur connaît des propriétés intéressantes de son programme, il peut les indiquer au compilateur. Nous présentons ici quelques propriétés acceptées par des compilateurs tels que `tmcc` de Philips ou `cc` de Sun.

1.5.1 Pointeurs restreints

Les pointeurs du langage C sont une source d'ambiguïté considérable dans l'analyse des accès à la mémoire [5, 41, 50]. Considérons l'exemple de la figure 1.10. La norme du C (au contraire du Fortran) ne garantit pas que les tableaux `a`, `b` et `c` correspondent à des zones mémoires distinctes. En conséquence, les lectures de `b[i+1]` et `c[i+1]` sont potentiellement dépendante de l'écriture de `a[i]`. L'extension du langage C `restrict` [91] signale qu'un pointeur ne référence aucune variable ni aucun autre pointeur restreint. Le prototype de la fonction `f` devient alors :
void f(**int** n, **int** ∗ **restrict** a, **int** ∗ **restrict** b, **int** ∗ **restrict** c)

```
void f(int n, int * a, int * b, int * c) {
  int i;

  for(i=0; i < n/2; i++) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1]+c[i+1];
  }
}
```

FIG. 1.10 – *Pointeurs restreints*

C'est à l'utilisateur de garantir que les zones mémoires sont distinctes et ne se recouvrent jamais. Notons que cette construction est une source d'erreurs de programmation car le compilateur ne vérifie pas que la propriété est vérifiée. L'indépendance des pointeurs est signalée dans la déclaration de la fonction, mais elle doit être garantie pour chaque appel.

L'utilisation de pointeurs restreints sur le TM1000 avec l'exemple de la figure 1.10 permet de diviser le temps d'exécution par deux.

1.5.2 Analyse imprécise des ambiguïtés sur la mémoire

Le compilateur `tmcc` de Philips possède plusieurs niveaux d'analyse des pointeurs. Le niveau 0 correspond à une analyse classique. Les niveaux 1 et 2 font des hypothèses sur l'utilisation des pointeurs qui permettent de lever un grand nombre d'ambiguïtés. Elles ne sont pas toujours vérifiées, mais sont vraies dans la majorité des cas. Ici encore, il est nécessaire que le programmeur comprenne bien les détails de son programme et les conséquences des optimisations réalisées par le compilateur.

- Niveau 1 : le compilateur suppose qu'aucun objet pointé par `p` ne contient `p`. Si `p` pointe sur une structure, il n'y pas de champ `f` tel que `p->f==p`, si `p` est un tableau de pointeurs, il n'y a pas d'indice `i` tel que `p[i]==p`. En outre le programme ne doit pas accéder à une variable à l'aide d'un pointeur sur une autre variable, ce qui est équivalent à dire que le programmeur ne connaît pas la position relative des variables en mémoire et ne peut donc pas l'exploiter.
- Niveau 2 : toutes les hypothèses du niveau 1 sont conservées. Le compilateur suppose en plus que l'adresse d'une variable globale n'est jamais calculée. Un pointeur ne peut donc jamais désigner une variable globale.

1.5.3 Précision des calculs en nombres flottants

Les représentations des nombres flottants respectent généralement la norme IEEE 754, les additions et multiplications sur les nombres en virgule flottante ne sont ni associatives ni commutatives. Ceci limite sévèrement le nombre d'optimisations possibles.

De nombreux compilateurs supportent une option `-fastmath` qui peut être utilisée si la vitesse d'exécution importe plus que la précision des résultats.

Le compilateur `tmcc` propose deux niveaux d'imprécision appelés `dirty float` qui l'autorisent à des propagations de constantes et à des réécritures autrement illégales.

1.5.4 Pragma et options de compilation

Les pragmas sont des indications que le programmeur donne au compilateur. Il peut par exemple signaler qu'une boucle est parallèle en ajoutant la ligne `#pragma MP` (compilateur C de Sun) avant le corps de la boucle.

Les options passées au compilateur sur la ligne de commande permettent aussi au programmeur de préciser des optimisations. Par exemple le compilateur de Sun permet de demander l'*inlining* des fonctions `fonc1` et `fonc2` avec l'option `-xinline=fonc1,fonc2`. Il est possible de signaler que les paramètres d'une fonction sont des pointeurs restreints : `-xrestrict=fonc`, ou que l'utilisation de chargements spéculatifs est autorisée : `-xsafe=mem`

1.6 Quelques outils

Un grand nombre d'outils ont été développés pour améliorer les performances. Certains consistent en une chaîne de compilation complète, d'autres se positionnent en

un point précis de la compilation. Ils sont soit dédiés à une architecture, soit à usage général. Nous présentons ici quelques uns de ces outils.

IMPACT est une chaîne de compilation développée à l'université de l'Illinois [31]. IMPACT supporte des optimisations dépendantes de la machine comme l'allocation de registres ou la prédiction de branchement grâce à une trace d'exécution ainsi que des optimisations indépendantes de la machine: optimisations locale et globale, dépliage de boucles. IMPACT travaille avec une représentation intermédiaire du programme.

Le projet IMPACT s'est récemment associé au projet ReaCT-ILP de l'université de New York et le groupe CAR des HP Labs pour former l'infrastructure Trimaran, un système de compilation et d'analyse de performances intégré [115].

Suif est un acronyme pour *Stanford University Intermediate Form* [110, 111]. C'est un outil de compilation et d'optimisation construit autour d'une représentation du code intermédiaire entre l'assembleur et un langage de haut niveau. Chaque transformation est construite de manière indépendante: elle lit un programme, le transforme et l'écrit, toujours dans cette forme intermédiaire. Cette approche permet d'insérer des optimisations ou de changer leur ordre simplement en modifiant l'ordre d'exécution des phases de transformation.

alto optimise des exécutables DEC Alpha [41]. Cet outil propose de retarder le maximum d'optimisations jusqu'au moment où l'ensemble de l'application est visible en un seul bloc, c'est-à-dire pendant l'édition des liens. `alto` analyse les flots de données et permet actuellement une douzaine d'optimisations parmi lesquelles: élimination de code mort, propagation de constantes, élimination de chargement redondants, ordonnancement, ...

OM est un environnement de manipulation de code objet, de bibliothèques et d'exécutables [109]. Les fichiers lus sont convertis dans la représentation interne de OM qui consiste en une table des symboles et des instructions RTL. L'ensemble du programme est divisé en procédures, et les procédures en blocs de base. Le graphe de flot de contrôle et le graphe d'appel sont calculés. Après manipulation (analyse ou optimisation), le système génère le code à nouveau. Il permet d'appliquer des optimisations intermodules, souvent absentes des compilateurs. Srivastava et Wall l'ont utilisé pour mettre en œuvre une analyse de variables interprocédurale et le déplacement d'invariants de boucles entre procédures différentes [109].

Dynamite est un environnement d'exécution conçu pour expérimenter les transformations de code dynamiques [2]. Les instructions du programme «cible» sont traduites en une représentation intermédiaire qui est exécutée. Cette représentation est mémorisée de façon à pouvoir être réutilisée quand une instruction réapparaît. Le système produit un *profile* et détecte les sections fréquemment exécutées. Il fait alors appel à un optimiseur pour produire un code de meilleure qualité. Quand aucune optimisation ne permet d'améliorer les performances, Dynamite élargit son champ de vision et considère un bloc plus grand.

Chameleon est un compilateur développé par IBM pour la recherche sur le parallélisme d'instructions [84]. Il permet d'évaluer l'intérêt de nombreuses modifications

architecturales et leur exploitation par un compilateur. Chameleon est conçu popur les architectures VLIW, il possède un grand nombre de paramètres comme la largeur des instructions, le nombre d'unités fonctionnelles, la latence des instructions, etc.

BEG est développé à l'université de Karlsruhe [44]. Il s'agit d'un générateur de générateurs de code qui fonctionne avec des règles de réécriture d'arbres. L'algorithme tente de déterminer une couverture minimale de l'arbre de syntaxe abstraite. BEG produit aussi automatiquement un allocateur de registres.

ISDL [60, 61] est un environnement semblable à notre outil SALTO. Pour cette raison il est abordé dans le paragraphe 2.1.8.

1.7 Contexte des travaux

L'essentiel des travaux menés pendant cette thèse se sont déroulés dans le cadre du projet Oceans et ont été validés avec le processeur TM1000. Ce processeur possède une architecture régulière, et son utilisation ne nuit pas à la généralité des techniques que nous avons développées ni aux résultats que nous présentons.

La compilation pour les systèmes enfouis se distingue de la compilation d'applications classiques pour les processeurs à usage général. Nous abordons dans la première partie un certain nombre de paramètres qui contraignent le programmeur, mais aussi les degrés de liberté supplémentaires dont il profite. Nous présentons ensuite rapidement trois points qui sont développés dans la suite de ce document : le besoin d'un environnement logiciel, le besoin de propager de l'information entre les différentes phases de compilation et l'évaluation globale des performances.

1.7.1 Problématique de la compilation pour systèmes enfouis

1.7.1.1 Contraintes

Les contraintes supplémentaires associées à la compilation pour les processeurs enfouis sont liées à leur utilisation spécifique.

Temps réel Ces processeurs sont fréquemment utilisés pour piloter des systèmes complexes et doivent réagir en temps réel. Pour cela le programmeur doit par exemple garantir des temps de réponse à la levée d'une interruption. Le TM1000 est interruptible uniquement pendant une instruction de saut spéciale. Le compilateur est donc responsable de l'utilisation de cette instruction pour assurer qu'il pourra réagir suffisamment rapidement.

Coût Tous les composants du système coûtent cher. Le coût de la mémoire est proportionnel à la quantité nécessaire et donc à la taille du code. Une réduction de la taille permet l'ajout de fonctionnalités.

Co-design Les processeurs enfouis sont fréquemment conçus dans une approche *co-design* : certaines fonctionnalités jugées importantes sont implémentées par le matériel. Le compilateur doit en avoir connaissance pour tirer le maximum de profit d'une architecture particulière.

Réactivité L'architecture peut être modifiée rapidement soit pour résoudre des difficultés techniques, soit simplement pour faire évoluer la gamme de processeur. Il est essentiel que le compilateur puisse suivre l'évolution sans nécessiter de réécriture à chaque changement.

1.7.1.2 Facilités

L'absence de système d'exploitation permet d'obtenir des résultats de simulations plus fiables. Notamment le cache n'est pas pollué à cause du partage entre plusieurs applications et son comportement n'est pas surestimé par les simulations.

Une fois compilé, le programme est inscrit dans une mémoire et définitivement enfoui dans le système. Il est essentiel que ses performances soient bonnes. Ceci autorise un temps de compilation beaucoup plus long que ne le tolérerait un utilisateur sur une station de travail pour un programme classique. En levant une contrainte sur le temps de compilation, nous autorisons une analyse plus détaillée du programme et des optimisations plus agressives.

1.7.2 Nécessité d'un environnement logiciel

La taille moyenne des applications, le nombre et la complexité des transformations de code et de leurs interactions ne cessent de croître. La programmation «à la main» est devenue une utopie. Dans le contexte des systèmes enfouis la situation est encore plus critique.

À notre sens l'assistance d'un environnement spécialisé est devenue indispensable. Un tel environnement réduit considérablement les temps d'expérimentation et de développement d'une nouvelle technique, il diminue le nombre d'erreurs en permettant au programmeur de travailler à un niveau plus abstrait que celui des instructions assembleur, et donc de consacrer moins de temps aux détails d'implémentation. Le portage vers une nouvelle architecture doit aussi être rapide et simple.

Aucun environnement à notre connaissance ne permet simultanément le développement d'outils d'analyse et d'optimisation tout en restant paramétrable en fonction de l'architecture cible. Seules des chaînes de compilation complètes atteignent ce but, au prix d'un coût de maintenance élevé. Nous avons développé le système SALTO (*System for Assembly Language Transformation and Optimization*) pour faciliter l'écriture de tous les outils qui manipulent des programmes écrits en assembleur. Nous présentons SALTO dans le chapitre suivant.

1.7.3 Nécessité de véhiculer de l'information

L'intégration des techniques classiques d'optimisation permet d'établir une communication entre deux processus qui habituellement s'ignorent.

La communication est nécessaire dans deux directions : chaque transformation doit pouvoir garder une trace de son impact et transmettre l'information qu'elle a détruite. Par exemple le générateur de code, en transformant un langage de haut niveau en assembleur, perd la notion de tableau. Il doit fournir l'information sur les tableaux

aux optimisations qui en ont besoin. Nicolau *et al.* ont proposé une technique appelée *Semantics retention* [88] pour conserver des informations utiles : les instructions de lecture et écriture en mémoire sont associées à la structure de données de haut niveau qui les a produites. Le programmeur a aussi la possibilité de signaler des propriétés de son algorithme grâce à une extension du langage C.

Le flot d'information est important aussi en sens inverse : si une optimisation ne peut pas s'appliquer, ou ne peut fournir que des performances médiocres, elle peut faire remonter l'information vers les transformations qui sont à l'origine du problème. Quand l'allocation de registres devient impossible, l'allocateur peut demander qu'une boucle soit dépliée d'un facteur moindre.

Les transformations de code connues permettent de couvrir de façon satisfaisante la majorité des situations. Le réel problème est aujourd'hui d'utiliser efficacement ces transformations en les faisant communiquer de manière à éviter que l'une d'entre elles ne voie son effet limité par manque de connaissances sur le code en cours d'optimisation ou par une transformation «aveugle» précédente. Il est aussi de choisir quelles transformations appliquer et avec quels paramètres.

1.7.4 Nécessité d'une vision globale

La globalité de l'approche est un facteur déterminant. En effet les interactions entre les optimisations appliquées à différentes parties d'un programme ne peuvent pas être négligées. La seule performance à maximiser est celle de l'application dans sa totalité. Si une boucle est optimisée au détriment d'une autre, le résultat net peut être défavorable. Considérons le cas d'une optimisation qui accroît la taille du code (dépliage de boucle, *inlining*) : si la taille de la mémoire est limitée, comme c'est le cas pour les systèmes enfouis, cet espace doit être réparti de façon optimale entre les différents fragments de code. Plus généralement, une optimisation qui améliore les performances localement peut entraîner une perte plus importante dans une autre partie du programme. L'interaction entre les différentes optimisations impliquées est complexe, et un compromis global s'impose.

Chapitre 2

Infrastructures logicielles

L'évolution rapide de la technologie a permis l'apparition d'un grand nombre de fonctionnalités nouvelles dans les microprocesseurs. Le matériel est de plus en plus complexe, le parallélisme d'instructions ne cesse de croître. La ré-émergence du VLIW ne contredit pas cette tendance. Le TM1000 de Philips inclut de nombreuses instructions pour le support du multimédia [30, 52]. Le futur processeur Merced du couple Intel-HP aura aussi un goût de VLIW tout en gardant une complexité certaine [59].

Nous avons vu comment ceci rend difficile l'adaptation des outils d'analyse et d'optimisation à bas niveau. Il y a des besoins forts pour un environnement logiciel capable d'être reconfiguré rapidement afin de suivre l'évolution de la technologie ou d'évaluer différentes configurations architecturales. Un tel environnement s'impose également dans le contexte des systèmes enfouis où une réactivité certaine est nécessaire alors que les processeurs souffrent souvent de leur complexité et de leur hétérogénéité.

Les mécanismes matériels mis en œuvre dans les processeurs sont complexes et le programmeur a peu de chances d'appréhender leur impact réel sur les performances d'un programme. Certaines évolutions des processeurs sont visibles au niveau du programmeur, par exemple l'enrichissement du jeu d'instructions lors de l'extension d'une gamme, d'autres lui sont invisibles : changement de la politique de prédiction de branchements, exécution en désordre, etc. Dans une approche *co-design*, d'importantes fonctionnalités sont implémentées par le matériel et il est essentiel que le compilateur soit capable d'exploiter ce fait.

L'infrastructure logicielle susceptible d'assister le programmeur doit avoir une connaissance précise du matériel, mais doit aussi être facilement reciblable.

Ce chapitre décrit dans une première section le système SALTO que nous avons développé pour faciliter la manipulation de programmes assembleur. Ensuite nous abordons l'infrastructure logicielle SEA, construite à l'aide de SALTO, qui nous a permis d'expérimenter des stratégies de compilation et d'étudier les interactions entre transformations de code.

2.1 Un système pour la manipulation et l'optimisation d'assembleur

Nous présentons ici succinctement notre système SALTO (*System for Assembly Language Transformation and Optimization*) dont le but est de fournir à l'utilisateur un moyen de manipuler aisément des programmes écrits en langage assembleur. Le système et son interface sont décrits en détail dans le manuel [23, 101]. Les deux caractéristiques essentielles de SALTO sont l'interface orientée objet qui permet une manipulation simple de structures de données relativement complexes et le fichier de description de machine qui permet d'abstraire les algorithmes des caractéristiques matérielles du processeur utilisé. SALTO nous a déjà permis de développer rapidement des outils d'instrumentation des blocs de base ou des accès à la mémoire ainsi que des optimisations telles que l'ordonnancement local ou le pipeline logiciel.

Après avoir justifié notre choix de travailler avec des programmes assembleur, nous décrivons l'organisation générale de SALTO, puis le fichier de description de machine et l'interface orientée objet. Nous montrons comment nous avons adapté le système pour décrire une architecture VLIW. Nous donnons ensuite quelques détails d'implémentation, puis nous décrivons les réalisations qui ont utilisé notre système. Nous comparons SALTO à un autre environnement avant de conclure.

2.1.1 Le choix de l'assembleur

Nous avons choisi de développer un outil qui manipule de l'assembleur plutôt qu'un langage de haut niveau, une représentation intermédiaire ou des exécutables. Cette approche présente plusieurs avantages :

- nous évitons d'avoir à maintenir toute une chaîne de compilation. Ceci impliquerait autant de *front-end* qu'il y a de langages et autant de *back-end* qu'il y a d'architectures. De cette façon, nous nous reposons sur l'ensemble des compilateurs existants ;
- l'assembleur est encore un langage «lisible» (quoique...), ce qui facilite le développement et la mise au point d'expériences par rapport à la manipulation d'exécutables ;
- les zones de programme et de données sont facilement identifiables, les fonctions sont clairement définies ainsi que les blocs de base, ce qui n'est pas le cas dans un exécutable ;
- le code produit est directement compilable et peut être exécuté réellement, à l'inverse d'une représentation intermédiaire.

2.1.2 Un système reconfigurable

SALTO est une bibliothèque de fonctions qui fournit à l'utilisateur un moyen de manipuler simplement des programmes écrits en assembleur. Ces fonctions permettent l'implémentation d'algorithmes d'optimisation, d'instrumentation ou d'analyse statique

des caractéristiques d'un programme. SALTO est paramétrable par rapport à l'architecture cible. Il est composé de trois parties (cf. figure 2.1) :

1. le noyau effectue toutes les tâches nécessaires, inintéressantes et souvent sources d'erreurs dont le programmeur a envie de se passer, notamment l'analyse lexicale et syntaxique du code, le calcul de la structure en blocs de base et du flot de contrôle, le calcul des dépendances entre instructions ;
2. la description de la machine est un fichier qui détaille le jeu d'instructions et l'ensemble des ressources matérielles de l'architecture cible qui sont susceptibles d'intervenir dans un processus d'optimisation. Le niveau de précision est variable au gré de l'utilisateur : une description simple peut s'intéresser simplement aux unités fonctionnelles tandis qu'une description plus fine peut faire intervenir les bus d'accès à la mémoire, les ports sur le fichier de registres, etc. ;
3. l'interface utilisateur orientée objets donne un moyen d'accès simple aux structures de données internes de SALTO. Un certain nombre de classes correspondent aux types de données connus. Elles sont détaillées dans la partie 2.1.4.

Un algorithme d'instrumentation ou d'optimisation fourni par l'utilisateur utilise les fonctions de l'interface pour accéder au code et éventuellement le modifier. Même si quelques algorithmes classiques sont fournis dans la distribution, SALTO en soi n'a aucun effet sur le programme assembleur : il se contente de fournir des abstractions du code et des méthodes à-même de faciliter l'implémentation d'algorithmes. C'est à l'utilisateur de spécialiser SALTO pour obtenir un outil correspondant à ses besoins.

Les parties suivantes décrivent plus en détail les composants de SALTO visibles par l'utilisateur.

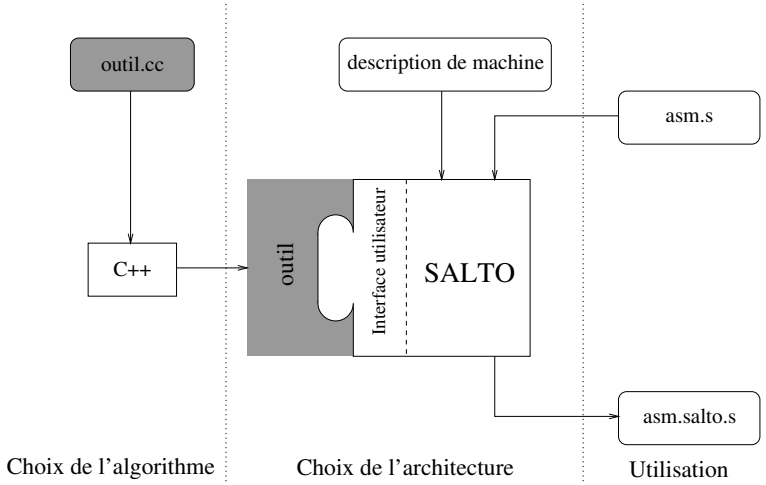


FIG. 2.1 – *Organisation de* SALTO

2.1.3 Description de l'architecture

SALTO est totalement reconfigurable par rapport à l'architecture utilisée. Ceci signifie qu'il est possible de créer ou de modifier rapidement une description de machine pour toutes les caractéristiques du processeur cible. Ce but est atteint grâce à un fichier de description de machine externe à SALTO. Ce fichier contient, dans un format lisible, toutes les caractéristiques utiles de l'architecture. Plus précisément, ce fichier est divisé en quatre sections :

- la première partie décrit la syntaxe de l'assembleur : la liste complète des mnémoniques reconnus par l'assembleur, ainsi que les différents modes d'adressage possibles ;
- la deuxième partie contient la liste de toutes les ressources matérielles existantes sur le processeur, avec pour chacune un type (registre, unité fonctionnelle ou mémoire), et un facteur de réplication ;
- l'utilisation du matériel par les instructions est ensuite détaillée à l'aide de tables de réservations dans une troisième partie ;
- la dernière partie contient éventuellement des informations sur certaines particularités architecturales comme le branchement retardé ou le «court-circuit» (*bypass*).

Le niveau de détail de la description de machine est laissé à l'appréciation de l'utilisateur. Ainsi il est possible d'avoir une vision «grossière» de l'architecture ne contenant que les registres, les unités fonctionnelles et la mémoire. À l'inverse, il est possible de prendre en compte tous les chemins de données, les ports sur les fichiers de registres ou sur la mémoire, etc.

Dans l'hypothèse où SALTO est utilisé pour faire de l'instrumentation, seule la syntaxe est nécessaire. La description du matériel est réduite au strict minimum : une seule unité fonctionnelle et toutes les instructions ont la même table de réservations.

SALTO permet aussi de définir des classes de ressources pour les besoins de l'utilisateur. Par exemple les registres entiers peuvent former une classe et les registres flottants une autre, pour les besoins d'un algorithme de renommage. Les classes peuvent être imbriquées les unes dans les autres, avec une profondeur quelconque. Nous pouvons raffiner la classification précédente en définissant la classe des registres généraux et celle des registres locaux, ces deux classes formant la «super classe» des registres entiers, elle-même faisant partie de la classe de tous les registres.

2.1.3.1 Formalisme des tables de réservations

La table de réservations [67] est un formalisme qui permet de décrire la structure du pipeline d'un processeur. Une table est associée à chaque instruction assembleur pour décrire l'utilisation des ressources à chaque étage du pipeline.

Le pipeline du processeur DLX[1] est schématisé sur la partie gauche de la figure 2.2. Les cinq étages sont : lecture de l'instruction (LI), décodage (DI), exécution (EX), accès à la mémoire (MEM) et écriture du résultat (ER). La table de réservations de

1. le processeur DLX est décrit dans [67]

l'instruction `ADDI` est illustrée sur la partie droite de la figure. `ADDI R1,R2,#3` ajoute le contenu du registre `R2` à la valeur immédiate `3` et écrit le résultat dans le registre destination `R1`. Cette instruction, comme toutes les autres, s'exécute complètement en cinq cycles d'horloge. Au premier cycle elle nécessite l'unité fonctionnelle `issue` qui lit le cache instructions, ce qui est représenté par une croix dans la table de réservations à l'intersection de la ligne `issue` et de la colonne 1. Au cycle suivant, l'instruction est décodée et les registres sont lus. La lettre `L` figure dans la deuxième colonne de la table. L'unité arithmétique et logique est utilisée au troisième cycle. Rien ne se passe au cycle 4, qui ne concerne que les accès à la mémoire. Au cinquième et dernier cycle, la valeur de résultat est écrite dans le banc de registres.

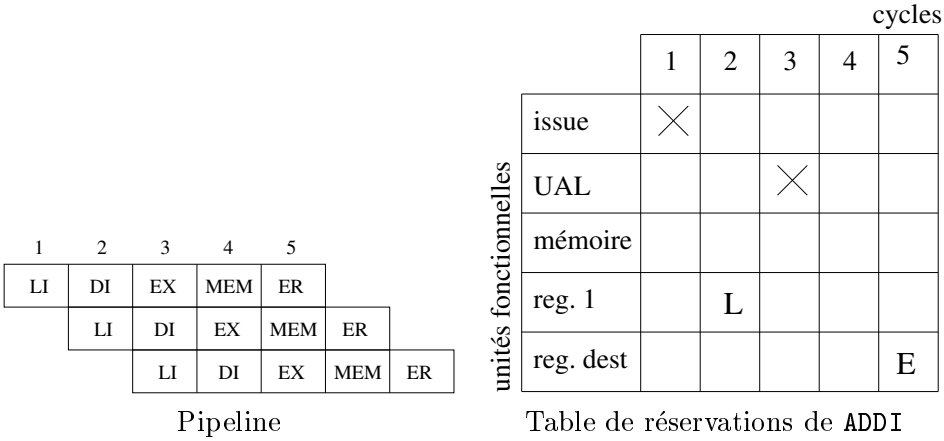


FIG. 2.2 – *Processeur DLX*

Limites du modèle Les tables de réservations se révèlent être un modèle simple et puissant pour exprimer le fonctionnement interne du microprocesseur avec toute la précision voulue. Si certains mécanismes tel que l'ordonnancement statique et les slots peuvent être décrits dans ce formalisme, il reste que certaines particularités ne peuvent être intégrées dans la description de machine.

- Citons par exemple le pipeline flottant du i860 d'Intel. Une instruction lancée dans le pipeline n'avance pas automatiquement d'un étage à chaque cycle d'horloge, mais est «poussée» par l'instruction suivante. Pour obtenir un résultat quand aucun nouveau calcul n'est lancé, il est nécessaire d'insérer des instruction *push pipeline*.
- Plus couramment, la plupart des pipelines flottants comportent un étage répété plusieurs fois. Le nombre d'itérations est dépendant des opérandes de l'instruction et ne peut donc être décrit statiquement.
- Les structures de pile créent aussi une difficulté, car s'il est possible de décrire qu'une instruction lit ou écrit une valeur dans la pile, il est impossible d'être plus précis. C'est notamment le cas avec la pile de registres flottants du Pentium, ou le *bytecode* Java.

Ce type de difficulté est inévitable, quelque soit le modèle choisi. Il est clair que la prochaine génération de processeurs proposera des mécanismes aujourd'hui inimaginables et donc potentiellement impossibles à décrire. Pour cette raison, une partie de la description peut être écrite directement en C++ dans des fonctions prédéfinies, qui seront appelées par le noyau au moment opportun.

2.1.3.2 RTL comme langage de description

La description de machine de SALTO utilise une adaptation du langage RTL (*Register Transfer Language*). C'est un langage inspiré par Lisp. Cinq types d'objets existent : expressions, entiers, entiers longs, chaînes de caractères et vecteurs. Les vecteurs sont représentés par une liste d'éléments entre crochets. Une expression est formée d'un mot-clé suivi de paramètres, le tout entre parenthèses. La grammaire complète du fichier de description de machine de SALTO peut être consultée dans [23].

La figure 2.3 contient un extrait de la description du DLX [67]. Les trois premières lignes déclarent des ressources du processeur (registres, unité arithmétique et logique, mémoire). Les lignes suivantes déclarent le mnémonique `ADDI` avec un adressage *registre + immédiat → registre*. Les caractères `1` et `d` sont des expressions régulières qui correspondent aux noms des registres. Le `i` indique une expression numérique de la forme `a+b-17` qui sera reconnue par un appel à la fonction `read_exp`.

```
 ; 32 registres flottants de 32 bits
(def_ress (base_name "%f" 0 31) [(type "reg") (width 32)] )
 ; Unité entière
(def_ress (name "alu") [(type "functional_unit")])
 ; Accès mémoire
(def_ress (name "mem") [(type "memory")])

(def_token "d" [(regex "R[0-9]\\|R[1-2][0-9]\\|R3[0-1]")])
(def_token "1" [(regex "R[0-9]\\|R[1-2][0-9]\\|R3[0-1]")])
(def_token "i" [(read_exp)])

 ; Définition de l'instruction  ADDI  registre <- registre + immédiat
(def_asm  "ADDI" [
  (input  "d,1,i")
  (reser_table [ (ress (name "issue") [(use) (at_cycle 1)]) \
                 (ress (match_arg 2) [(read)  (at_cycle 2) ]) \
                 (ress (name "alu")  [(use)   (at_cycle 3)]) \
                 (ress (match_arg 0) [(write) (at_cycle 5) ]) ])   ] )
```

FIG. 2.3 – *Extrait de la description du microprocesseur DLX*

Nous disposons actuellement des fichiers de description de machines pour les processeurs superscalaires Sparc (version 7), MIPS R4000, DEC Alpha, Intel Pentium et DLX. Le processeur VLIW TM1000 de Philips est complètement décrit, ainsi que le DSP TMS C6x de Texas Instruments.

2.1.4 Interface utilisateur orientée objets

L'interface utilisateur orientée objets est un moyen simple et efficace d'accéder aux structures de données internes complexes de SALTO. Les informations disponibles peuvent être regroupées en deux grandes catégories : celles qui concernent le code et celles qui concernent les données. Les instructions se manipulent soit individuellement, soit par bloc de base soit par procédure par l'intermédiaire des classes d'objets `INST`, `BB` et `CFG` respectivement. Les données, opérandes des instructions, correspondent à la classe `operandInfo` et les ressources à la classe `RES`. Un mécanisme générique d'attributs permet d'ajouter à un objet une information quelconque (classe `saltoAttribute`).

Les fonctionnalités de SALTO permettent un grand nombre de manipulations sur le programme assembleur :

- accès au code à trois niveaux différents : procédure, bloc de base ou instruction, avec possibilité d'ajout, de suppression et d'altération ;
- production du code (directement compilable) correspondant à un objet d'un des trois niveaux précédents, ou à tout le programme ;
- accès aux tables de réservations et calcul des dépendances entre instructions ainsi que du nombre de cycles perdus par les gels du pipeline ;
- accès aux opérandes des instructions ;
- manipulations des ressources et des classes de ressources.

Grâce à l'interface, l'utilisateur peut facilement déplacer ou dupliquer des fragments de code pour inclure des appels de fonctions, connaître l'adresse d'un accès mémoire pour résoudre des ambiguïtés, modifier un opérande pour renommer des registres, calculer des délais et déplacer des instructions pour réduire le chemin critique.

SALTO a déjà permis de construire des outils simples d'instrumentation qui ont prouvé leur efficacité pour mener quelques études sur la prédiction de branchements, la prédiction de valeurs ou le placement de code avec des programmes réalistes comme les SPEC'92 et SPEC'95 et différentes architectures. Des algorithmes d'optimisation plus complexes comme l'ordonnancement par liste fonctionnent, ainsi que le pipeline logiciel [117], l'allocation de registres [97], la formation de superblocs [69].

Deux petits exemples permettent de comprendre comment ces classes interagissent et comment elles s'utilisent pour résoudre un problème concret. L'exemple de gauche de la figure 2.4 renomme le registre `reg_s` en `reg_d` dans l'instruction `inst`. Cela se fait simplement en parcourant toutes les ressources lues par l'instruction puis toutes les ressources écrites. Si l'une d'entre elles correspond au registre à renommer, la modification est effectuée. L'exemple de droite, applicable aux architectures qui possèdent un branchement retardé, parcourt tout un programme en cherchant les instructions de branchement pour ajouter une instruction vide *nop* immédiatement après.

2.1.5 Adaptation à une architecture VLIW

Les *slots* sont une particularité du processeur TriMedia. Sur ce type d'architecture, chaque opération ne peut apparaître qu'à certaines positions dans le mot d'instruction. Prenons l'exemple du TriMedia de Philips, un processeur VLIW capable de lancer

```
void rename(INST *inst, RES *reg_s, RES *reg_d) {
  int nInput, nOutput, i;

  nInput = inst → numberOfInput();
  for(i=0; i < nInput; i++)
    if (inst → getInput(i) == reg_s)
      inst → setInput(i, reg_d);
    ... // idem pour output
}
```

```
void ajNOP(void) {
  int i, nbInst=numberOfInstructions();
  INST *inst, *nop;

  for(i=nbInst-1; i ≥ 0; i--) {
    inst = getInstruction(i);
    if (inst → isBranch()) {
      nop = newNOP();
      insertInstruction(i+1, nop);
    }
  }
}
```

FIG. 2.4 – *Deux exemples d'utilisation de* SALTO

cinq opérations par cycle : une addition peut être placée à n'importe quelle position dans l'instruction (on peut donc lancer cinq additions simultanément), par contre un branchement ne peut être placé que dans les *slots* 2, 3 ou 4 et un accès à la mémoire dans les *slots* 4 ou 5. Ce sont des contraintes supplémentaires à prendre en compte lors de l'ordonnancement.

La technique classique d'ordonnancement consiste à considérer un groupe d'instructions et à former un graphe biparti G en connectant chaque instruction à l'ensemble des *slots* qu'elle peut occuper (cf. figure 2.5). Attribuer un *slot* à chaque instruction est équivalent à trouver un sous-graphe de G dans lequel chaque nœud est de degré un, ce qui se fait en appliquant un des algorithmes classiques de la littérature sur la théorie des graphes. Sur la figure, une solution est indiquée par les arcs en gras. Sur le TM1000, 1346 combinaisons d'opérations admettent une solution au problème des *slots*.

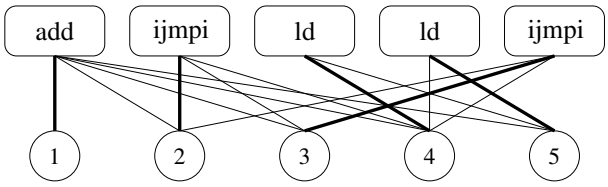


FIG. 2.5 – *Résolution des conflits de* slots *avec graphe biparti*

Cette méthode présente l'inconvénient de ne pas être adaptée au modèle de tables de réservations utilisé par SALTO, et il est nécessaire de programmer la contrainte directement en C++, donc en dehors de la description de machine de SALTO. En outre, cela était impossible pour le logiciel de calcul de pipeline logiciel PiLo (présenté dans le paragraphe 2.2.3), et plus généralement pour tout ordonnanceur. Nous avons donc proposé une nouvelle modélisation de la contrainte des *slots* qui n'utilise que le concept de ressources et le formalisme des tables de réservations, et qui s'intègre naturellement dans la description de SALTO, et par suite de PiLo. Le principe est de convertir les contraintes de *slots* en conflits de ressources en ajoutant des ressources virtuelles. Ces

ressources sont construites de la façon suivante: les opérations sont groupées en catégories, toutes les opérations d'une catégorie pouvant utiliser les mêmes *slots*. Soit n le nombre de catégories et $S_1, \ldots, S_n$ les ensembles de *slots* associés à chaque catégorie. Nous considérons alors toutes les unions possibles R_P de S_k :

$$\forall P \subset [1;n], R_P = \bigcup_{k \in P} S_k$$

et à chaque R_P nous associons une ressource virtuelle, répliquée en $Card(R_P)$ exemplaire(s) et utilisée par toutes les instructions des catégories $S_k, k \in P$.

Reprenons l'exemple précédent : il y a $n = 3$ catégories, chacune ne contenant qu'une instruction, et $S_1 = \{1;2;3;4;5\}$ pour le `add`, $S_2 = \{4;5\}$ pour le `ld` et $S_3 = \{2;3;4\}$ pour le `ijmpi`. Nous obtenons les ensembles $R_{\{2;3\}} = S_2 \cup S_3 = \{4;5\} \cup \{2;3;4\} = \{2;3;4;5\}$, et $R_{\{1;2\}} = R_{\{1;3\}} = R_{\{1;2;3\}} = \{1;2;3;4;5\}$. Nous ajoutons donc une nouvelle ressource r, disponible en quatre exemplaires et utilisée par les instructions `ld` et `ijmpi`. Les ressources associées aux ensembles $R_{\{1;2\}}$, $R_{\{1;3\}}$ et $R_{\{1;2;3\}}$ n'apportent pas réellement d'information puisque le cardinal de l'ensemble est 5 ; elles peuvent être ignorées. De même lorsque les S_k sont deux à deux disjoints, le séquencement en parallèle de ces opérations n'induit pas de contrainte supplémentaire.

Cette technique peut théoriquement engendrer un nombre considérable de ressources virtuelles (2^n, le cardinal de l'ensemble des parties de $[1;n]$, au maximum). Toutefois un grand nombre de simplifications interviennent, résultant en un nombre de ressources raisonnable. Dans le cas du TriMedia qui contient dix catégories, 17 ressources suffisent pour modéliser l'ensemble des conflits de *slots*. La preuve de la technique ainsi que les simplifications possibles peuvent être trouvées dans [13].

Grâce à cette approche, nous pouvons intégré le problème de l'affectation des instructions dans des *slots* dans le formalisme des tables de réservations. Nous pouvons ainsi utiliser les algorithmes de réordonnancement classiques pour des processeurs VLIW.

2.1.6 Détails d'implémentation

Structures de données SALTO construit et tient à jour la structure du programme assembleur grâce à plusieurs listes chaînées (cf. figure 2.6). Chaque instruction est insérée dans une liste dès qu'elle est reconnue par l'analyseur syntaxique. Des marqueurs sont ensuite insérés dans cette liste pour délimiter les blocs de base (`BASIC_BLOCK_BEGIN` et `BASIC_BLOCK_END`) et les procédures (`PROC_BEGIN` et `PROC_END`). La liste des blocs de base en est déduite, puis la liste des procédures. La liste des blocs et les instructions de saut sont ensuite utilisées pour construire le graphe de flot de contrôle. Dans le cas où l'adresse de destination d'un branchement est calculée, SALTO n'est pas capable de déterminer le (ou les) bloc(s) cible(s). Dans ce cas, il suppose que tous les blocs de la procédure sont potentiellement la cible du saut.

Chaque instruction possède ses propres informations sur ses opérandes et sa table de réservations.

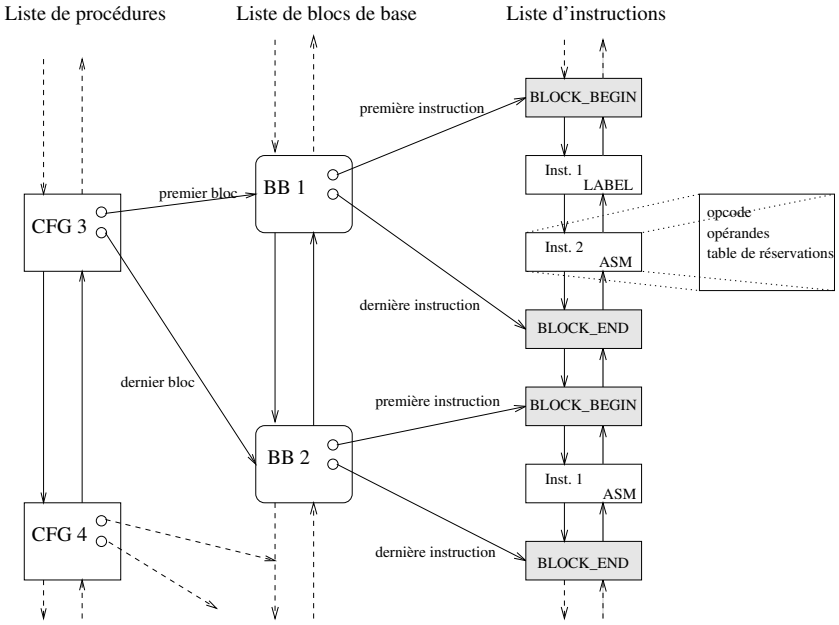


FIG. 2.6 – *Représentation interne de* SALTO

Détection des aléas et calcul des délais Proebsting et Fraser [95] présentent un algorithme de détection rapide des aléas de structure entre instructions. Le jeu d'instructions de l'architecture cible est décrit grâce à un formalisme à base de tables de réservations. Le système construit alors un automate dont les états correspondent aux états du pipeline. À chaque état est associé une matrice booléenne, et à chaque instruction un vecteur. Le test de conflit se fait par comparaison, et la transition par décalage des colonnes de la matrice suivi d'opérations logiques avec le vecteur de l'instruction.

Nous calculons les aléas de données entre différentes instructions d'un programme par simple confrontation des tables de réservations. Un aléa existe dès lors que l'une des deux instructions écrit dans un registre ou en mémoire et que l'autre lit le même emplacement ou y écrit. Le délai associé est calculé par la différence entre les positions des étages de production et de consommation des données. Dans le cas de la figure 2.7, le délai est $d = 5 - 2 + 1 = 4$.

Nous avons besoin de connaître les aléas de structure (ou conflits de ressources) pour réaliser un ordonnancement du programme. L'utilisation de chaque unité fonctionnelle à chaque cycle est tenue à jour. Avant de séquencer une instruction à un cycle particulier, sa table de réservation est comparée à l'état du processeur, et nous vérifions qu'il reste suffisamment d'unités fonctionnelles libres pour que le pipeline puisse progresser d'un étage à chaque cycle.

Prise en compte du *bypass* Le mécanisme de *bypass* est présent dans tous les processeurs pipelinés. Il permet d'utiliser le résultat d'une précédente instruction dès

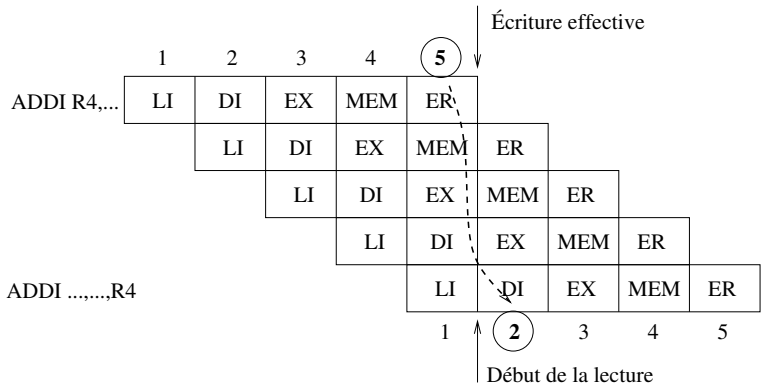


FIG. 2.7 – *Calcul des délais entre instructions*

qu'il est produit et avant qu'il ne soit écrit dans le fichier de registres. Le délai réel minimal entre deux instructions est donc inférieur à celui calculé par les tables de réservations, mais si la différence entre la valeur calculée et la valeur réelle est constante sur le Sparc (égale à un cycle), sur d'autres processeurs, tel que le MIPS R4000, elle est fonction des deux instructions concernées. Il peut y avoir autant de valeurs que de couples d'instructions.

Il était difficile d'intégrer ce mécanisme dans le fichier de description de machine, et nous avons préféré laisser la correction du délai à une fonction écrite en C++. Celle-ci reçoit en paramètre les deux instructions concernées, le type de dépendance et le délai théorique. Elle retourne la valeur corrigée.

2.1.7 Réalisations avec Salto

Le logiciel SALTO a été déposé à l'agence pour la protection des programmes (APP) sous le numéro IDDN.FR.001.070004.00.R.C.1998.000.10600. Il est distribué gratuitement sur simple demande, en version source. Le manuel de l'utilisateur [23] contient une description complète du système et de son interface.

SALTO est utilisé dans plusieurs projets de recherche, à l'Irisa et à l'extérieur et est en cours d'évaluation dans plusieurs entreprises. Il a fait l'objet de communications à des communications en France [100] et à l'étranger [102]. Nous présentons ici quelques exemples d'applications significatives d'utilisation auxquelles nous ne participons pas directement. Ces exemples illustrent le spectre d'applications qui peuvent être développées à l'aide d'un outil tel que SALTO. Les applications auxquelles nous participons sont décrites plus largement dans ce document.

calvin CALVIN (*cloning assembler and looking into veritable instrumentation needs*) est un outil développé dans le projet Caps par Thierry Lafage [75] à l'aide de SALTO. CALVIN permet la collecte de traces sur de grosses applications (coûteuses en temps). Un échantillonnage est réalisé par l'exécution alternative de deux copies du code. Une des copies est presque identique au code original et s'exécute la plupart du temps (à

une vitesse proche de celle de l'application originale). La seconde copie du code est instrumentée plus lourdement pour permettre la collecte effective de la trace. Celle-ci est exécutée avec parcimonie pour ne pas trop ralentir l'application testée. À l'exécution, certains événements permettent le basculement d'une copie à l'autre. Pour le moment, CALVIN est utilisé avec des applications mono-tâches sur architecture Sparc. Les résultats obtenus avec des applications instrumentées par CALVIN, s'exécutant sans générer de trace, ont fait apparaître de faibles ralentissements de l'exécution (20 à 30 %), ce qui ouvre la voie de la collecte de trace de l'activité complète d'une station de travail. Pour cela, CALVIN sera bientôt étendu pour permettre de tracer des applications multi-tâches ainsi que le système d'exploitation (Linux).

projet API Les processeurs programmables jouent un rôle de plus en plus important dans la conception des systèmes enfouis mais les mécanismes architecturaux de nombreux processeurs spécifiques empêchent l'utilisation de compilateurs et de langages de haut niveau.

Le projet Api de l'Irisa tente de rapprocher les processeurs à usage spécifique des outils utilisés pour la génération de code en utilisant des outils de compilation pendant la phase de conception de l'architecture. Ce processus est itératif et à chaque étape des parties de l'application sont compilées sur l'architecture hypothétique. Les résultats sont utilisés pour améliorer l'architecture, par exemple en modifiant le jeu d'instructions. Le résultat final doit consister en une architecture performante dans un domaine d'applications et un ensemble d'outils de génération de code.

Dans un tel processus, les outils de compilation doivent être facilement adaptables à différentes architectures. Le projet Api développe un environnement de conception pour compilateurs, basé sur un modèle de processeur [32]. SALTO constitue la brique de bas niveau de cet environnement. Le haut niveau produit un code séquentiel et SALTO applique des optimisations comme le compactage ou le pipeline logiciel. Dans cette approche, la description de machine de l'architecture cible est générée automatiquement à partir du modèle de processeur.

Analyse de dépendances sur la mémoire Amme, Braun et Zehendner (de l'université de Jena, Allemagne) et Thomasset (Inria) ont utilisé SALTO pour mettre en œuvre une technique d'analyse de dépendances dans des programmes écrits en assembleur Sparc [5]. L'analyse du code leur permet d'écrire les adresses des accès à la mémoire comme des polynômes du premier degré à plusieurs variables et à coefficients entiers. Les variables sont utilisées pour représenter des valeurs de registres inconnues statiquement, par exemple au début d'un procédure ou après un chargement en mémoire. Pour lever une ambiguïté, le système teste si les deux polynômes peuvent prendre les mêmes valeurs entières. Si aucune solution n'existe, les accès sont clairement indépendants.

Divers SALTO est actuellement utilisé pendant des séances de travaux pratiques dans le cadre d'enseignement d'architecture des processeurs.

2.1.8 Travaux connexes

ISDL (*Instruction Set Description Language*), décrit en détail dans [60], est un langage de description très semblable à celui de SALTO. Sa conception l'oriente vers les outils de génération automatique. Il a déjà été utilisé avec succès dans le générateur de code AVIV [65].

ISDL est utile quand les applications et les coûts requièrent une architecture dédiée, et simultanément un temps de développement court. Dans cette approche il est utile d'avoir un outil capable de produire automatiquement un générateur de code, un assembleur, un désassembleur et un simulateur à partir d'une description de l'architecture. Les choix peuvent ainsi être validés rapidement ou au contraire les difficultés répercutées sur la phase de conception.

Il s'agit donc fondamentalement d'une description comportementale du jeu d'instructions : chaque instruction décrit son action et ses effets de bord en RTL. Un poids et un coût sont aussi affectés à chaque instruction. SALTO ne propose pas explicitement cette possibilité, mais l'un des champs associé aux mnémoniques (de type chaîne de caractères) est réservé à l'utilisateur qui peut l'utiliser pour associer une sémantique. Cette fonctionnalité a été exploitée pour le développement d'un simulateur de programmes écrits en assembleur.

Les descriptions des ressources matérielles disponibles sont similaires, toutefois celle de ISDL est nécessairement très précise puisqu'elle doit être suffisante pour un générateur de code, alors que celle de SALTO n'est pas imposée. Une différence importante distingue les deux outils quant à l'utilisation des ressources par les instructions : SALTO dispose d'un mécanisme général de ressources qui sont accédées en lecture, écriture ou utilisation et de tables de réservations qui permet de modéliser tous les aléas qui viennent perturber le bon fonctionnement du pipeline. Les conflits et les délais associés se déduisent de la comparaison des tables de réservations. Au contraire, ISDL nécessite que l'utilisateur décrive explicitement toutes les combinaisons d'instructions non valides : instructions qui ne doivent pas apparaître dans le même mot VLIW ou qui doivent être espacées d'au moins un certain nombre de cycles. ISDL ne semble pas capable de gérer le concept de *slot* ou d'unités fonctionnelles indifférenciées.

Si ISDL et SALTO ont la même puissance d'expression relativement à la description d'architectures, ISDL semble plus adapté pour la génération automatique d'outils (assembleurs, simulateurs, etc.) : sa déclaration des contraintes est particulièrement inefficace et SALTO la réalise de façon plus élégante.

2.1.9 Conclusion

Alors que la complexité des architectures matérielles va croissant, le temps de développement des outils doit aller en diminuant. À la lumière des résultats que nous avons déjà obtenus, il nous semble définitivement acquis qu'une infrastructure comme SALTO est nécessaire pour le développement d'outils d'analyse et d'optimisation. Nous l'utilisons actuellement pour la construction d'une boîte à outils de transformation de code pour le parallélisme d'instructions.

2.2 SEA

SALTO est un environnement utile pour la manipulation de code assembleur, comme le prouve le large spectre d'applications d'analyse de code et d'optimisation qui ont été développées.

Nous avons choisi d'explorer les transformations de programme et leurs interactions. Nous avions besoin de pouvoir implémenter un ensemble de transformations de programmes et de les tester rapidement. Le niveau d'abstraction proposé par SALTO n'est pas adéquat, notamment la représentation est trop détaillée. Nous avons donc construit un environnement mieux adapté à cet objectif: SEA (SALTO *Enhanced Abstraction*).

Dans le première partie nous présentons la philosophie et les motivations de SEA. Nous abordons ensuite plus en détail le fonctionnement et l'utilisation de l'environnement. Dans la troisième partie nous donnons des exemples de réalisations.

2.2.1 Philosophie

L'idée sous-jacente à SEA consiste à séparer les structures de données qui représentent des fragments de code de celles qui représentent des transformations. Les objets qui représentent le code ont tous un type dont dépend leur structure, et sont capables de fournir des informations quant à leur composants. Une boucle, par exemple, connaît la liste de ses instructions, mais aussi si elle possède un prologue ou un épilogue, éventuellement ses bornes quand elles sont connues statiquement.

Chaque transformation doit répondre à un cahier des charges minimum pour être intégrée au système. Elles calquent leur comportement sur un modèle: chacune sait déterminer si elle est légale sur un fragment de code, elle sait s'appliquer, évaluer son impact, etc.

Il est ainsi possible de s'intéresser aux stratégies de compilation sans se soucier de l'implémentation des différentes optimisations qu'elle met en œuvre. De même les transformations peuvent être implémentées sans qu'il soit nécessaire de connaître le détail des objets qui représentent les fragments de code.

- La représentation sous forme de flot de contrôle est convertie en une représentation plus adaptée, qui exprime des relations de contenance et des concepts plus évolués que ceux qui apparaissent dans le graphe. La notion de boucle par exemple est essentielle pour de nombreuses optimisations. Même s'il existe des algorithmes pour les détecter dans le graphe de flot (cf. [3]), les boucles ne sont pas directement utilisables au niveau de l'assembleur. C'est le cas aussi du superbloc qui permet des transformations intéressantes. SEA dispose de représentations de ces objets et de leur caractéristiques, comme le nom du registre d'index ou les blocs qui forment le corps de la boucle.

  La figure 2.8 illustre cette transformation. SALTO se charge de la lecture et de l'analyse syntaxique du programme assembleur. La représentation interne de SALTO est utilisée pour construire la nouvelle abstraction. Il est possible d'utiliser certaines informations de haut niveau fournies par le générateur de code, comme décrit dans le paragraphe 4.3.3. C'est cette représentation qui est manipulée par

les diverses transformations de programme. Lorsque toutes les transformations ont eu lieu, les structures internes de SALTO sont modifiées pour correspondre au nouveau programme. SALTO se charge alors de produire le code sous sa forme assembleur.

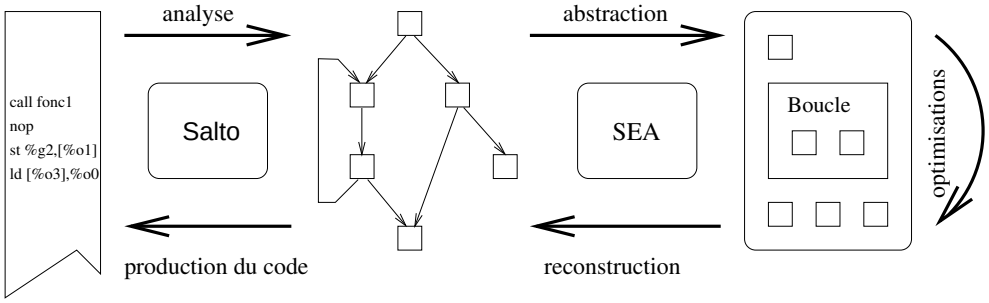


FIG. 2.8 – *Philosophie de SEA*

- Au cours de la construction de ses structures de données, SEA ne considère que les informations utiles à ce niveau d'abstraction. Les détails non significatifs tels que le code opératoire des instructions, les tables de symboles ou les structures internes de SALTO sont ignorés. Ceci est important car ces objets sont intensivement dupliqués et détruits pendant les phases d'optimisations.
  Ces informations restent accessibles si elles s'avèrent nécessaires grâce à la possibilité de consulter les structures de données de SALTO qui restent disponibles.
- Un mécanisme général est intégré à SEA pour la transmission d'information entre les différentes transformations. Nous avons défini un langage très simple qui permet la communication entres objets dans un format unique. Ce langage, nommé IL, est décrit dans le paragraphe 4.3.3.
- SEA se veut très modulaire, notamment en distinguant nettement les deux types d'objets que sont les fragments de code et les transformations. Cette «orthogonalité» permet d'écrire et de modifier facilement des optimisations sans connaître le détail d'implémentation des fragments de code, mais aussi de modifier cette implémentation sans remettre en cause la fonctionnement des transformations déjà programmées. Seules les interfaces des deux types d'objets doivent être clairement définies. La conception et la mise au point de stratégies de compilation en est aussi grandement facilitée.

2.2.2 Utilisation

SEA a été développé en C++ et utilise la bibliothèque de fonctions fournies par SALTO. Nous profitons ainsi des capacités d'analyse et de l'interface utilisateur. En outre SEA profite ainsi de l'indépendance de l'architecture cible grâce à la description de machine.

Les transformations et les fragments de code sont représentés par des objets C++.

Dans chaque cas, une classe abstraite est l'ancêtre de toutes les autres. Elle contient les informations minimales requises ainsi qu'un certain nombre de méthodes virtuelles qui doivent être surchargées par les descendants.

2.2.2.1 Fragments de code

La version actuelle de SEA définit les types de fragments de codes illustrés sur la figure 2.9. L'ancêtre commun à tous les objets définit le comportement minimal d'un fragment de code. Celui-ci doit être capable de répondre à des requêtes concernant son nombre de composants, ses successeurs et prédécesseurs, les instructions qu'il contient, ...

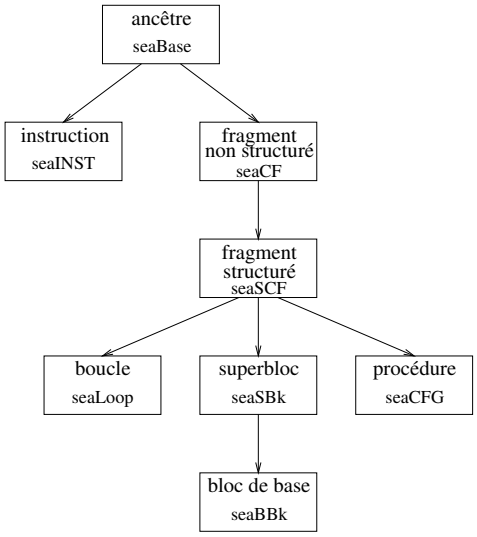


FIG. 2.9 – *Hiérarchie des fragments de code*

2.2.2.2 Transformations

De manière similaire aux fragments de code, les transformations sont des classes C++, qui héritent toutes de l'ancêtre commun `transformation`. Un sous-ensemble du graphe d'héritage est présenté sur la figure 2.10. La classe ancêtre définit les fonctionnalités minimales que doivent posséder toutes les transformations. La figure 2.12 reprend sa définition. Une transformation doit être capable d'une certaine «autonomie» : elle peut déterminer si elle est applicable, elle sait s'appliquer à un fragment de code et surtout elle peut évaluer son action. En cas d'échec, un mécanisme de diagnostic permet d'en connaître les raisons.

Les quatre fonctionnalités essentielles que doit définir une transformation sont les suivantes :

Test de légalité : une transformation est capable de déterminer si elle peut s'appliquer à un fragment de code. Pour chaque fragment proposé une fonction doit

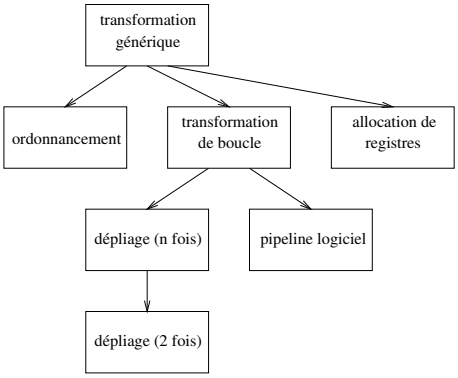


FIG. 2.10 – *Hiérarchie des transformations de code*

retourner une valeur booléenne. La complexité du test est variable, mais reste à la charge de la transformation. Tester si le dépliage s'applique consiste simplement à vérifier que l'objet est une boucle. Par contre le blocage d'un nid de boucles nécessite de calculer les dépendances entre instructions.

Application : Après avoir vérifié qu'elle est légale, une transformation sait s'appliquer à un fragment de code. Elle fabrique pour cela un nouveau fragment retourné par la fonction.

Évaluation : Quand la transformation a été appliquée avec succès, elle doit être en mesure de quantifier son impact selon des critères qui lui sont propres. Cette mesure est utilisée par un algorithme (par exemple d'optimisation) pour évaluer l'opportunité de la transformation.

Analyse : Étant donné un fragment de code, une transformation peut analyser le code pour trouver des sous-fragments auxquels elle pourrait s'appliquer. La fonction renvoie un ensemble qui contient tous les sous-fragments candidats à une optimisation. Considérons le programme représenté sur la moitié gauche de la figure 2.11. Il consiste en un bloc de base suivi d'une grande boucle qui contient elle-même deux petites boucles ainsi qu'un bloc de base. La transformation «dépliage» est capable de parcourir récursivement tout le programme pour en extraire les trois boucles (partie droite de la figure), c'est-à-dire tous les fragments auxquels elle pourrait s'appliquer.

Tous les points importants d'une stratégie de compilation, c'est-à-dire l'estimation de la validité d'une transformation, son application et son évaluation sont pris en charge par la transformation elle-même. Il nous semble en effet fondamental qu'une transformation de code soit «responsable» de son impact et elle est généralement la seule à disposer de suffisamment d'informations pour fournir une évaluation fiable.

Reconstruction du programme La reconstruction du programme après transformation pose un réel problème : généralement seulement une partie du code assembleur a été traitée par SEA. Dans ce fragment, des boucles ont été dépliées ou transformées par

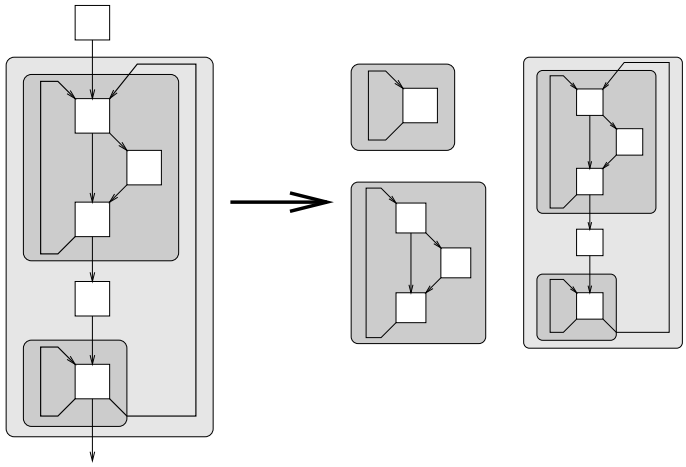


FIG. 2.11 – *Recherche de fragments susceptibles d'être optimisés*

pipeline logiciel, des blocs de base ont été supprimés ou dupliqués (cf. 2.13). Par contre le reste du code reste inchangé. Il convient donc de s'assurer que les «connexions» entre les deux zones de code (de l'extérieur vers l'intérieur et de l'intérieur vers l'extérieur) restent valides. Cette difficulté est en grande partie résolue par l'utilisation d'un objet de la classe `seaSCF` qui ne possède qu'un point d'entrée.

2.2.2.3 Stratégies

La mise au point d'une stratégie d'optimisation à bas niveau à l'aide de SEA est aisée. Pour l'illustrer nous considérons une stratégie que nous utilisons au chapitre 4 à haut niveau, mais que nous appliquons ici à bas niveau. Une boucle est dépliée tant que le gain de performance estimé par rapport à l'itération précédente dépasse 10 %. L'estimation statique du nombre de cycles nécessaires à l'exécution d'un corps de boucle sur un processeur VLIW consiste simplement à compter le nombre d'instructions (nous ignorons ici les effets du cache). La figure 2.14 montre le code C++ qui réalise cette stratégie.

2.2.3 Réalisations avec SEA

L'infrastructure SEA a donné lieu à plusieurs réalisations qui ont confirmé l'intérêt de ce type d'outil. Nous présentons brièvement son intégration le projet Oceans. Nous développons ensuite la construction d'un environnement d'expérimentation du pipeline logiciel à partir d'un outil abstrait existant et de SEA. Une étude du compromis entre la taille d'un programme et sa performance, présentée au chapitre 3, a aussi utilisé cette infrastructure.

Projet européen Oceans La majeure partie du travail de définition et de réalisation exposé dans ce chapitre a été réalisée dans le cadre du projet européen LTR Esprit

```
typedef enum transfStatus { notApplied, success, failure } transfStatus;

class transformation {
public:
   // Constructeurs - Destructeurs
   transformation();
   transformation(char *theName);
   virtual ~transformation();

   char *getName();
   void setName(char *name);
   transfStatus getStatus();

   // Méthodes à surcharger obligatoirement
   virtual Set *preCond(seaSCF *) = 0;
   virtual seaCF *apply(seaCF *) = 0;
   virtual bool isValid(seaCF *) = 0;
   virtual int evaluate() = 0;

   virtual void report(seaCF *from, seaCF *to, FILE *where = stdout);

   // Diagnostiques
   void setDiag(int n);
   void clearDiag(int n);
   bool isDiagSet(int n);
   void resetDiag();
protected:
   ...
};
```

FIG. 2.12 – *Définition d'une transformation*

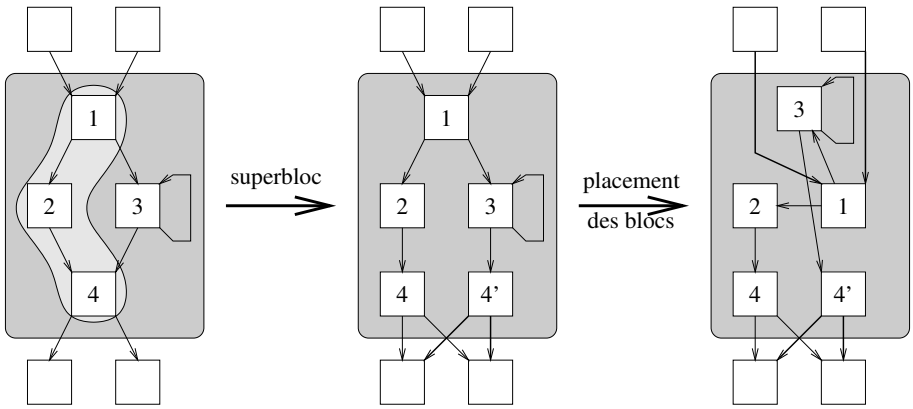
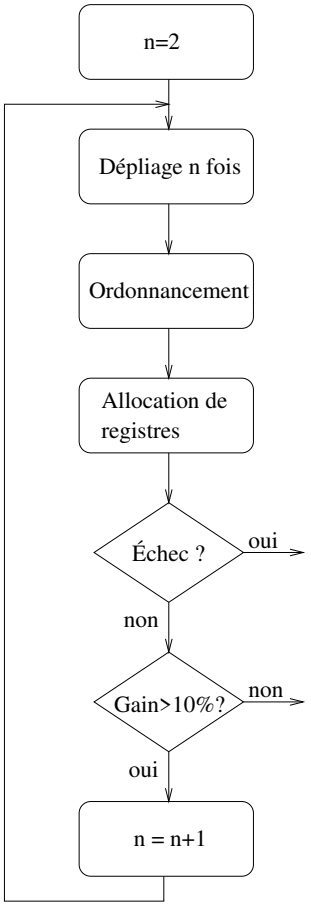


FIG. 2.13 – *Reconstruction du programme par SEA*




```
#include "unroll.h"
#include "tm1Sched.h"

seaCF *strategie(seaLoop *code, int maxUnroll) {
  int v, precV;
  unroll *u;
  tm1Sched *sched;
  seaCF *code2, *precCode = NULL;

  u = new unroll();
  sched = new tm1Scheduler();

  for(i=2; i < maxUnroll; i++) {
    u → setUnrollFactor(i);
    code2 = u → apply(code);
    if (u → getStatus() == failed) seaFatal("Erreur\n");
    sched → apply(code2);
    if (sched → getStatus() == failed) seaFatal("Erreur\n");
    v = sched → numberOfCycles();
    if ((i>2 ) && ((precV-v)/precV < 0.1)) break;
    precV = v;
    precCode = code2;
  }
  return precCode;
}
```

FIG. 2.14 – *Implémentation d'une stratégie avec SEA*

Oceans. Nous exposons ici l'intégration de SEA dans l'environnement de compilation. Le projet européen Oceans (*Optimizing Compilers for Embedded Applications*) a pour but de concevoir et de développer un compilateur qui intègre des optimisations de haut et bas niveau et utilise des techniques d'analyses très agressives. L'approche consiste en trois points : tout d'abord des restructurations de haut niveau, comme la transformation des structures de données, doivent être définies. Ensuite des optimisations de bas niveau doivent prendre en compte les spécificités des systèmes enfouis VLIW ou superscalaires et être facilement paramétrables en fonction de l'architecture. Enfin ces tâches doivent être intégrées grâce à un mécanisme de rétroaction entre les deux étages du compilateur.

L'architecture globale du compilateur retenue pour le compilateur est présentée sur la figure 2.15. Les transformations de haut niveau sont appliquées au programme Fortran par le système de transformation MT-1 [18] de l'université de Leiden. Le générateur de code utilise le programme optimisé pour produire un code assembleur séquentiel. Ce code utilise des registres virtuels et annote les instructions de façon établir une correspondance entre les objets visibles à haut et à bas niveau. Le fichier `fichier.il` contient

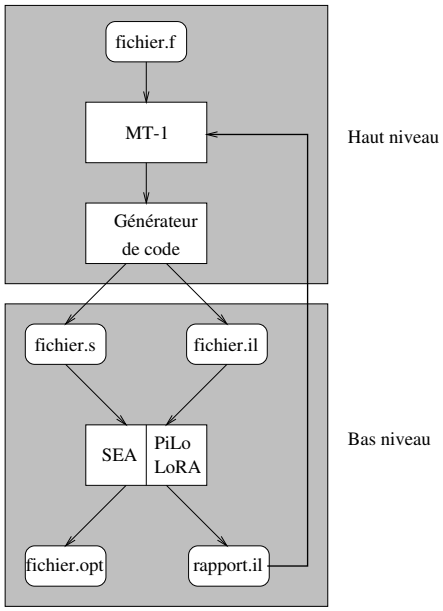


FIG. 2.15 – *Infrastructure du compilateur Oceans*

des informations utiles sur la structure du programme assembleur qui sont «oubliées» pendant la phase de génération de code. Il est détaillé dans le paragraphe 4.3.3.

SEA prend en charge les optimisations du code assembleur, l'ordonnancement, l'allocation des registres et la génération du code. La mise en œuvre de la boucle de rétroaction est aussi à sa charge.

Pipeline logiciel Le pipeline logiciel est une technique qui permet d'augmenter le parallélisme d'instructions au sein d'une boucle. Elle consiste à briser la structure de la boucle originale et à reconstruire un nouveau corps de boucle à partir d'instructions qui proviennent d'itérations différentes. C'est aussi une technique complexe qui nécessite de résoudre de nombreux sous-problèmes :

- il est nécessaire de propager les informations disponibles à haut niveau sur les dépendances entre accès à la mémoire. En cas d'ambiguïté, il est nécessaire de supposer que des accès sont dépendants. Si les ambiguïtés sont trop fréquentes, il est impossible d'atteindre un parallélisme d'instructions élevé.
- dans le cas d'architectures VLIW, l'ordonnancement statique ajoute une difficulté. Lorsque le nouveau corps de boucle est construit, il reste à ajouter l'instruction de saut de la boucle. Or cet ajout ne peut pas provoquer de décalage dans le code sans risquer d'altérer la correction de la transformation. La partie gauche de la figure 2.16 est un exemple de corps de boucle. Les arcs indiquent les dépendances de données entre instructions (en supposant une latence de trois cycles pour `ld`). La valeur lue est propagée au deuxième `add`, le premier recevant une valeur calculée précédemment. S'il s'avère nécessaire d'insérer une instruction VLIW pour

placer le branchement, la valeur de `r4` est incorrectement propagée au premier `add`.

FIG. 2.16 – *Ordonnancement statique et insertion d'un branchement*

- l'existence de gardes complique l'analyse des dépendances. En effet il est impossible de savoir statiquement si une instruction gardée modifie un registre. La durée de vie du registre destination est donc une inconnue supplémentaire.
- il peut être nécessaire de déplier la boucle pour adapter la durée de vie des registres à l'intervalle d'initialisation.
- les registres doivent être renommés.

Ces problèmes ne présentent pas de difficulté théorique majeure et sont pour la plupart bien connus. Toutefois ils ajoutent à la complexité du programme et influent de façon non négligeable sur le temps nécessaire à l'implémentation d'un pipeline logiciel et sur la robustesse. Cet aspect est d'autant plus crucial que le pipeline logiciel n'est pas toujours la meilleure optimisation possible pour une boucle. Il n'est pas applicable systématiquement, et s'accompagne d'une forte expansion du code qui peut être dommageable pour le comportement de la hiérarchie mémoire ou le coût du système enfoui. D'où l'intérêt d'un système qui permet d'évaluer le gain apporté par cette technique en fonction de l'architecture et des applications avant de décider de l'incorporer dans une chaîne de compilation.

Nous considérons ici l'algorithme du pipeline logiciel comme une «boîte noire». Le chapitre suivant (3.2.2.1) traite en détail de ses effets sur les performances et la taille du code produit. PiLo et LoRA sont des outils d'optimisation de boucles développé à l'Inria. PiLo calcule un ordonnancement modulo d'une boucle selon l'algorithme DESP [117]. LoRA prend en charge le renommage de registres associé à l'ordonnancement modulo. Ces outils travaillent sur une représentation abstraite de l'architecture (à l'aide de tables de réservations) et du programme (sous la forme du graphe de dépendances de données).

Nous avons bâti le système SPS [20] en interfaçant SALTO avec PiLo et LoRA de façon à avoir une chaîne complète, qui part d'un programme assembleur et fournit un autre programme assembleur. La figure 2.17 illustre l'organisation de SPS. SALTO reçoit trois fichiers en entrée : la description de machine (MDF), le code assembleur à optimiser (`seq.s`) et un fichier (ILF) qui contient des informations utiles relatives au fichier assembleur. Ce dernier fichier est produit par le générateur de code. Typiquement, il contient la liste des instructions qui font des accès dépendants à la mémoire (information

difficile à obtenir au seul vu de l'assembleur) et la structure des boucles (nom du registre d'index, espace d'itération quand il est connu statiquement, etc.). SALTO utilise ces trois fichiers pour produire un fichier dans le format reconnu par PiLo. Il contient les détails suivants :

1. les tables de réservations des instructions de la boucles ;
2. la liste des unités fonctionnelles existantes ;
3. le nombre et le type des registres disponibles, les conventions utilisées par le compilateur ou l'architectures : certains registres sont constants, certains contiennent des valeurs spéciales comme l'adresse de retour de la procédure courante, et ne doivent donc pas être renommés ;
4. les dépendances de données entre instructions.

PiLo et LoRA font appel à des techniques de transformation et de coloriage de graphes pour calculer la nouvelle structure de boucle. Le fichier résultat (OSPF) contient l'information suffisante pour construire la boucle, ce que fait SALTO avant de générer le code assembleur ordonnancé. Si nécessaire, SALTO peut aussi produire un fichier ILF qui décrit la nouvelle boucle pour un traitement ultérieur.

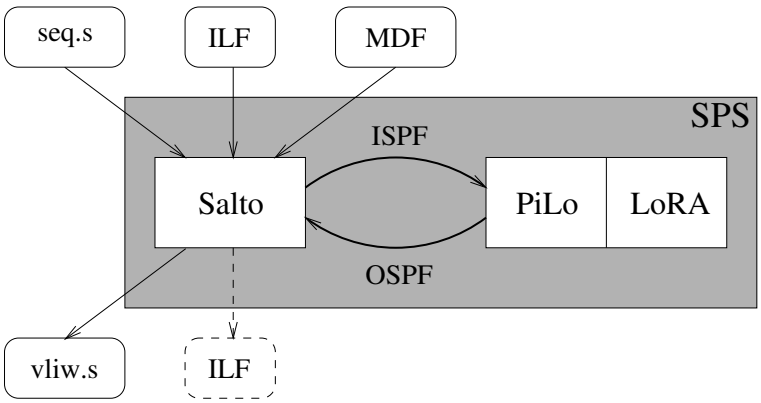


FIG. 2.17 – *Organisation de SPS*

Tous les flots d'information entre les différentes parties du système sont des fichiers «texte», ce qui facilite la mise au point en cas de problème.

Compromis taille/performance Les optimisations qui permettent d'accroître le parallélisme d'instructions ont une fâcheuse tendance à augmenter aussi la taille du code. Quand cette expansion est trop forte, elle devient dommageable pour les performances, par exemple à cause de la dégradation du comportement du cache instructions. Pour les systèmes enfouis, la taille du code est aussi un paramètre critique puisqu'elle conditionne le coût.

Nous avons développé une stratégie de compilation adaptée à la recherche d'un compromis entre la taille du code généré et la performance atteinte à l'aide de SEA. Cette étude fait l'objet du chapitre suivant.

2.3 Conclusion

Les deux infrastructures que nous avons développées : SALTO et SEA nous ont permis de développer et mettre au point rapidement des expériences. Nos principaux résultats, exposés dans les chapitres suivants, ont été obtenus à l'aide de ces outils. De nombreuses autres manipulations sur des fichiers assembleurs ont été rendues possibles, notamment des vérifications de résultats, études de conjectures. Il est clair sans ces environnements, elles auraient été beaucoup plus laborieuses.

Nous sommes fermement convaincus de la nécessité de disposer de tels outils pour expérimenter de nouvelles approches et les valider. La diffusion de SALTO dans d'autres équipes et l'intérêt de quelques industriels nous conforte dans cette idée.

Chapitre 3

Une stratégie globale de compilation

L'approche classique choisie par la majorité des compilateurs actuels consiste à décomposer le programme source en un certain nombre de parties qui sont compilées indépendamment les unes des autres. Ces parties sont généralement les différentes fonctions et procédures qui constituent le programme. Certains outils d'optimisation intègrent un module d'analyse interprocédurale qui permet ensuite d'analyser l'interaction entre une procédure appelante et une procédure appelée, mais ils restent peu développés. Quelques approches pour s'affranchir de ce cloisonnement artificiel ont été proposées [58, 63] : elles consistent à définir une notion de «région» qui permet au compilateur de travailler sur des fragments de codes provenant de différentes procédures.

Toutefois aucune des approches actuelles ne s'intéresse à l'interaction globale des optimisations elles-mêmes. En effet l'impact de chaque optimisation sur tel ou tel fragment de code a une répercussion au niveau global, par exemple sur la taille du code, le temps d'exécution, la pression sur les registres, etc. L'amélioration d'une propriété s'accompagne souvent de la dégradation d'une autre et des compromis sont nécessaires.

La structure des compilateurs n'est pas adaptée à la prise en compte de tels phénomènes. Intégrer cette dimension impose de reconsidérer la structure du compilateur.

Dans ce chapitre nous commençons par présenter l'importance de la recherche d'un compromis entre taille du code et performance, puis nous l'utilisons pour illustrer notre approche globale de l'optimisation. Nous détaillons ensuite nos expérimentations et nos résultats. Enfin nous concluons le chapitre.

3.1 Compromis taille/performance

La plupart des transformations de programme qui visent à augmenter le parallélisme d'instructions ont aussi pour conséquence d'augmenter la taille du code (citons par exemple le dépliage de boucle, l'*inlining*, la construction de superblocs,...). Cet effet secondaire risque d'avoir des conséquences désastreuses sur les performances si le comportement du cache instructions est dégradé.

Dans le contexte des systèmes enfouis, où la taille du code est limitée par la capacité d'une PROM, taille du code et performance sont traditionnellement des paramètres cruciaux. La conséquence directe est soit l'écriture des parties critiques à la main, soit le surdimensionnement du matériel qui naturellement implique un surcoût. Avec l'apparition des nouveaux microprocesseurs comme le TriMedia ou les C6x de Texas Instruments, l'ordre de grandeur de la taille des applications change, et le codage manuel devient extrêmement coûteux.

Pour être adopté par cette communauté, un compilateur doit être capable de générer un programme rapide, tout en contrôlant l'expansion de code. Il lui faut donc explorer localement un grand nombre de techniques de génération de code, d'allocation de registres, d'ordonnancement. Ensuite il doit faire un choix entre toutes les variantes étudiées pour déterminer le meilleur compromis entre taille du code et performance. Pour résumer, un compilateur pour systèmes enfouis doit pouvoir répondre à l'une des questions suivantes :

> «Étant donnée une taille de code maximale, quelle est la meilleure performance possible?»

ou la question duale :

> «Étant donnée une performance minimale, quelle est la taille de code minimale possible?»

3.2 Prise en compte des contraintes globales

3.2.1 Principes

Les compilateurs traditionnels ne prennent pas en compte de contraintes globales. Ils appliquent systématiquement une heuristique immuable sur chaque fragment de code. Dans le meilleur cas, un ensemble de quelques heuristiques est prévu, mais la décision est toujours prise au niveau local grâce à une approximation de la performance. Et il est impossible d'exercer un contrôle sur la taille du code.

A contrario, l'idée qui motive notre approche, GCDS (pour *global constraints driven strategy*), est la prise de décision globale, au vu des résultats locaux atteints par différentes optimisations sur différentes sections de code. Le principe de GCDS est illustré sur la figure 3.1. Il repose sur trois étapes :

1. Pour chaque fragment de code susceptible d'être optimisé, un certain nombre de transformations sont appliquées. Les propriétés significatives de chaque variante du code produit sont mesurées, par exemple la taille du code et le temps d'exécution.
2. Ces informations sont fournies à un module de décision qui choisit l'optimisation la plus appropriée à chaque fragment de code, de sorte à obtenir le meilleur compromis possible, selon les critères retenus.
3. Le code final est généré en utilisant les versions appropriées de chaque fragment.

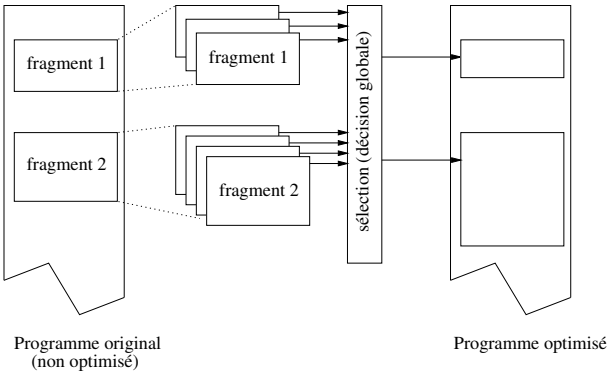


FIG. 3.1 – *Principe de* GCDS

3.2.2 GCDS et le compromis taille/performance

Le paragraphe 3.1 présente les enjeux du compromis entre la taille du code produit et sa vitesse d'exécution. Cette partie s'intéresse plus précisément à ce problème sans que cela nuise à la généralité de l'approche que constitue GCDS. Nous nous sommes attachés à l'optimisation des boucles à bas niveau, puisque que ce sont nécessairement les boucles qui sont les parties critiques d'une application, ainsi que la possibilité d'*inliner* des fonctions. Nous présentons d'abord le modèle de boucle qui nous permet d'évaluer les performances, puis nous proposons une fonction de sélection qui réalise le compromis entre la taille du code et la performance.

Chaque fragment de code retenu pour optimisation est dupliqué autant de fois qu'il y a d'optimisations. Une optimisation est appliquée à chaque copie et les paramètres qui nous intéressent (taille du code et nombre de cycles) sont évalués. Ces valeurs sont utilisées pour déterminer quelle optimisation doit finalement être choisie pour chaque fragment.

3.2.2.1 Modèle de performance pour les boucles

Pour chaque version de chacune des boucles transformées, il est nécessaire d'évaluer la taille du code ainsi que la performance, c'est-à-dire le nombre de cycles nécessaires au processeur pour exécuter l'ensemble du code. Pour cela nous introduisons le modèle suivant : à toute boucle L nous associons le nombre de cycles par itération (ou vitesse asymptotique) noté a, et le surcoût de la boucle, noté b, qui correspond au temps passé dans la boucle en sus de la partie itérée (par exemple au prologue et à l'épilogue de la boucle). La valeur de b peut dépendre du nombre d'itérations n de la boucle. Le *poids* de la boucle W est le nombre d'exécutions de la boucle dans l'application. Certaines transformations de code ne sont valides que lorsque le nombre d'itérations dépasse une certaine valeur. Nous notons cette valeur seuil min_{it}. Avec ce modèle simple, le temps d'exécution de la boucle L est donné par :

$$\text{si } n > min_{it} \text{ alors } t = W \times (a.n + b(n)) \tag{3.1}$$

L'hypothèse implicite faite jusqu'ici est l'existence des valeurs n et W. Elle implique deux choses :

1. chaque exécution de la boucle au sein d'une exécution de l'application est identique aux autres, c'est-à-dire n constant ;
2. l'exécution de l'application avec différents jeux de données conduit aux mêmes comportements, notamment W constant.

Il est clair que cette hypothèse est trop forte. Toutefois si les différentes valeurs de W gardent approximativement les mêmes valeurs relatives – intuitivement si les boucles critiques restent les mêmes – d'une exécution à l'autre, alors il est possible de considérer les valeurs moyennes correspondant à chaque valeur de n. Le paragraphe 3.4 montre que cette nouvelle hypothèse est raisonnable. Notons $W^{(k)}$ et $n^{(k)}$ les différentes valeurs de W et n, et nbn le nombre de valeurs différentes, l'équation 3.1 devient :

$$t = \sum_{k=1}^{nbn} W^{(k)} \times (a.n^{(k)} + b(n^{(k)})) \tag{3.2}$$

Dans la suite de cette étude et jusqu'à ce que les résultats soient présentés, nous utilisons les notations de l'équation 3.1, de façon à alléger le texte.

Les valeurs de W et n sont des caractéristiques de l'application et sont données par *profiling*. Par contre les valeurs de a et b dépendent aussi de la transformation que subit la boucle. Les trois exemples suivants présentent le calcul dans le cas de l'ordonnancement simple du corps de la boucle, dans le cas d'une boucle dépliée et dans le cas du pipeline logiciel.

Ordonnancement simple L'ordonnancement ne modifie pas la structure de la boucle, seules les instructions qui composent le corps changent de place. La valeur de a est donc le nombre de cycles de la boucle après transformation, et $b = 0$.

Dépliage de boucle Le dépliage est une opération plus complexe. Dans notre étude, le dépliage est suivi d'une phase de construction de superblocs [69] et d'insertion de gardes qui permettent de supprimer les sauts (voir la figure 3.2). L'ordonnancement local a ainsi une plus grande marge de manœuvre en travaillant sur un bloc plus grand. Par contre tout le corps de la nouvelle boucle est exécuté à chaque itération. Si u est le facteur de dépliage, et que n ne soit pas un multiple de u, les $n - (n \bmod u)$ dernières itérations seront exécutées inutilement et annulées par une garde. La valeur de b dépend de n, caractéristique de l'application, aussi bien que de u, caractéristique de la transformation de code. Le surcoût est donné par la formule suivante :

$$b(n) = \begin{cases} (u - (n \bmod u)) \times a & \text{si } (n \bmod u) \neq 0 \\ 0 & \text{sinon} \end{cases} \tag{3.3}$$

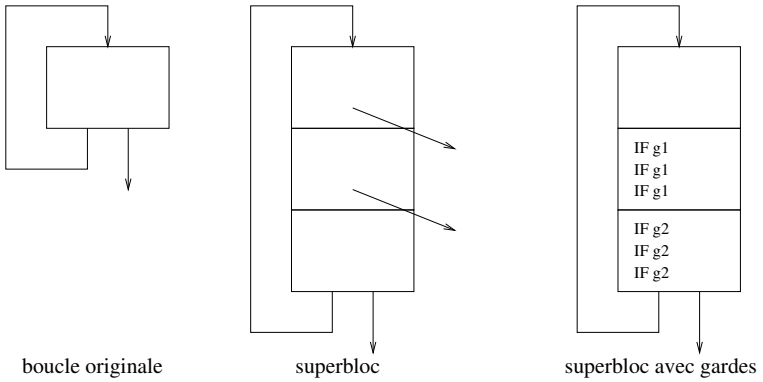


FIG. 3.2 – *Dépliage de boucle et insertion de gardes*

Pipeline logiciel Le pipeline logiciel est sans doute une des transformations de boucles les plus complexes qui soient. De nombreuses variantes sont présentées dans [4]. La structure de la boucle originale est entièrement détruite et le nouveau corps de boucle contient des instructions qui proviennent de différentes itérations de la boucle de départ (voir le paragraphe 1.3.2.2). La figure 3.3 schématise le processus de transformation. Le pipeline ne peut exécuter qu'un nombre d'itérations de la forme $k \times U + c$ (où U est le facteur de dépliage et c le nombre d'itérations lancée par le prologue), et au minimum min_{it}. Or le nombre d'itérations n'est souvent connu qu'à l'exécution. Il est donc nécessaire d'avoir aussi une petite boucle qui exécute soit les itérations restantes quand le pipeline logiciel s'applique, soit l'ensemble des itérations s'il ne s'applique pas ($n < min_{it}$).

Posons $r = n \bmod U$, le nombre d'itérations restantes. Si $n-r < min_{it}$ nous sommes dans le cas défavorable où le pipeline logiciel n'est pas applicable et dans ce cas $a = 0$ et $b = n \times d$, où d désigne la vitesse de la boucle de compensation. Dans l'autre cas, une itération est commencée tous les II cycles (II est l'intervalle d'initialisation). Nous obtenons donc $t = (n - r) \times II + l + r \times d = a.n + b$. En identifiant les termes, il vient $b = r.d + l - r.II$.

En résumé :

$$\begin{array}{ll} \text{si} \quad n - r > min_{it} & \left\{ \begin{array}{l} a = II \\ b(n) = r.d + l - a.r \end{array} \right. \\ \text{sinon} & \left\{ \begin{array}{l} a = 0 \\ b(n) = n \times d \end{array} \right. \end{array} \tag{3.4}$$

Nous savons désormais calculer le temps passé dans une boucle, il est facile de calculer le temps passé dans l'ensemble des l boucles retenues pour optimisation :

$$t = \sum_{i=1}^{l} t_i = \sum_{i=1}^{l} W_i \times (a_i.n_i + b_i(n_i)) \tag{3.5}$$

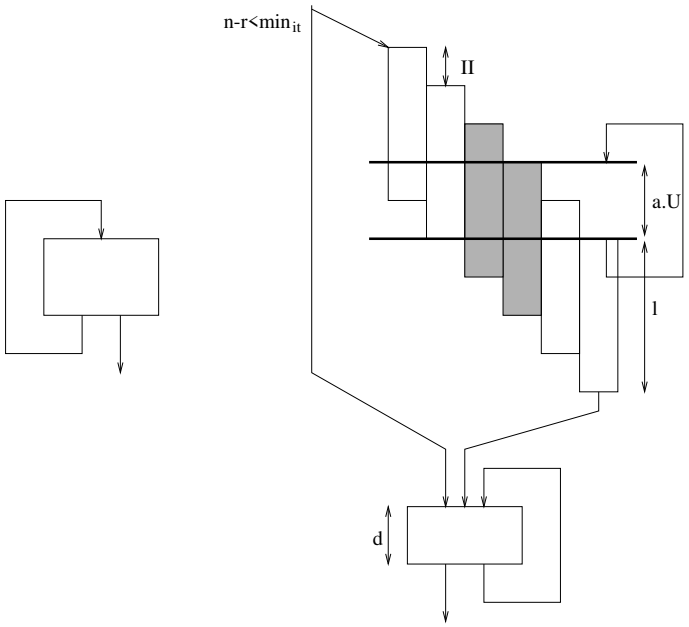


FIG. 3.3 – *Pipeline logiciel*

3.2.2.2 Résolution par programmation linéaire en nombres entiers

L'expression formelle du temps nécessaire à exécuter les boucles d'une application en fonction de l'optimisation choisie, ainsi que la taille du code résultant permet de trouver un compromis. La linéarité des équations précédentes et la taille raisonnable des systèmes permettent d'utiliser des techniques de programmation linéaire en nombres entiers.

Soit L_i une boucle, et opt_i^j les optimisations applicables à L_i pour $j \in [1, nbopt]$. Soit s_i^j la taille du code de L_i pour l'optimisation j et $t_i^j(n) = a_i^j.n + b_i^j(n)$ le nombre de cycles nécessaires à l'exécution de n itérations de L_i. Les symboles de Kronecker [1] δ_{ik} sont les variables entières qui nous permettent de coder l'association d'une optimisation à une boucle. $\delta_{ik} = 1$ signifie que le meilleur choix est d'appliquer la transformation k à la boucle L_i. Sinon $\delta_{ik} = 0$. Pour chaque boucle nous obtenons les équations entières

1. Leopold Kronecker, mathématicien allemand, 1823 (Liegnitz)–1891 (Berlin). *God made the integers; all the rest is the work of man.*

suivantes, $T_i(n)$ dénotant le temps d'exécution de la boucle et S_i sa taille :

$$T_i(n) = \sum_{k=1}^{nbopt} t_i^k(n) \times \delta_{ik} \tag{3.6}$$

$$S_i = \sum_{k=1}^{nbopt} s_i^k \times \delta_{ik} \tag{3.6'}$$

$$1 = \sum_{k=1}^{nbopt} \delta_{ik} \tag{3.6''}$$

Si n_i désigne le nombre d'itérations de la boucle L_i et W_i le poids de la boucle, nous obtenons finalement le temps d'exécution total T et la taille du programme S :

$$T = \sum_{l=1}^{nbl} W_l \times T_l(n_l) \tag{3.7}$$

$$S = \sum_{l=1}^{nbl} S_l \tag{3.7'}$$

Le choix d'un compromis entre taille et performance revient soit à minimiser S sous la contrainte $T \leq T_{max}$ soit à minimiser T sous la contrainte $S \leq S_{max}$.

Exemple Un exemple simple va éclairer cette approche. Considérons les deux boucles de la figure 3.4. Dans le deuxième fragment de code, seule la boucle interne subit des transformations. La séquence d'optimisations comprend un ordonnancement simple (noté S), un dépliage de facteur 3 (U(3)) et le pipeline logiciel (SP).

L'assembleur a été produit par le compilateur `tmcc` de Philips pour le processeur TM1000, et les transformations ont été appliquées par l'infrastructure SEA. La taille du code et le nombre de cycles exécutés sont exprimés en nombre d'instructions VLIW. Les valeurs expérimentales sont indiquées dans les tableaux de la figure 3.4. D'après l'équation 3.3, $b_1^2(n) = 0$ si n est multiple de 3 et $b_1^2(n) = 7 \times (3 - n \bmod 3)$ sinon. De même $b_2^2(n) = 0$ si n est multiple de 3 et $b_2^2(n) = 10 \times (3 - n \bmod 3)$ sinon. Les valeurs de b_1^3 et b_2^3 sont données par l'équation 3.4. L'équation 3.7 devient :

$$\begin{aligned}
T &= W_1 \times (a_1^1.n_1 + b_1^1) \times \delta_{1,1} + \\
&\quad W_1 \times (a_1^2.n_1 + b_1^2) \times \delta_{1,2} + \\
&\quad W_1 \times (a_1^3.n_1 + b_1^3) \times \delta_{1,3} + \\
&\quad W_2 \times (a_2^1.n_2 + b_2^1) \times \delta_{2,1} + \\
&\quad W_2 \times (a_2^2.n_2 + b_2^2) \times \delta_{2,2} + \\
&\quad W_2 \times (a_2^3.n_2 + b_2^3) \times \delta_{2,3} \\
S &= s_1^1 \times \delta_{1,1} + s_1^2 \times \delta_{1,2} + s_1^3 \times \delta_{1,3} + \\
&\quad s_2^1 \times \delta_{2,1} + s_2^2 \times \delta_{2,2} + s_2^3 \times \delta_{2,3}
\end{aligned}$$

Boucle L_1

```
for(i=0; i < n; i++) {
   a[i] = b[i] + c[i];
}
```

	S	**U(3)**	**SP**
a_1	8	7	5 (U = 3, min_it = 7)
b_1	0	$b_1^2(n)$	$b_1^3(n)$
s_1	8	20	64

Boucle L_2

```
// boucle externe
// for(j=0; j < m; j++) {
      for(i=0; i < n; i++) {
            a[j][i] = b[j][i]*s+val;
      }
// } fin de la boucle externe
```

	S	**U(3)**	**SP**
a_2	14	10	5 (U = 4, min_it = 10)
b_2	0	$b_2^2(n)$	$b_2^3(n)$
s_2	14	31	95

FIG. 3.4 – *Illustration du compromis taille/performance*

En supposant que les boucles ne sont parcourues que une ou quatre fois ($W \in \{1, 4\}$), la minimisation de la taille du code sous la contrainte d'une performance à atteindre est présentée dans le tableau 3.1. Le problème dual, la maximisation de la performance avec une contrainte de taille de code maximale, correspond au tableau 3.2. L'impact de la contrainte de taille apparaît clairement dans ce deuxième tableau : la contrainte se relâche progressivement de 50 instructions à 200 instructions. Les optimisations choisies s'orientent en conséquence vers des transformations qui impliquent une grosse expansion de code mais offrent des performances supérieures (de S vers SP puis de $U(3)$ vers SP).

		$n = 9$ $T \leq T_{max} = 400$				$n = 10$ $T \leq T_{max} = 400$				$n = 100$ $T \leq T_{max} = 4000$			
W_1	W_2	L_1	L_2	***S***	T	L_1	L_2	***S***	T	L_1	L_2	***S***	T
1	1	S	S	**22**	198	S	S	**22**	220	S	S	**22**	2200
4	1	U(3)	S	**34**	378	N/A (T_{max} trop faible)				U(3)	U(3)	**51**	3876
1	4	N/A (T_{max} trop faible)				N/A (T_{max} trop faible)				S	SP	**103**	3100

TAB. 3.1 – *Minimisation de la taille du code avec contrainte de temps d'exécution*

		$n = 100$ $S \leq S_{max} = 50$				$n = 100$ $S \leq S_{max} = 100$				$n = 100$ $S \leq S_{max} = 200$			
W_1	W_2	L_1	L_2	$\boldsymbol{T}$	S	L_1	L_2	$\boldsymbol{T}$	S	L_1	L_2	$\boldsymbol{T}$	S
1	1	S	U(3)	**1820**	39	SP	U(3)	**1684**	95	SP	SP	**1239**	159
4	1	S	U(3)	**4220**	39	SP	U(3)	**3676**	95	SP	SP	**3231**	159
1	4	S	U(3)	**4880**	39	SP	U(3)	**4744**	95	SP	SP	**2964**	159

TAB. 3.2 – *Minimisation du temps d'exécution avec contrainte de taille de code*

3.3 Expérimentation

3.3.1 Protocole expérimental

Le protocole expérimental est le suivant : les boucles critiques sont déterminées par *profiling* grâce à `tcov` qui réalise un test de couverture de l'application et ajoute au début de chaque ligne du programme source (en C) un nombre indiquant sa fréquence d'exécution (cf. figure 3.5). Six boucles ont été retenues pour H-263 et cinq pour mpeg2play. La mesure du temps réel passé dans ces boucles confirme qu'elles représentent un pourcentage significatif du temps total d'exécution du programme. Des compteurs sont alors placés dans les boucles de façon à déterminer leur *profile* d'exécution sous la forme d'un ensemble de couples (f, n) : la boucle a exécuté f fois n itérations. Pour mesurer ces valeurs, plusieurs fichiers de données sont utilisés pour chaque application pour tenter de capturer un comportement le plus représentatif possible. Le code de chaque boucle est extrait de l'application, placé dans un fichier distinct et traité indépendamment selon le principe présenté en figure 3.1.

```
  6704 → if (base.scalable_mode==SC_DP)
 ### → ld = &base;

  6704 → for (comp=0; comp<blk_cnt; comp++)
 40224 → clearblock(comp);

                    /* reset intra_dc predictors */
  6704 → dc_dct_pred[0]=dc_dct_pred[1]=dc_dct_pred[2]=0;
```

FIG. 3.5 – *Résultat de* `tcov`

3.3.2 Réalisation

La chaîne de compilation proprement dite est illustrée sur la figure 3.6. Dans cette étude nous nous intéressons à l'optimisation à bas niveau, c'est-à-dire la partie grisée de la figure. Grâce à l'infrastructure SEA, nous avons développé les transformations de programme assembleur utilisées pour cette étude. Les transformations de haut niveau

et la production d'assembleur sont accomplies par MT-1 [18], un générateur de code développé à l'université de Leiden, l'un de nos partenaires dans le projet européen Oceans, soit par le compilateur `tmcc` de Philips. Notre chaîne de compilation établit une connexion avec un serveur localisé aux Pays-Bas et envoie le code source. Après traitement, deux fichiers nous sont renvoyés : l'assembleur et un fichier qui permet l'interface entre les deux outils. Le contenu de ce fichier, que nous avons dénommé IL, est présenté en plus de détails dans le paragraphe 4.3.3. Il contient des informations faciles à obtenir à haut niveau, mais qui sont perdues pendant la traduction en assembleur. Il s'agit des dépendances entre les accès mémoires (les tableaux), de la structure des boucles et des espaces d'itérations.

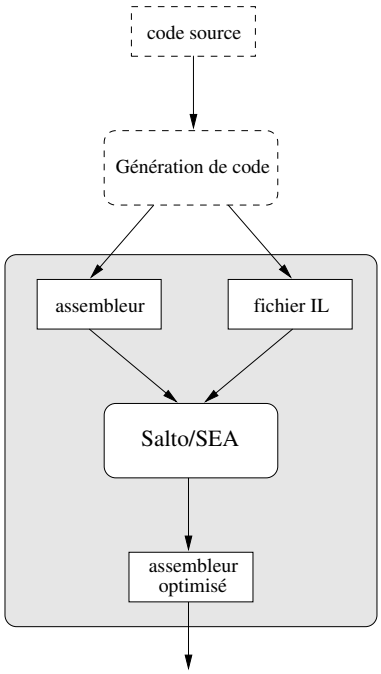


FIG. 3.6 – *Processus de compilation*

3.3.3 Optimisations retenues

Cette étude se concentre sur l'optimisation des boucles. L'ensemble des transformations que nous avons implémentées comprend le dépliage, la construction de superblocs, l'ordonnancement local des blocs de base et des superblocs, l'*inlining* et le pipeline logiciel. L'ordonnancement modulo est basé sur la méthode décrite dans [117]. L'allocation de registres peut-être accomplie soit avant, soit après l'ordonnancement. L'insertion de gardes permet de supprimer des instructions de saut et donc de construire des blocs plus grands qui présentent un parallélisme potentiel plus élevé.

Grâce à ces transformations, cinq séquences d'optimisations ont été appliquées à

	S_0	U_n	U'_n	SP	I
étape 1	ordo. local	dépliage $\times n$	dépliage $\times n$	Pipeline logiciel	*inlining*
étape 2	alloc. reg.	superbloc	superbloc		ordo. local
étape 3		ordo. local	alloc. reg.		reg. alloc.
étape 4		alloc. reg.	ordo. local		

FIG. 3.7 – *Séquences d'optimisation*

chaque boucle. Elle sont résumées en figure 3.7.

S_0 est la transformation la plus simple. Tout d'abord les registres sont renommés de façon à éliminer le maximum de fausses dépendances et faciliter la compaction du code. L'ordonnancement local est appliqué et enfin l'allocation de registres termine le travail.

U_n déplie la boucle n fois. Le nouveau corps de boucle est transformé en superbloc et les sauts conditionnels sont éliminés par insertion de gardes. De même que pour S_0, l'allocation de registres est appliquée après l'ordonnancement local. Le renommage n'est pas appliqué car les gardes ne permettent pas de déterminer avec précision la durée de vie des registres.

U'_n est similaire à U_n, mais l'allocation de registres est réalisée avant l'ordonnancement local. Ceci diminue les performances de l'ordonnanceur, mais a l'avantage de nécessiter moins de registres. Cette séquence a donc des chances de réussir quand U_n échoue à cause d'une pression sur les registres trop élevée.

SP transforme la boucle en pipeline logiciel. Cette séquence est actuellement applicable uniquement pour les boucles composées d'un seul bloc de base.

I remplace un appel de fonction dans une boucle par le corps de la fonction (*inlining*). L'ordonnancement du nouveau corps de boucle a lieu avant l'allocation de registres. L'*inlining* conduit à une expansion de la taille du corps de boucle qui offre davantage de possibilités à l'ordonnanceur. Il peut aussi permettre d'autres transformations, par exemple le pipeline logiciel.

3.3.4 Applications testées

Le processeur utilisé pour nos tests est le TriMedia de Philips. Il a pour vocation d'être un processeur enfoui et d'exécuter des applications «multimédia» de téléphonie sans fil, de télévision haute résolution, etc. Pour cette raison nous avons expérimenté notre approche avec les deux applications suivantes :

H-263 est un nouveau standard de compression vidéo. Nous avons utilisé l'implémentation `tmndecode` de Telenor R&D. Les informations de *profiling* proviennent de six jeux de données distincts.

mpeg2play est un décodeur MPEG 2. La partie visualisation a été désactivée pour éviter de prendre en compte le temps passé dans les bibliothèques X11. Les mesures sont réalisées avec cinq séquences vidéo différentes.

#	n	W	code C

1	$\{4, 8\}$	$\{1137984, 574912\}$	**for**(i=xa; i < xb; i++) {
			d[i] = s[i] ∗ om[i];
			}

2	$\{8\}$	$\{568768\}$	**for**(i=xa; i < xb; i++) {
			d[i] += s[i] ∗ om[i];
			}

3	$\{4, 8\}$	$\{188888, 92480\}$	**for**(i=xa; i < xb; i++) {
			dp[i] += (((**unsigned int**)(sp[i] + sp2[i]+1)) ≫ 1)∗om[i];
			}

4	$\{4, 8\}$	$\{167784, 82936\}$	**for** (i = xa; i < xb; i++) {
			dp[i] += (((**unsigned int**)(sp[i] + sp[i+1]+1)) ≫ 1)∗OM[c][j][i];
			}

5	$\{8\}$	$\{166400\}$	**for** (i = xa; i < xb; i++) {
			dp[i] += (((**unsigned int**)(sp[i]+sp2[i] +sp[i+1]+sp2[i+1]+2)) ≫ 2)∗om[i];
			}

6	$\{5\}$	$\{114196\}$	**for**(k=0; k < 5; k++) {
			xint[k] = nx[k] ≫ 1;
			xh[k] = nx[k] & 1;
			yint[k] = ny[k] ≫ 1;
			yh[k] = ny[k] & 1;
			s[k] = src + lx2∗(y+yint[k]) + x + xint[k];
			}

FIG. 3.8 – *Boucles critiques extraites de H-263*

3.4 Résultats

3.4.1 Application des séquences de transformations

Les figures 3.8 et 3.9 présentent les boucles sélectionnées pour subir une optimisation. La deuxième colonne indique les nombres d'itérations recensés. Certaines boucles ont un espace d'itérations constant de 4, 5 ou 8, alors que d'autres ont un espace qui varie au cours de l'exécution de l'application. Par exemple la boucle 5 de mpeg2play itère 1, 2 ou 3 fois. La troisième colonne donne la fréquence associée à chaque espace d'itération. La quatrième colonne contient le code source en C de la boucle. Ces boucles représentent en moyenne 40 % du temps d'exécution des applications.

Toutes ces boucles ont de petits espaces d'itérations : huit itérations au maximum. Ce fait accentue l'impact des surcoûts associés à certaines transformations de programme : dans le cas du dépliage, par exemple, le surcoût est un nombre entier d'itérations $(u - (n \bmod u)$ d'après l'équation 3.3). La plus petite valeur non nulle (1) représente déjà 12,5 % de l'espace d'itération total.

Nos expérimentations ne comprennent que cinq ou six boucles, nous avons toutefois vérifié les capacités de notre *simplexe* avec des systèmes aléatoires de 200 équations. Le

#	n	W	code C

1	{8}	{363258}	**for**(i=0; i < 8; i++) {
			  idctrow(block+8*i);
			}

2	{8}	{363258}	**for**(i=0; i < 8; i++) {
			  idctcol(block+i);
			}

3	{8}	{211668}	**for**(i=0; i < 8; i++) {
			  rfp[0] = clp2[bp[0]];
			  ...
			  rfp[7] = clp2[bp[7]];
			  rfp += iincr;
			  bp += 8;
			}

4	{8}	{151590}	**for**(i=0; i < 8; i++) {
			  rfp[0] = clp2[bp[0] + rfp[0]];
			  ...
			  rfp[7] = clp2[bp[7] + rfp[7]];
			  rfp += iincr;
			  bp += 8;
			}

5	{1, 2, 3}	{977286, 83741, 311}	**do** {
			  bfr |= *ld → ptr++ ≪ (24-incnt);
			  incnt += 8;
			} **while** (incnt ≤ 24);

FIG. 3.9 – *Boucles critiques extraites de mpeg2play*

temps de calcul sur Ultrasparc est en moyenne de quelques minutes, avec un maximum de une heure, ce qui reste raisonnable dans le contexte des systèmes enfouis. Notons que si le temps de calcul devient excessif, il est possible d'utiliser d'autres outils [48] et de nous en tenir à une solution approchée.

Notre système a tenté d'appliquer chaque séquence d'optimisation à chaque fragment de code. Dans tous les cas où cela fut possible, la vitesse asymptotique de la boucle, le surcoût et la taille sont mesurées. Les résultats sont présentés dans les tableaux 3.3 et 3.4. Un tiret indique que la transformation n'est pas valide.

Pour obtenir davantage de gain en performance, nous avons voulu de déplier les boucles 1 et 2 de mpeg2play après *inlining*. Cette tentative s'est malheureusement soldée par un échec pour U_1 dû à un nombre insuffisant de registres. Il en est de même pour le pipeline logiciel.

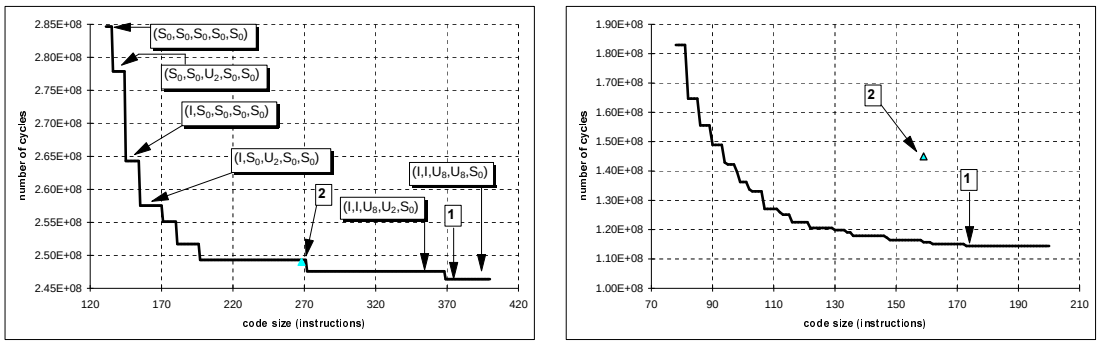
Dans tous les cas où le pipeline logiciel s'applique, un faible nombre de cycles par itérations est atteint. Toutefois, à l'exception de la boucle 2 de H-263, le nombre minimum d'itérations requis (min_{it}) se trouve être supérieur au nombre d'itérations déterminés par *profiling*. En conséquence le pipeline logiciel n'est pas exécuté, et seul le code de compensation est utilisé, d'où un gaspillage de place et aucun gain de performance.

	S_0	U_2	U_3	U_4	U_2'	SP
a_1	8	6	5	5	7	3 (U = 6, min_{it} = 16)
b_1	0	0	{10,5}	0	0	{32,64}
s_1	8	12	16	20	13	75
a_2	9	7	6	6	10	5 (U=3, min_{it} = 6)
b_2	0	0	6	0	0	22
s_2	9	13	18	22	19	55
a_3	12	8	8	7	12	6 (U=6, min_{it} = 12)
b_3	0	0	{16,8}	0	0	{48,96}
s_3	12	16	24	28	24	121
a_4	15	10	9	9	16	6 (U=9, min_{it} = 18)
b_4	0	0	{18,9}	0	0	{60, 120}
s_4	15	20	28	34	31	172
a_5	15	10	10	8	17	7 (U=8, min_{it} = 16)
b_5	0	0	10	0	0	120
s_5	15	19	29	33	33	179
a_6	19	13	12	11	30	nombre
b_6	0	13	12	33	30	de registres
s_6	19	25	36	44	59	insuffisant

TAB. 3.3 – *Résultats des transformations de code pour H-263*

3.4.2 Analyse

Les données récoltées permettent de construire le système de l'équation 3.7 page 77. En faisant varier la contrainte de taille et en calculant la performance optimale pour chaque valeur nous obtenons les courbes de la figure 3.10. Comme prévu, la performance est une fonction décroissante de la taille allouée au code. À noter que le nombre de paliers est déjà conséquent pour une étude qui ne porte que sur cinq ou six boucles : si la détermination de la configuration idéale peut encore être faite «à la main» par un expert dans ce cas, il est clair qu'il est irréaliste d'envisager d'y parvenir quand plusieurs dizaines de fragments de code sont en jeu. L'assistance d'un outil devient primordiale.



mpeg2play H-263

FIG. 3.10 – *Performance optimale en fonction de la taille de code*

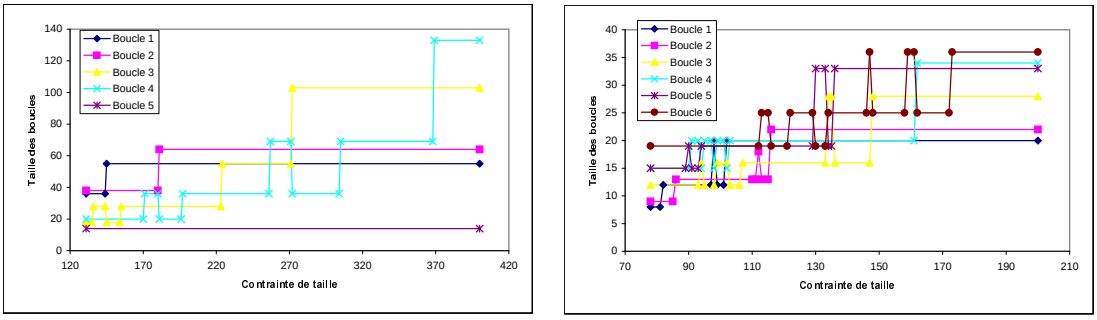
	S_0	U_2	U_3	U_4	U_8	U_2'	SP	I
a_1	36					42	-	29
b_1	0	échec	échec	échec	échec	0	-	0
s_1	36					83	-	55
a_2	38					48	-	36
b_2	0	échec	échec	échec	échec	0	-	0
s_2	38					95	-	64
a_3	18	14	13	14	13	67	13 (U=3,min_{it}=9)	-
b_3	0	0	13	0	0	0	48	-
s_3	18	28	39	55	103	134	124	-
a_4	20	18	18	18	17	69	16 (U=3,min_{it}=9)	-
b_4	0	0	18	0	0	0	54	-
s_4	20	36	53	69	133	138	150	-
a_5	14	11	10	9	9	24	5 (U=4,min_{it}=16)	-
b_5	0	{11,0,11 }	{20,10,0}	{27,18,9}	{63,54,45}	0	{37,46,55 }	-
s_5	14	21	29	36	65	48	85	

TAB. 3.4 – *Résultats des transformations de code pour mpeg2play*

L'étude de l'interaction entre les optimisations des différents fragments de code quand la contrainte de taille diminue révèle un comportement complexe. Chaque courbe de la figure 3.11 illustre la taille occupée par une boucle en fonction de la taille globale allouée. Lorsque la taille n'est pas limitée, les boucles sont déroulées avec le facteur maximal, les fonctions sont *inlinées*. Seule la boucle 5 de mpeg2play semble ne subir aucun traitement. Ceci s'explique simplement par le fait que le corps est, dans la majorité des cas, exécuté seulement une fois (cf. figure 3.9). Quand la taille diminue en dessous d'un certain seuil, la solution initiale n'est plus acceptable et l'une (au moins) des boucles doit être dépliée avec un facteur moindre. Dans le cas de mpeg2play, ce phénomène apparaît à $S = 368$: la boucle 4 passe de l'optimisation $U_1(8)$ à $U_1(4)$. En dessous de $S = 304$, c'est à nouveau la place allouée à la boucle 4 qui est entamée, et le meilleur choix consiste à déplier la boucle deux fois. C'est à $S = 271$ que se produit un événement intéressant : la boucle 3 passe $U_1(8)$ à $U_1(4)$, ce qui permet à la boucle 4 de revenir à l'optimisation précédente. Les choix d'optimisations correspondant à quelques points de la courbe sont explicités sur la figure 3.10. Ce processus par lequel les fragments de code s'échangent la place disponible est particulièrement complexe, comme le montrent les deux figures. Certains choix, judicieux pour une contrainte donnée, doivent être sans cesse remis en cause. Ceci met en lumière la nécessité de considérer le processus d'optimisation globalement.

Comparons notre approche globale avec des techniques plus classiques qui n'ont qu'une vision locale du code.

1. la première choisit localement (c'est-à-dire pour chaque fragment) la séquence d'optimisations qui conduit à la meilleure performance pour ce fragment particulier. Cette approche est équivalente à GCDS avec une taille de code illimitée ;
2. la seconde technique sélectionne systématiquement l'optimisation qui donne la



mpeg2play H-263

FIG. 3.11 – *Évolution des tailles des boucles*

vitesse asymptotique la plus élevée, c'est-à-dire celle qui donne la plus petite valeur de a_i. Aucune information de *profiling* ne guide le choix, toutefois les informations disponibles statiquement dans le programme sont exploitées, par exemple des bornes de boucles constantes. Pour limiter l'explosion de la taille du code, nous supposons que le compilateur cesse de déplier une boucle s'il n'obtient pas un gain supérieur à 10 %.

Comme la première technique ne prend pas la taille en considération, elle produit le code le plus efficace en dépliant toutes les boucles et en *inlinant* toutes les fonctions. Le code correspondant est représenté par le point 1 sur la figure 3.10. Cette approche a le gros défaut de gaspiller beaucoup de place, comparée à GCDS. Par exemple en acceptant une augmentation de 1 % du nombre de cycles pour mpeg2play, il est possible de diviser par deux la taille du code (de 400 mots à 200 mots). De même, un accroissement de 3 % du temps d'exécution de H-263 conduit à un gain de 21 % en place. Ce phénomène serait exacerbé en présence de plusieurs dizaines de fragments.

La seconde technique n'utilise pas d'information de *profiling*, ce qui lui fait défaut simultanément pour la performance et la taille du code. La boucle 5 de mpeg2play est caractéristique : la meilleure optimisation est visiblement le pipeline logiciel, avec une vitesse asymptotique $a_5 = 5$. Malheureusement la valeur seuil $min_{it} = 16$ n'est pas atteinte une seule fois pendant l'exécution de l'application. Un espace important est occupé pour un gain de performance nul. Il aurait pu être exploité beaucoup plus efficacement pour déplier la boucle. Un code relativement efficace est pourtant produit pour mpeg2play. Par contre la même performance peut être atteinte avec un code 26 % plus petit. Dans le cas de H-263, le code obtenu pêche par sa taille autant que par son manque d'efficacité. Les points sont représentés sur la figure par le chiffre 2.

3.4.3 Validation

Dans la mesure où notre approche s'appuie sur le *profiling* et qu'elle nécessite des informations précises, il est essentiel de vérifier la validité de ces informations quand d'autres jeux de données sont utilisés.

Nous avons tout d'abord vérifié que les boucles qui consomment le plus de temps ne changent pas en fonction des jeux de données. Ensuite il reste à mesurer l'efficacité de notre optimisation avec des données différentes de celles utilisées pour construire le *profile*. Nous avons procédé comme suit : un jeu de données — différent de ceux utilisés précédemment — est utilisé pour déterminer une séquence d'optimisation. Le nombre de cycles est mesuré. Nous mesurons alors le nombre de cycles exécutés pour ce jeu de données, mais pour l'application optimisée par GCDS. Notre métrique consiste en le rapport de ces deux valeurs. Pour chaque valeur de la contrainte de taille, nous avons calculé cette métrique pour sept séquences vidéo mpeg2play et six H-263. Dans la majorité des cas, l'application optimisée par GCDS atteint encore la performance optimale. La différence en nombre de cycles n'excède jamais 1,3 % pour mpeg2play et 6 % pour H-263.

3.5 Conclusion

Comparée aux deux techniques précédentes, GCDS souligne l'importance de la prise de décision globale, ainsi que de la précision du *profiling*. Les stratégies locales ne permettent pas de faire de compromis.

Si GCDS est illustré dans ce chapitre par la recherche d'un compromis entre la taille d'un code et la performance atteinte, son domaine d'application est beaucoup plus vaste : les grandeurs utilisées comme contrainte et fonction objective peuvent être modifiées. Il est par exemple possible, avec le modèle *ad hoc* de déterminer un compromis entre la performance et la puissance consommée. Le *simplexe* est un moteur bien adapté à notre modèle, constitué d'équations linéaires. Il peut aussi être remplacé par tout autre algorithme linéaire, non-linéaire, voire approximatif. Enfin GCDS n'est pas limité aux optimisations de boucles à bas niveau. Le chapitre 4 montre comment il s'intègre dans une chaîne de compilation pour prendre des décisions à haut niveau.

Chapitre 4

Une approche itérative pour l'optimisation de code

Un compilateur dispose d'un certain nombre de transformations de code, paramétrables ou non, qu'il applique pour traduire une application écrite en langage de haut niveau en un programme exécutable. La difficulté majeure de conception consiste à définir une stratégie qui spécifie quelles transformations appliquer, dans quel ordre et avec quels paramètres.

Il n'existe pas de séquence d'optimisations qui produit systématiquement un code aux performances optimales. L'intégration de transformations donne un gain certain, mais ne résout pas le problème. L'enchaînement idéal des transformations dépend du programme et de l'architecture matérielle considérés. Le compilateur est donc amené à considérer un vaste ensemble de combinaisons de transformations et doit déterminer laquelle va probablement aboutir à de bonnes performances. La définition de la stratégie d'un compilateur est un problème difficile qui a été peu abordé dans la littérature.

L'espace des optimisations est vaste, et ne peut être parcouru de manière systématique. Nous avons développé une approche itérative qui évalue successivement plusieurs séquences de transformations pour appréhender la «forme» de cet espace et localiser des régions où la performance est élevée. Dans la structure classique d'un compilateur : restructuration à haut niveau, génération de code et optimisations à bas niveau (cf. 1.1), les modules qui travaillent sur le code assembleur ne peuvent remettre en cause les décisions prises avant la génération de code. Notre approche permet cette rétroaction. Elle requiert un support pour les flots d'information nécessaires aux communications entre les différents modules : haut et bas niveau. L'exploration de multiples séquences se fait au détriment du temps de compilation, mais reste toutefois raisonnable pour les systèmes enfouis.

Nous présentons ici une première approche d'un vaste domaine de recherche. Nous avons validé l'approche itérative à l'aide de quelques exemples : la stratégie GCDS présentée au chapitre précédent s'intègre naturellement dans ce cadre ; notre deuxième exemple prend en compte les discontinuités de l'espace liées aux paramètres entiers des transformations de code ; dans le dernier exemple nous capturons les interactions

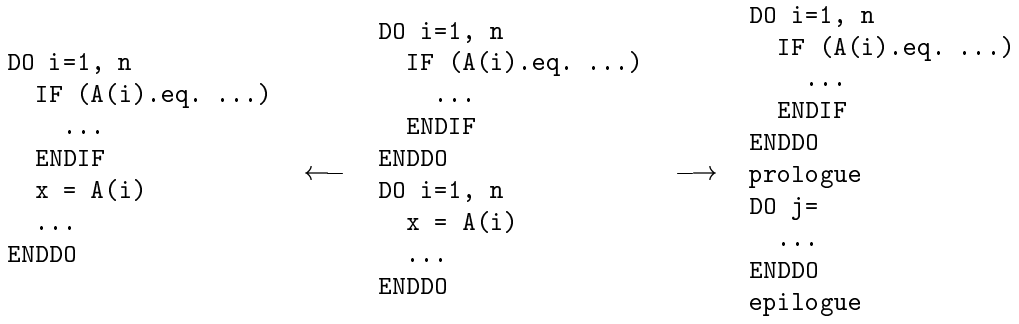
profondes entre deux optimisations.

Dans la première section de ce chapitre nous présentons les difficultés inhérentes à l'exploration de l'espace des optimisations. Nous décrivons ensuite notre approche du problème puis l'infrastructure que nous avons développée pour expérimenter diverses stratégies de compilation. La quatrième section illustre le fonctionnement de notre système à l'aide de trois exemples. La cinquième et dernière partie conclut.

4.1 Un problème complexe

Les séquences d'optimisations ne doivent plus être déterminées statiquement au moment de la conception du compilateur, mais au moment de la compilation, c'est-à-dire pendant l'exécution du compilateur. Une vaste quantité d'informations sur le programme, les optimisations disponibles et l'architecture cible doivent être intégrées de façon à obtenir une séquence de transformations efficace, adaptée à chaque cas particulier. Il est irréaliste d'envisager d'explorer toutes les combinaisons tant leur nombre est élevé. À l'opposé, les stratégies monolithiques des compilateurs ont montré leurs limites dans la recherche de performance. Il va donc être nécessaire d'explorer une partie de l'espace pour appréhender sa «forme» et se diriger vers régions les plus intéressantes.

La première étape consiste à déterminer dans le programme des fragments de code qui vont être optimisés indépendamment. Ce choix initial a des conséquences sur la suite des optimisations. Considérons le code de la figure 4.1 (b) : s'il est considéré dans son ensemble, il sera possible de fusionner les deux boucles pour obtenir la version 1 et améliorer la localité des données. Si par contre les deux boucles sont traitées séparément, la seconde pourra être transformée par pipeline logiciel. La fusion des deux boucles devient impossible. La première version contient maintenant une conditionnelle qui empêche le pipeline logiciel. Les deux versions évoluent sur des chemins distincts.



(a) Version 1 (b) Code original (c) Version 2

FIG. 4.1 – *Choix de fragments de code indépendants*

Une fois ce choix de décomposition fait, l'espace des séquences d'optimisations applicables à chaque fragment est extrêmement vaste, et ce dans plusieurs directions. Tout

d'abord le nombre de transformations de code connues applicables à un programme est conséquent. Une optimisation requiert généralement l'enchaînement de plusieurs transformations à haut niveau puis à bas niveau, ce qui fait croître le nombre de séquences envisageables de façon exponentielle.

Un grand nombre de ces transformations dépendent d'un ou plusieurs paramètres. Le dépliage, par exemple, en utilise un, le blocage en utilise plusieurs en fonction de la profondeur du nid de boucles. L'impact de la transformation de code dépend fortement de la valeur du paramètre, comme nous le montrons dans ce chapitre. Les variations de ces paramètres augmentent d'autant la taille de l'espace des optimisations.

Ensuite les transformations applicables à un instant dépendent du succès des précédentes. Si l'*inlining* a pu être appliqué, un appel de fonction a été supprimé, ce qui peut permettre d'appliquer de nouvelles transformations qui n'étaient pas légales auparavant. Si le code ne contient pas d'appel de fonction, l'*inlining* n'a pas lieu d'être, mais ce fait n'est pas connu au moment de la conception du compilateur. L'espace des optimisations évolue ainsi dynamiquement, ce qui le rend potentiellement infini.

4.2 Une approche itérative

La mise en œuvre de techniques plus agressives que celles employées dans les compilateurs classiques n'est possible que si une contrainte est supprimée. Nous faisons l'hypothèse que le temps de compilation peut être beaucoup plus long, ce qui est réaliste dans le contexte des systèmes enfouis.

Notre approche est fondée sur le fait qu'il impossible de quantifier *a priori* les interactions de plusieurs transformations de code. En utilisant le fait que le temps de compilation n'est plus une contrainte, nous proposons une approche itérative fondée sur la recherche d'un minimum par approximations successives : l'algorithme applique successivement des transformations de code, à haut et bas niveau et est guidé par les informations que lui fournissent les différentes transformations au fur et à mesure de leur application. La figure 4.2 schématise l'organisation de notre système. Il se décompose en trois parties :

1. le moteur, chargé de la mise en œuvre de la stratégie. Il contrôle l'exécution des optimisations et prend des décisions en fonctions des informations qu'il reçoit ;
2. les transformations appliquent les optimisations décidées par le moteur ;
3. la rétroaction est constituée d'informations fournies par les transformations sur leur déroulement (succès, échec) et sur les propriétés du code qu'elles ont produit (taille, registres utilisés, chemin critique, etc.).

4.2.1 Parcours de l'espace d'optimisation

La tâche du moteur de l'algorithme consiste à définir un espace des transformations de code applicables au fragment donné et à le parcourir de façon efficace pour trouver le minimum, ou à défaut un point qui donne de bonnes performances.

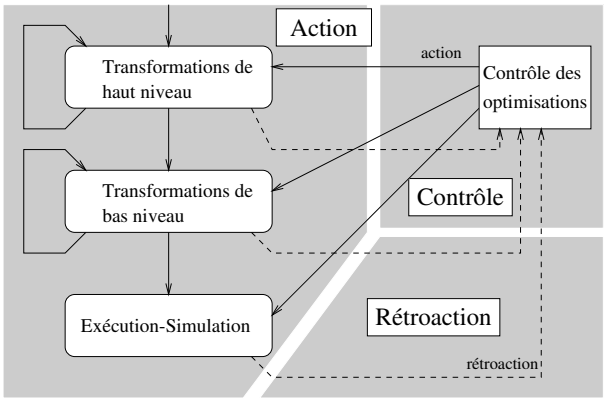


FIG. 4.2 – *Approche itérative de la compilation*

Dans un souci d'homogénéité, nous appelons ici «transformation» toutes les opérations effectuées sur le programme, y compris la génération de code et l'exécution. L'exécution ne produit pas de nouveau programme, mais des résultats tels que le temps d'exécution, le nombre de défauts de cache, etc. Elle est aussi couverte par le terme «transformation».

L'espace des transformations est représenté sous la forme d'un arbre (cf. figure 4.3) dans lequel les arcs représentent les transformations et les nœuds les fragments de code. Comme nous l'avons vu, il est construit dynamiquement. En effet :

- les «fils» d'un nœud ne peuvent pas toujours être déterminés statiquement. Par exemple la distribution d'une boucle n'est pas toujours possible, les transformations suivantes sont conditionnées par sa légalité car la structure même du code en dépend (présence d'une ou de deux boucles) ;
- l'évaluation de «frères» ou «sœurs» d'un nœud peut devenir inutile au vu des informations déjà obtenues. Le dépliage d'une boucle améliore généralement les performances jusqu'à un certain point. Au delà, elles recommencent à chuter. Dès que ce point a été atteint, il est inutile de continuer à déplier la boucle. Nous revenons sur ce phénomène plus loin dans ce chapitre.

Partant d'un arbre initial qui correspond à une première ébauche du travail à accomplir, le compilateur va le raffiner par ajouts et suppressions successifs jusqu'à l'avoir parcouru entièrement. La meilleure version du programme produite est alors fournie comme résultat.

L'essentiel de travail repose sur la définition d'une «stratégie» qui va permettre d'utiliser les résultats des transformations passées pour en proposer de nouvelles.

4.2.2 Flots d'information

Le succès de notre approche repose sur une communication efficace entre les composants du système. Chaque transformation produit des informations sur le code qu'elle a généré soit à destination de l'algorithme qui l'utilise comme rétroaction pour essayer

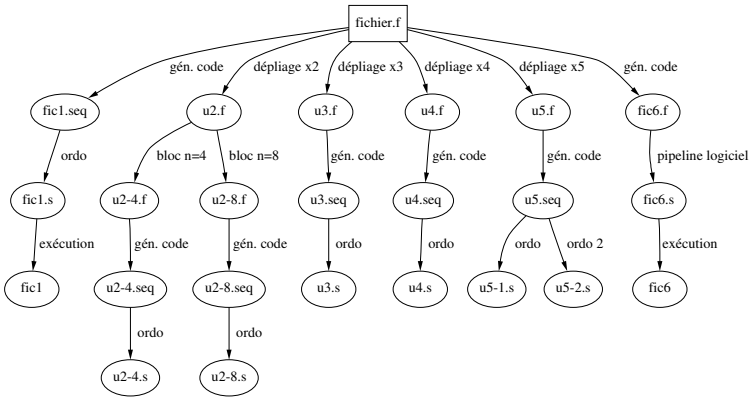


FIG. 4.3 – *Espace d'optimisations*

une nouvelle direction, soit directement à destination d'autres transformations pour leur permettre d'être plus efficace.

Les trois niveaux d'information sont le langage de haut niveau (C, C++, Fortran), le bas niveau (assembleur) et l'information dynamique collectée pendant l'exécution. Les flots d'informations utiles entre les différents niveaux sont illustrés sur la figure 4.4.

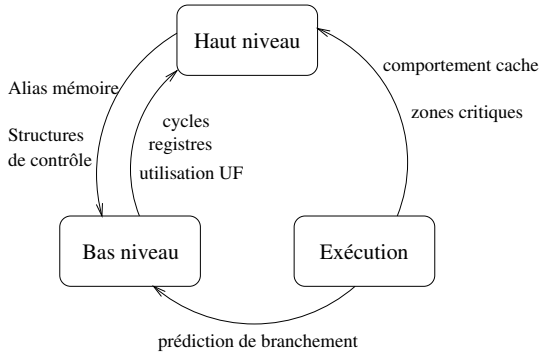


FIG. 4.4 – *Flots d'information entre les différents niveaux*

4.2.2.1 Haut vers bas

Le haut niveau correspond au code source, écrit par le programmeur, généralement en C/C++ ou Fortran. C'est à ce niveau que l'algorithme du programmeur et les structures de données qu'il emploie sont lisibles. Les transformations de programme faites à ce niveau vont affecter le parallélisme à gros grain.

Les principales structures de contrôle sont les boucles ou les nids de boucles et les conditionnelles. Les structures de données choisies par un programmeur pour exprimer

son algorithme ne sont reconnaissables qu'à haut niveau. Au niveau assembleur ne subsistent que des accès à la mémoire, en lecture ou en écriture.

Boucles Les boucles constituent une cible de grand intérêt pour les optimisations puisqu'elle concentrent la majorité du temps d'exécution des applications et un très grand nombre d'optimisations existent. Les boucles sont facilement détectables à haut niveau (`for() { ... }` en C, `DO ... ENDDO` en Fortran) et de nombreux outils commencent par identifier la structure de l'application sur laquelle ils vont travailler [24, 19]. Cette structure est bien moins facile à construire à partir de l'assembleur. Cela reste bien sûr possible: un algorithme bien connu est donné dans [3], même si dans le cas du TriMedia, qui autorise des branchements gardés, le problème est plus délicat. Transmettre la description des boucles permet d'éviter de refaire un travail déjà fait. Des informations comme le nom du registre utilisé comme index, les bornes (quand elles sont connues) peuvent aussi être exploitées par des phases d'optimisation ultérieures.

Conditionnelles Les conditionnelles ont aussi leur impact : elles introduisent des branchements. Deux grands types d'optimisations s'y intéressent : la prédiction de branchement et la suppression des branchements par insertion de gardes. La connaissance du haut niveau donne des indications sur le comportement probable du branchement. Si l'on considère l'exemple de la figure 4.5, le branchement `bne .LL4` a de fortes chances d'être pris, puisque les pointeurs «utiles» sont rarement nuls. Ball et Larus ont défini un ensemble d'heuristiques de ce genre qui permettent d'améliorer la précision de la prédiction de branchements statique [8].

```
for(i=0; i < n; i++) {
  if (ptr == NULL) {
    fprintf(stderr, "Fatal error\n");
    exit(1);
  }
  ...
}
```

```
.LL5:
        ld [%fp-28],%o0
        cmp %o0,0
        bne .LL4
        nop
        ...
        call fprintf,0
        nop
        mov 1,%o0
        call exit,0
        nop
.LL4:
```

FIG. 4.5 – *Prédiction de branchement «facile»*

Tableaux Il est difficile de savoir, en regardant le code assembleur de la figure 4.6, si les adresses des `ld` et `st` sont distincts. Des algorithmes d'analyse d'alias permettent parfois de lever l'ambiguïté, mais celle ci est parfois inhérente au langage de haut niveau utilisé: en C les tableaux passés en paramètres d'une fonction peuvent se recouvrir, ce qui est interdit en Fortran. Il est donc préférable de propager cette

information quand elle est connue.

```
void f(int n)
{
  int i, n, a[MAX], b[MAX];

  ...
  for(i=0; i < n; i++) {
    a[i] = a[i+1] + b[i];
  }
  ...
}
```

```
.LL5:
        sll %o1,2,%o0
        add %o1,1,%o1
        sll %o1,2,%g2
        ld [%o2+%g2],%g2
        ld [%o3+%o0],%g3
        add %g2,%g3,%g2
        cmp %o1,%o4
        bl .LL5
        st %g2,[%o2+%o0]
.LL3:
```

FIG. 4.6 – *Accès aux éléments d'un tableau*

En conséquence il est intéressant de mémoriser ces informations au moment où elles sont disponibles et de les transmettre au besoin pour éviter un travail inutile.

4.2.2.2 Bas vers haut

À bas niveau les transformations de programme ont une vision fine de l'architecture du processeur. L'exploitation des ressources peut être quantifiée et des problèmes engendrés par les transformations précédentes apparaissent.

Nombre de cycles statique Le temps d'exécution d'une boucle peut être estimé par le nombre de cycles du corps. Seul manque le comportement de la hiérarchie mémoire pour obtenir le temps réel. Certaines transformations ne modifient pas l'ordre des accès aux données (le dépliage par exemple) et dans ce cas le nombre de cycles statiques donne une indication sur l'évolution des performances.

Pression de registres Lorsque le nombre de valeurs vivantes en un point du programme dépasse le nombre de registres disponibles, l'allocateur de registres est contraint d'insérer du *spill code*, c'est-à-dire des accès à la mémoire qui conduit à un ralentissement. Cela peut être le signal que les transformations de haut niveau ont produit un code trop long (par exemple un facteur de dépliage trop élevé) et que le gain obtenu à haut niveau risque d'être perdu à bas niveau.

Utilisation des unités fonctionnelles Un faible taux d'utilisation des unités fonctionnelles devrait inciter à appliquer des transformations connues pour augmenter le parallélisme d'instructions. Au contraire, si une unité est saturée, il est inutile de continuer à augmenter la taille des blocs pour extraire du parallélisme.

4.2.2.3 Exécution vers haut et bas niveau

Les informations de *profiling*, collectées pendant l'exécution, peuvent guider le placement du code à bas niveau, ou améliorer la prédiction des branchements en changeant

la direction des sauts ou en indiquant leur destination probable. Les traces d'exécution permettent aussi de construire des superblocs. Si des motifs répétitifs d'accès aux données apparaissent, il peut être intéressant d'utiliser du pré-chargement.

Les informations dynamiques sont toutefois sujettes à caution. Il est essentiel de vérifier la validité de ce type d'information. En effet le *profile* caractérise la comportement du programme pour un jeu de données. Il n'est pas certain que le comportement sera le même pour d'autres jeux de données. En d'autres termes, le comportement du programme est-il dépendant de son entrée ? Si oui, peut-on déterminer un ensemble fini, voire relativement petit, de jeux de données qui caractérisent tous les comportements possibles du programme avec une bonne précision.

La première mesure de ce type a été faite par Wall [116]. Fisher et Freudenberger [47] se sont aussi intéressés au problème pour la prédiction des branchements. La définition d'un *profile* est simplement un tableau à deux colonnes, la première contenant la liste des objets auxquels on s'intéresse et la deuxième des nombres (les «poids») correspondant à une mesure des objets de la première colonne. Les lignes du tableau sont triées par ordre décroissant de la deuxième colonne. Nous pouvons par exemple compter combien de fois chaque procédure d'un programme est appelée : la première colonne contient le nom des procédures et la deuxième le nombre d'appels.

Wall définit deux méthodes pour mesurer la validité d'un *profile* pour une exécution différente :

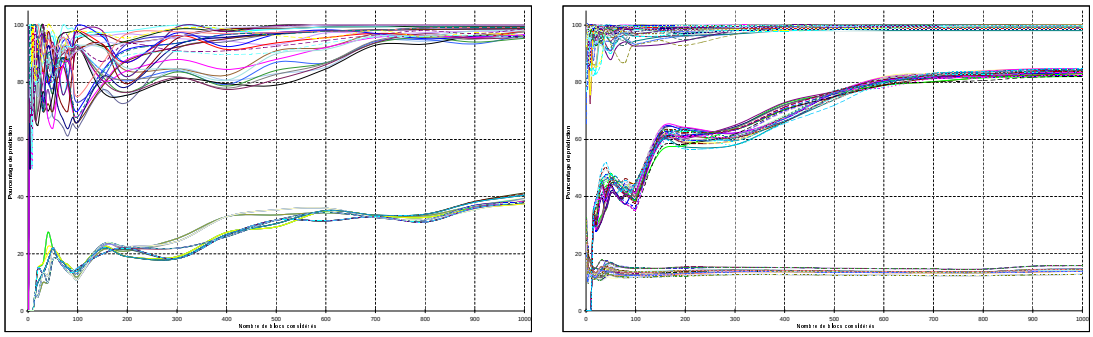
key matching : il considère les n premières entrées du *profile* réel et du *profile* de référence et compte le nombre p d'entrées communes. Le taux de validité est définie comme p/n.

weight matching : il considère les n premières entrées du *profile* de référence et calcule la somme de leur poids dans le *profil réel*. Il divise ce nombre par la somme des poids des n premières entrées du *profile* réel.

Notons que seule la comparaison *key matching* est réflexive.

Nous avons appliqué ces deux méthodes au comptage de la fréquence des blocs de base de l'application H-263. La figure 4.7 illustre les résultats. Nous disposons de 12 jeux de données. Chaque courbe représente la validité d'un *profile* pour une exécution avec un autre jeu de données (soit $C_{12}^2 = 66$ courbes pour *key matching* et $12 \times 11 = 132$ pour *weight matching*) en fonction du nombre de blocs considérés. H-263 comporte 2550 blocs. Les mesures *key matching* font clairement apparaître deux comportements distincts qui correspondent à deux familles de jeux de données. Chaque élément d'une famille prédit relativement bien les autres membres de sa famille, mais assez mal l'autre famille. Ces deux familles correspondent à deux algorithmes de compression des flux vidéo. Les mesures *weight matching* laissent penser que trois familles existent. En fait ce phénomène est lié à la relation de comparaison qui n'est pas réflexive : l'algorithme le plus complexe utilise des routines du plus simple, alors que l'inverse n'est pas vrai. Le premier parvient donc à prédire relativement bien le second.

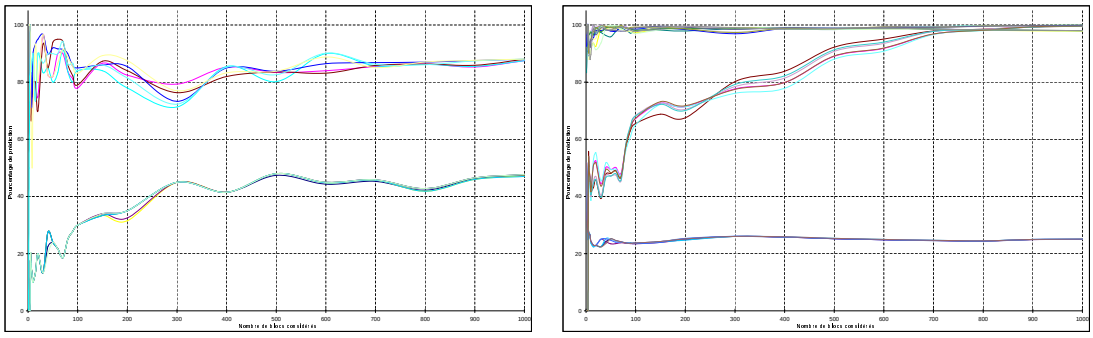
Nous avons défini un *profile* moyen en calculant la moyenne des poids de chaque bloc dans tous les *profiles*. Pour déterminer sa validité nous avons répété les expériences en comparant chaque jeu de données à cette nouvelle référence. Les résultats sont illustrés



Key matching *Weight matching*

FIG. 4.7 – *Validité des* profiles *pour H-263*

sur la figure 4.8.



Key matching *Weight matching*

FIG. 4.8 – *Validité du* profile *moyen pour H-263*

Il apparaît que même un *profile* moyen ne capture pas efficacement l'ensemble des comportements individuels. Il améliore légèrement la prédiction dans le cas le plus défavorable, au prix d'une dégradation du cas favorable pour *key matching*. La précision de la prédiction dépend ainsi de l'application, du (ou des) jeu(x) de données de référence, mais aussi de la métrique employée. L'utilisation de *profile* pour guider une optimisation doit donc faire l'objet de précautions et de vérification.

4.3 Un prototype de compilateur

Nous avons implémenté un prototype pour valider notre approche. La modularité de l'infrastructure SEA (cf. paragraphe 2.2) a fait ses preuves et nous avons décidé de conserver cette conception. La distinction entre contrôle, action et rétroaction du modèle a été conservée au niveau du codage de façon à faciliter la maintenance et l'évolution du système. L'ensemble a été écrit en C++. Notre prototype fonctionne et

a été utilisé pour mener les expériences décrites dans ce chapitre.

Nous décrivons d'abord l'implémentation de notre système de compilation itérative, puis des transformations de code. Nous présentons ensuite le langage que nous avons proposé pour véhiculer l'information nécessaire entre les différents modules du système. Dans la dernière partie nous insistons sur l'aspect distribué du système.

4.3.1 Moteur de recherche

Notre implémentation repose sur le parcours d'un arbre de recherche identique à celui de la figure 4.3. Les transformations sont des objets qui héritent tous d'un comportement minimal, de la même façon que dans SEA. Le terme «transformation» est toujours pris dans une acceptation très large: la génération de code est considérée comme une transformation qui lit un fichier dans un langage de haut niveau et produit de l'assembleur, de même que l'assemblage lit du code assembleur et produit un exécutable. L'exécution d'un programme lit un exécutable et ne produit rien. Le type des fichiers en entrée et sortie d'une transformation permet de déterminer si les transformations peuvent être cascadées ou non. Cette uniformisation permet au moteur de recherche de manipuler les transformations de façon plus simple.

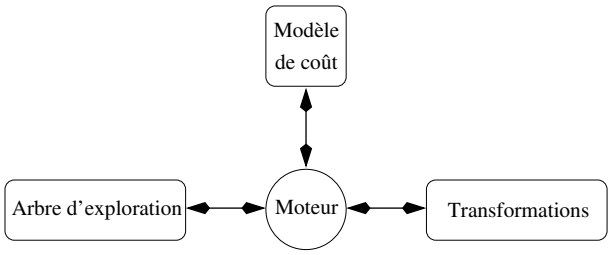


FIG. 4.9 – *Implémentation du moteur*

L'arbre de recherche est parcouru en profondeur d'abord et chaque transformation est appliquée. Si l'une échoue, le reste de la branche est ignoré et la branche suivante est explorée. Toutes les versions du programme à optimiser sont simplement conservées dans des fichiers, dont l'algorithme garde le nom. Lorsque la meilleure version est déterminée, les versions inutiles peuvent être détruites.

4.3.2 Transformations

Chaque transformation conserve des informations sur le code qu'elle a produit. Celles-ci peuvent être consultées à tout instant par une autre transformation ou comme rétroaction par le moteur du système. L'ensemble de l'information disponible sur le programme en cours d'optimisation est ainsi répartie sur tous les nœuds de l'arbre. Aucune distinction n'est faite entre les informations statiques et dynamiques, si ce n'est pour des raisons de performance.

L'implémentation de chaque transformation est totalement indépendante du reste du système. Ceci permet soit une approche directe, soit l'utilisation d'outils existants.

Nous avons notamment tiré parti des possibilités de restructuration de haut niveau de MT-1 et des langages TDL et SSL (cf. [18]). Le paragraphe 4.3.4 montre comment cette modularité nous a permis de distribuer géographiquement notre système.

Certaines transformations ne peuvent pas être évaluées immédiatement. L'impact d'un dépliage, par exemple, n'est connu qu'une fois le code assembleur ordonnancé. Nous avons donc mis en place un mécanisme qui permet à une transformation de déclencher une action quand une information devient disponible. Dans le cas de notre exemple, l'objet «dépliage» demande le déclenchement d'une de ses méthodes quand la taille du code est connue en plaçant son adresse, le nom de la méthode à invoquer et l'information demandée dans une table. Chaque transformation vérifie si l'une des informations qu'elle produit a été demandée auparavant. Si c'est le cas, elle déclenche directement l'appel. La méthode reçoit en paramètre l'information qu'elle a demandée. Le dépliage, au vu des performances atteintes, peut décider qu'il est inutile de continuer et modifier en conséquence l'arbre de recherches.

4.3.3 IL : vecteur d'information

Nous avons vu la nécessité de propager de l'information entre les différentes étapes de compilation, et particulièrement entre le haut niveau et le bas niveau.

IL est le nom d'un petit langage conçu pour les besoins de notre chaîne de compilation. IL signifie aussi bien *interface language* que *intermediate language*. Son but est de véhiculer de l'information entre les différentes étapes de la compilation. Il permet, de façon générale, de nommer des fragments de code et de leur attribuer des propriétés. Après une description du langage, nous allons voir comment l'utiliser pour établir des correspondances entre fragments de code, mais aussi pour propager de l'information et conserver un historique des transformations de programme.

4.3.3.1 Description

Un fichier IL comporte deux parties. La première, l'en-tête, consiste en la définition des mots-clés qui seront utilisés dans la suite et d'un niveau de langage auquel il est fait référence quand aucune précision n'est donnée. Dans l'exemple d'en-tête de la figure 4.10, nous définissons trois mots-clés qui permettront de communiquer des informations sur la fréquence d'exécution de portions de code, le nombre de cycles nécessaires et le nombre d'accès à la mémoire. Ces mots-clés n'ont aucune signification intrinsèque, ils doivent être interprétés de façon cohérente par tous les modules qui l'utilisent.

```
Keywords := { NbTimeExecuted, NbCycles, NbLoads }
DefaultLevel = asm
```

FIG. 4.10 – *En-tête d'un fichier* IL

La deuxième partie contient la définition de tous les objets manipulés ainsi que

les propriétés qui y sont attachées (cf. figure 4.11). Un objet est créé la dès qu'il est référencé. Trois types d'objets existent :

INST désigne une instruction. À bas niveau, c'est un mnémonique assembleur, à haut niveau il peut s'agir d'une expression simple en langage C ;

BB représente un bloc de base ;

SS (*subset*) est utilisé pour tout fragment de code qui ne correspond pas à l'une des deux catégories précédentes.

Les objets sont nommés par un identificateur entier. Un objet est désigné de manière unique par le triplet (type, niveau, identificateur).

```
#(BB 1).Size := 5
#(BB 3, DESC = { #(INST 4), #(INST 5) } ).NbCycles := 5
#(INST 2).FlowDep := { #(INST 5) }
#(BB 42).Size := <?>
```

FIG. 4.11 – *Corps d'un fichier* IL

Un objet peut être facilement défini par l'ensemble des objets qu'il contient. Un bloc de base, par exemple, est formé d'un certain nombre d'instructions. Pour cela nous utilisons le mot réservé `DESC`. Dans l'exemple de la figure 4.11, le bloc 3 est composé des instructions numérotées 4 et 5. Par contre au niveau le plus simple, celui des instructions, cette technique n'est pas applicable, une instruction assembleur n'étant pas composée d'objets plus élémentaires. Nommer les instructions par leur numéro de ligne dans le fichier source n'est pas une bonne solution : en effet, la phase de génération de code ne produit pas les instructions de façon séquentielle, et certaines optimisations sont susceptibles de supprimer des instructions (propagation de constantes) ou d'en ajouter (dépliage). L'approche la plus satisfaisante consiste à attribuer à chaque instruction un numéro unique via une nouvelle directive placée juste avant l'instruction concernée (cf. figure 4.12). Ce numéro est attribué soit par le générateur de code, soit directement par notre système s'il n'existe pas de fichier de haut niveau.

```
      .global _mxm
_mxm:
      .instid 2
      IF iadd r1 r1 r0 -> r37;
      .instid 3
      IF ld32d r1 (0) r8 -> r128;
      .instid 4
      IF iimm r1 (0xFFFFFFFD) -> r129;
      .instid 5
      IF isub r1 r128 r129 -> r130;
```

FIG. 4.12 – *Numérotation des instructions assembleur*

Un nombre quelconque de propriétés sont affectées à un objet par notation pointée.

Une propriété possède un type, qui doit être déclaré dans l'en-tête et une valeur qui peut être un entier, une chaîne de caractères, une référence à un objet ou un ensemble de ces éléments.

La grammaire de notre langage est relativement simple. Elle est implémentée avec les outils Lex et Yacc [79] à partir des règles présentées en annexe A.

4.3.3.2 Carte de correspondance

Le langage IL permet d'établir une carte de correspondance entre le langage de haut niveau et l'assembleur (cf. figure 4.13) et de la maintenir à jour au cours des transformations de code. Nous pouvons ainsi savoir quel partie du code écrit en langage de haut niveau a conduit à la version actuelle du code. Le traitement d'un problème peut ainsi être répercuté en amont. Si par exemple l'ordonnancement d'une boucle conduit à un taux d'utilisation des unités fonctionnelles très faible, il peut être intéressant de le signaler à l'outil de transformation de haut niveau qui va alors savoir quelle boucle déplier.

```
DefaultLevel = asm

#(BB[mt1] 0, DESC = { #(INST 2), #(INST 3), ... , #(INST 12) })
#(BB[mt1] 0).SrcStatements := { #(INST[src] 1) }
```

FIG. 4.13 – *Carte de correpondance*

4.3.3.3 Support d'information

IL est utilisé pour propager de l'information entre différentes étapes de compilation. L'analyse des boucles est avantageusement fournie aux transformations qui travaillent sur le code assembleur, notamment le registre de comptage, les instructions de test et de saut. La figure 4.14 décrit la structure d'un programme : le code à optimiser est constitué de deux blocs de base (n^{os} 1 et 54) et d'un fragment nommé SS 2. Celui-ci contient les blocs n^{os} 3, 4 et 5, et se trouve être une boucle. Les lignes suivantes permettent d'apprendre, entre autres choses, que le registre `r37` est utilisé comme compteur et que le test de sortie de boucle est réalisé par l'instruction n^{o} 14.

4.3.3.4 Historique des transformations

En plus des propriétés relatives au code généré, chaque transformation doit pouvoir produire un historique des modifications du code (cf. figure 4.15). La concaténation des historiques de toutes les transformations fournit l'évolution complète d'un programme, à partir du code écrit par le programmeur jusqu'à la production d'un exécutable optimisé. Cet historique s'avère utile pour la mise au point d'une stratégie qui implique des dizaines de transformations.

```
#(SS 1).ToOptimize := true
#(SS 1, DESC = { #(BB 1), #(SS 2), #(BB 54) } )
#(SS 2, DESC = { #(BB 3), #(BB 4), #(BB 5) } )
#(SS 2).Loop := true
#(SS 2).iteratorRsrc := "r37"
#(SS 2).lowerBoundValue := 1
#(SS 2).strideValue := 4
#(SS 2).incrCode := { #(INST 463) }
#(SS 2).loopBack := { #(INST 465) }
#(SS 2).loopExit := { #(INST 15) }
#(SS 2).loopBackTest := { #(INST 14) }
```

FIG. 4.14 – *Flots d'information*

```
#(SS 1).became := {#(SS 780)}
#(SS 780).copy := {{order, 0}, {from, 1}, {to, 976}, {idTra, 0}, {Status, oK}}
#(SS 780).copy := {{order, 1}, {from, 976}, {to, 780}, {idTra, 2},
    {Status, oK}}
#(SS 780).unroll := {{order, 2}, {from, 780}, {to, 780}, {idTra, 3},
    {Status, oK}, {Unroll,  2}}
#(SS 1).became := {#(SS 958)}
#(SS 958).copy := {{order, 0}, {from, 1}, {to, 976}, {idTra, 0},
    {Status, oK}}
#(SS 958).copy := {{order, 1}, {from, 976}, {to, 958}, {idTra, 1},
    {Status, oK}}
#(SS 958).softPipe := {{order, 2}, {from, 958}, {to, 958}, {idTra, 4},
    {Status, oK}, {Unroll,  3}, {II, 5}, {LoopSize, 57}}
```

FIG. 4.15 – *Historique des transformations*

4.3.4 Mise en œuvre distribuée

La modularité du système que nous avons développé nous a permis de le distribuer entre plusieurs sites.

Motivations Le développement du système, dans le contexte d'un projet européen, a fait intervenir plusieurs partenaires. Le développement, la mise au point et la maintenance sont grandement facilités quand chaque outil n'existe qu'en un seul exemplaire sur le site du partenaire qui en est responsable. La correction de problèmes ou l'ajout de fonctionnalités ne nécessitent pas des utilisateurs qu'ils se procurent et installent la nouvelle version du logiciel : celle-ci remplace simplement la précédente, et l'opération est effectuée par le responsable.

Cette approche permet d'être indépendant de l'environnement matériel et logiciel de l'utilisateur et évite la maintenance de dizaines de versions d'un même logiciel.

Les temps de transfert des fichiers, s'ils sont sensibles, restent toutefois raisonnables. n dénotant la longueur d'un programme, le transfert s'effectue en un temps $O(n)$. Les optimisations utilisent souvent des algorithmes dont la complexité est en $O(n^2)$ ou $O(n^3)$, et devraient rester le facteur limitant.

Organisation actuelle Notre système est distribué géographiquement sur plusieurs sites en Europe (cf. figure 4.16, partie gauche). Le moteur qui contrôle l'évolution des optimisations est situé à l'Irisa, mais certaines transformations sont appliquées à l'Inria Rocquencourt ou à l'université de Leiden (Pays Bas). Nous avons prévu que le module d'évaluation du coût des transformations soit localisé à l'université de Manchester (Royaume Uni).

Au sein de l'Irisa, le système est aussi distribué sur deux stations de travail : Sun et SGI pour une raison pratique : nous utilisons PFA (*Power Fortran Accelerator*) pour appliquer certaines transformations de haut niveau sur des programmes Fortran, or PFA n'est disponible à l'Irisa que sur une station de travail SGI. La station Sun fait tourner le dispositif de contrôle des optimisations et possède des interfaces pour communiquer avec les divers outils distants. Elle propose aussi un service qui permet à nos partenaires d'utiliser SPS, le pipeline logiciel construit avec SALTO et SEA (présenté au chapitre 2).

Évolution Nous pensons que la distribution est une évolution intéressante des systèmes de compilation complexes. Notre prototype implémente les transferts de fichiers à l'aide des IPC système V *sockets* mais un développement complet pourrait faire appel à des technologies plus sophistiquées telles que Corba [85, 92] et recourir à des mécanismes d'authentification et de cryptographie pour assurer la confidentialité des données.

Il existe toutefois un inconvénient de taille pendant la phase de mise au point : ne pas maîtriser l'emploi du temps des autres développeurs est parfois à l'origine de grandes frustrations !

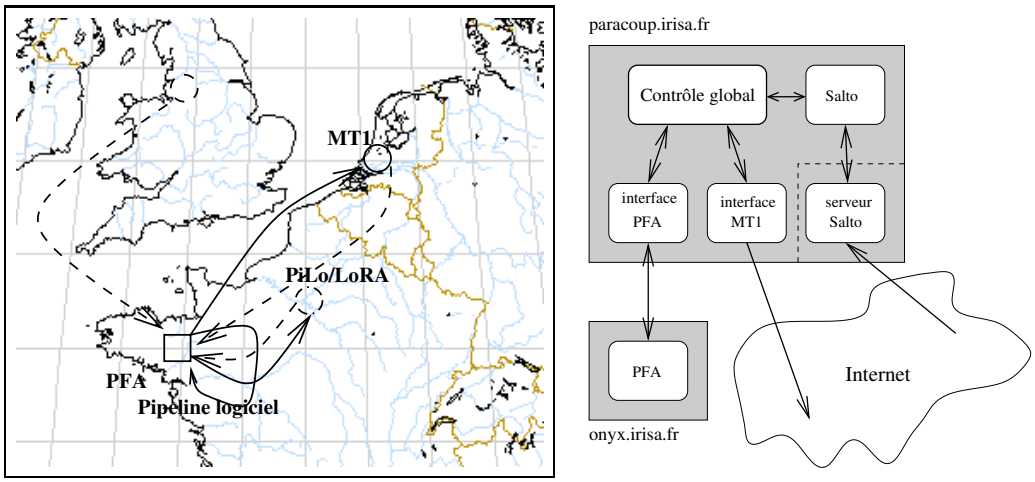


FIG. 4.16 – *Organisation actuelle de notre système de compilation itérative*

Les principaux écueils sont actuellement le manque de bande passante et la saturation du réseau qui risquent de provoquer un déni de service. Ces problèmes devraient être résolu grâce au développement rapide du réseau et l'apparition de nouveau protocoles qui permettent la réservation de bande passante.

4.4 Étude de cas

Nous avons utilisé notre système pour expérimenter trois stratégies de compilation. La première intègre l'approche GCDS présentée au chapitre 3 dans un contexte itératif. La deuxième déplie les boucles et estime le facteur de dépliage optimal grâce à l'information que lui fournit le générateur de code sur le programme produit. Enfin la dernière explore un vaste espace en restreignant la recherche l'aide d'une approche multi-grille.

4.4.1 GCDS

GCDS choisit l'optimisation la plus appropriée à chaque fragment de code considéré en prenant en compte son impact global sur le programme. Dans le chapitre précédent nous avons présenté une restriction de GCDS à des programmes écrits en assembleur. Nous intégrons ici cette approche globale dans un contexte itératif.

L'approche GCDS étant décrite en détails dans le chapitre 3, nous ne la reprenons pas ici. Notre système itératif s'applique simultanément à plusieurs fragments de code. Comme le montre la figure 4.17, des stratégies différentes peuvent contrôler l'optimisation des divers fragments. Quand toutes sont terminées, nous mesurons la taille et la performance de chacun des fragments obtenus et nous construisons le système de contraintes 3.6.

Nous obtenons des résultats similaires à ceux présentés dans le paragraphe 3.4, notamment des courbes avec de nombreux paliers qui représentent autant de compromis

possibles. Le nombre de possibilités est d'autant plus grand que l'éventail des transformations s'est élargi.

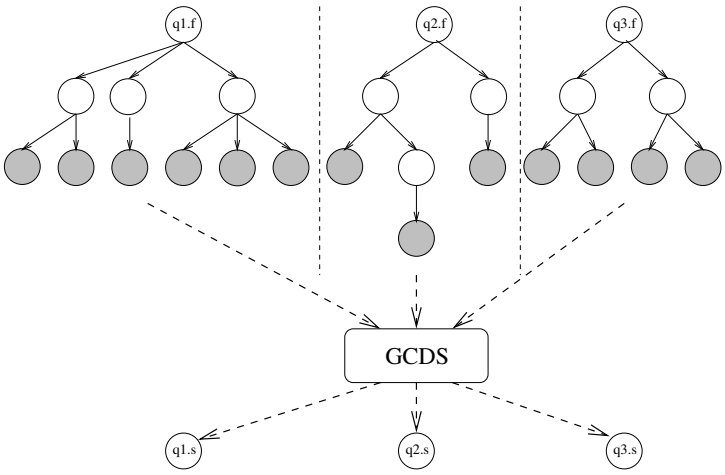


FIG. 4.17 – *Intégration de* GCDS

4.4.2 Contrôle des discontinuités

Notre deuxième exemple s'intéresse au dépliage des boucles. Le dépliage est connu pour augmenter le parallélisme d'instructions (cf. [7]), ce qui est particulièrement intéressant pour un processeur VLIW comme le TM1000. Par contre le dépliage atteint ses limites assez rapidement et n'apporte plus de gain de performance au delà d'un certain facteur. Il risque au contraire de perturber le fonctionnement du cache instructions. De nombreux compilateurs travaillent en aveugle en dépliant les boucles un nombre de fois fixé, ou calculé en fonction de divers paramètres, mais sans être capables d'évaluer le gain apporté.

Nous définissons la «vitesse asymptotique» de la boucle comme le nombre de cycles nécessaires à l'exécution d'une itération. Nous avons sélectionné six boucles et nous les avons dépliées de 2 à 16 fois. Si la vitesse de la boucle dépliée n fois est notée s_n, nous définissons l'accélération par $a_n = \frac{s_1}{s_n}$. Les variations de la vitesse et de l'accélération sont illustrées sur la figure 4.18 (l'ensemble des chiffres est donné en annexe B, ainsi que le code source des boucles). Nous constatons que le dépliage améliore les performances de la boucle, mais aussi que le gain apporté par un dépliage supplémentaire est de plus en plus faible. En outre le gain maximal apporté par le dépliage varie dans de larges proportions : de 1,8 pour `quan` à 6,4 pour `MxM`.

L'accélération apportée par un dépliage n fois ne donne donc pas d'indication sur le potentiel restant. Par contre le taux d'accroissement permet de détecter le ralentissement. La «variation d'accélération» mesure le gain apporté par un dépliage supplé-

mentaire. Elle s'exprime à partir des vitesses :

$$\Delta_n = \frac{a_n - a_{n-1}}{a_{n-1}} = \frac{\frac{s_1}{s_n} - \frac{s_1}{s_{n-1}}}{\frac{s_1}{s_{n-1}}} = \frac{s_{n-1} - s_n}{s_n}$$

Δ_n est représentée sur la figure 4.18.

Notre stratégie consiste à déplier la boucle tant que l'accélération, calculée à chaque itération, s'accroît d'au moins 10 %. La boucle est dépliée à haut niveau, puis le code est généré et ordonnancé. L'ordonnanceur mesure le nombre de cycles (statiques) requis pour le corps de boucle et envoie l'information au moteur de recherche. Celui-ci calcule l'accélération, la compare à l'accélération obtenue à l'itération précédente et décide ou non de continuer. Dans cette étude nous avons continué à déplier les boucles jusqu'à un facteur 16 uniquement pour évaluer l'efficacité de notre approche.

La boucle `MxM` a un comportement qui fait échouer notre stratégie : la variation d'accélération passe sous le seuil des 10 % alors que le potentiel d'optimisation est encore grand. Ce phénomène est dû à l'utilisation des ressources par notre ordonnanceur : le cinquième dépliage a saturé l'utilisation de plusieurs unités fonctionnelles, les instructions qui proviennent du sixième corps de boucle provoquent un accroissement plus important du nombre de cycles. Il est facile de contourner la difficulté en décidant d'arrêter le dépliage quand un nouveau dépliage ne parvient pas non plus à dépasse le seuil. Le facteur de dépliage choisi est donc 3 pour les boucles `quan` et `mkcenter-flat`, 4 pour `reco`, 5 pour `mkcenter` et `fdct` et 9 pour `MxM`. Dans tous les cas, l'accélération atteinte est supérieure à 70 % de celle atteinte pour un dépliage 16 fois.

L'évaluation statique de la vitesse n'est pas un problème : en effet le comportement mémoire est quasiment identique pour les données, le dépliage n'affectant pas l'ordre des accès aux données sauf pour quelques variables d'induction. Tant que la taille du corps de boucle reste inférieure à la taille du cache instructions (ce qui est facilement contrôlable), seuls quelques défauts de cache «à froid» supplémentaires sont introduits. Ils sont négligeables si la boucle est exécutée souvent ou itère un grand nombre de fois. Les branchements conditionnels internes au corps de la boucle ne sont pas pris en compte. Dans le cas de boucles qui possèdent un flot de contrôle complexe, il serait important de l'intégrer dans le modèle. Pour l'étude de noyaux de calculs simples que nous avons utilisés, notre estimation est suffisante.

Notre approche a l'avantage de connaître l'accélération apportée à la boucle et peut donc contrôler le facteur de dépliage en termes de performances. Cet exemple illustre l'approche itérative avec un exemple très simple, mais quasiment impossible à mettre en œuvre dans un compilateur standard du fait de la répétition des transformations de haut niveau et de bas niveau, incompatible avec la structure classique des compilateurs.

4.4.3 Restriction de l'espace de recherche

Dans cette étude nous appliquons la compilation itérative à la résolution du problème suivant : l'optimisation d'un programme soumis à l'action de plusieurs transformations de programme [21]. En plus d'être paramétrées par des nombres entiers

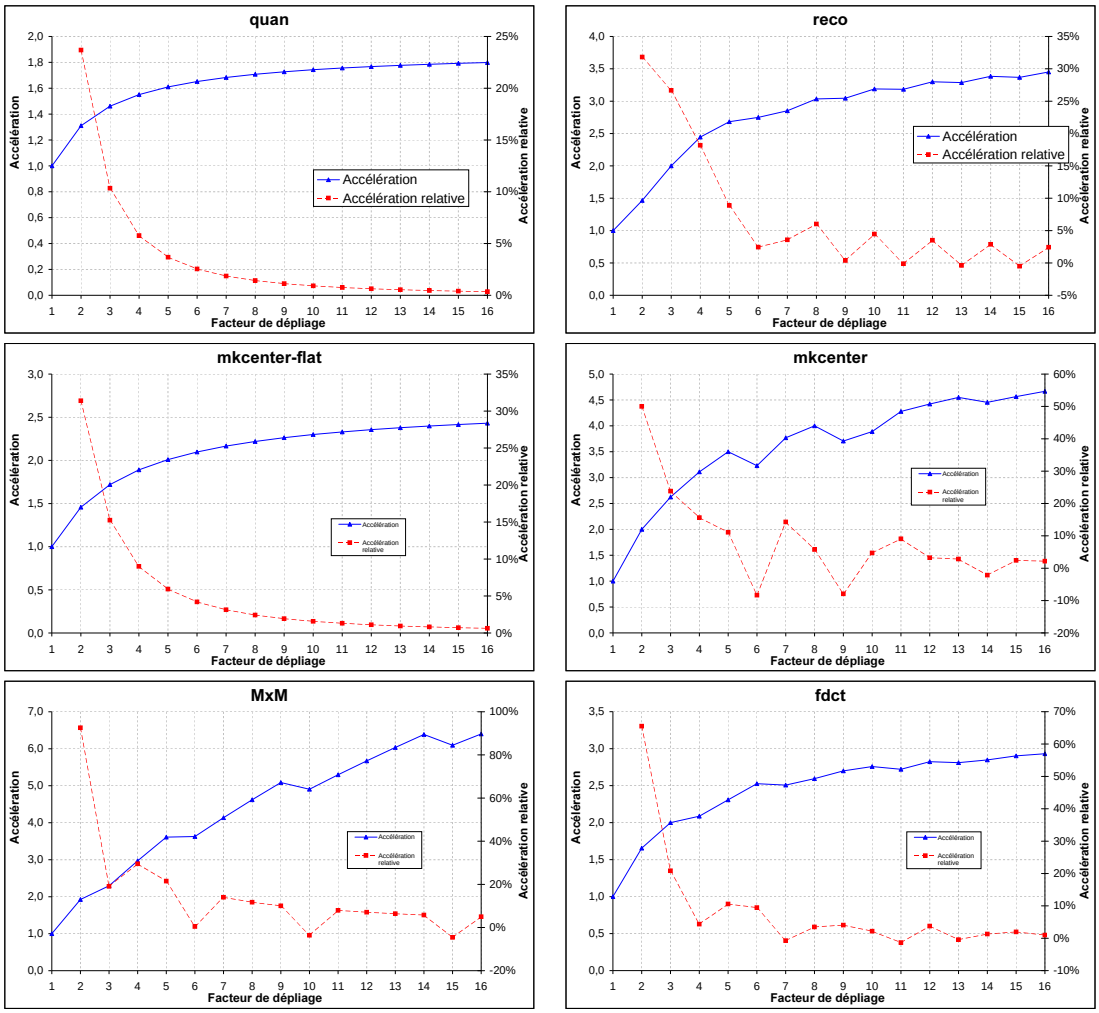


FIG. 4.18 – *Influence du dépliage sur la vitesse des boucles*

comme dans l'exemple précédent, ces transformations interagissent de telle façon qu'il n'est pas possible de déterminer le meilleur paramètre pour chaque transformation indépendamment de l'autre. Notre approche consiste donc à échantillonner l'espace des optimisations pour restreindre la recherche à des régions susceptibles de contenir l'optimum.

Nous nous sommes intéressés à l'optimisation d'un produit de matrices pour plusieurs raisons :

- c'est un exemple bien connu, il est facile de comparer les résultats d'autres études avec les nôtres ;
- c'est la base d'une fonction fondamentale d'un décodeur MPEG-2 ;
- un grand nombre de transformations de programmes sont possibles.

Nous avons évalué notre approche avec différents processeurs : Ultrasparc, R10000,

Pentium Pro, Alpha et TM1000. Deux tailles de matrices sont utilisées : $N = 400$ et $N = 512$.

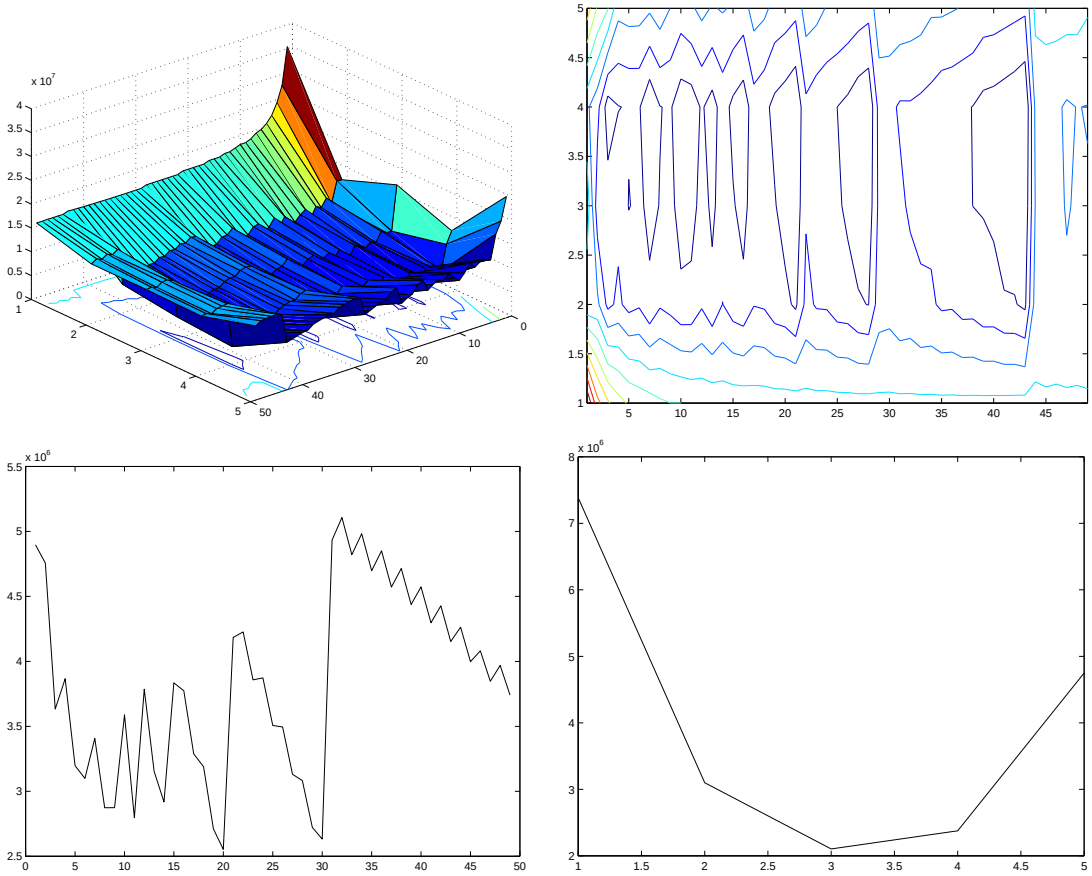
4.4.3.1 Combiner blocage et dépliage

Le blocage et le dépliage (cf. le paragraphe 1.3.1) sont des transformations de code de haut niveau paramétrables. Les espaces d'itérations des deux boucles externes du produit de matrices sont décomposées en pavés carrés, puis les deux boucles internes sont dépliées (les codes Fortran correspondant au blocage puis au dépliage sont présentés en annexe C). Nous avons fait varier la taille des pavés de 2 à 100 et le facteur de dépliage de 1 à 20. Notre mesure de la performance d'un programme est simplement le temps d'exécution réel. La figure 4.19 montre comment évolue ce temps en fonction de la taille des pavés et du facteur de dépliage des boucles. Ces valeurs correspondent au TM1000 pour $N = 64$ mais sont représentatives de toutes nos expériences avec une surface irrégulière, très accidentée, des oscillations. La partie droite en haut de la figure représente les courbes de niveaux de la surface. Elle confirme l'existence de nombreux minima locaux. Les deux courbes du bas représentent des vues en coupe de la surface selon les deux axes. Nous avons isolé sur la figure 4.20 les paramètres pour lesquels le temps ne dépasse pas de plus de 20 % le temps minimum. Il apparaît que la configuration idéale dépend fortement du type de processeur utilisé, mais aussi un peu de la taille des données : l'Ultrasparc et l'Alpha obtiennent les meilleurs performances avec un facteur de dépliage faible et des tailles de pavés variables, le Pentium montre des minima très dispersés tandis que le R10000 possède le plus grand nombre de points proches du minimum. À l'inverse, l'Alpha possède peu de points à moins de 20 % du minimum et promet donc une recherche plus difficile. Sur tous nos exemples, l'optimisation conduit à un gain de performance compris entre 1,8 et 10.

4.4.3.2 Approche multi-grille

Comme nous pouvons le constater sur la figure 4.19, la surface qui représente la performance comme une fonction du facteur de dépliage et de la taille des blocs comporte de nombreux minima locaux. Le processeur et la taille du problème influent sur leur répartition. L'algorithme utilisé pour localiser le minimum global doit être suffisamment robuste pour éviter d'être happé par un de ces minima locaux, ce qui exclut par exemple des approches basées sur le gradient. Toutefois il doit être capable de restreindre fortement le nombre d'évaluations en évitant de s'attarder dans des régions où aucun minimum n'existe. Le problème est d'autant plus difficile que certains minima se trouvent très proches de points où la fonction prend une valeur très élevée.

Principe de notre approche Notre algorithme visite successivement un certain nombre de points régulièrement espacés (selon un pas fixé) et applique la transformation de code correspondant aux coordonnées du point courant. Le programme transformé est compilé et exécuté, et le temps d'exécution est mesuré. Tous les points dont la valeur est comprise entre le minimum et la moyenne courants sont mémorisés. Ils servent de



Variation avec la taille du pavé — Variation avec le facteur de dépliage

FIG. 4.19 – *Aspects de l'espace des optimisations*

centre pour une nouvelle région à étudier plus finement, par exemple avec un pas moitié. Le processus se répète jusqu'à ce qu'un nombre de points fixé à l'avance soient évalués, la transformation qui a fourni le meilleur programme est alors sélectionnée.

Réglage des paramètres La taille du pas dans chacune des directions est un paramètre clé de l'algorithme. Nous avons trouvé que le pas qui donne la meilleure efficacité dépend non seulement de l'application, du processeur et du jeu de données, mais aussi du nombre maximum d'évaluations autorisées. Un certain pas peut conduire au minimum très rapidement si seulement 20 évaluations sont autorisées, mais ne pas être très efficace dès lors que le coût passe à 200 évaluations. Toutefois un pas initial de 5, c'est-à-dire une division de l'espace de transformations en cinq dans les deux directions, donne une efficacité raisonnable pour toutes les configurations.

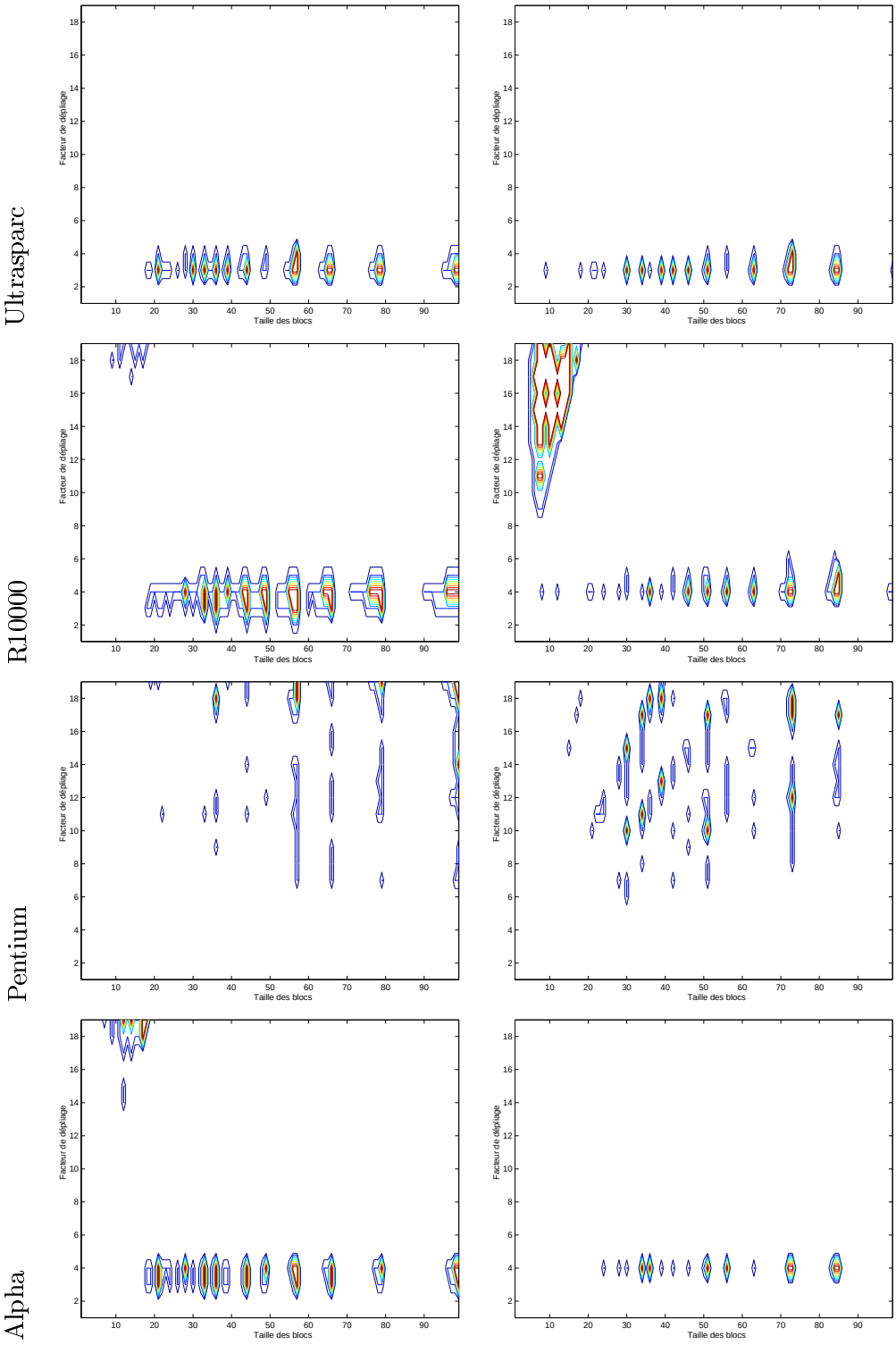
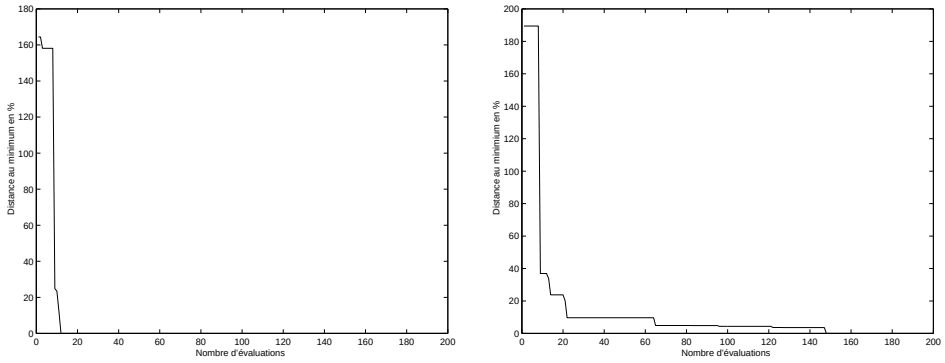


FIG. 4.20 – *Espaces des transformations pour différentes architectures et différentes tailles de problème*

Performance de l'algorithme La figure 4.21 montre les résultats de notre approche : l'axe des abscisses indique le nombre d'évaluations réalisées par notre système, l'axe des ordonnées la distance en pourcentage au minimum global. Par exemple sur le premier graphique, le premier point se trouve à 165 %, le programme correspondant s'exécute donc 2,65 fois moins vite que le plus rapide. Dans le cas de l'Alpha il s'agit d'un facteur 5,25. Les courbes correspondant aux autres processeurs présentent les mêmes caractéristiques. Dans tous les cas notre algorithme trouve le minimum en moins de 200 itérations, soit 10 % de l'espace complet. Il lui suffit de 20 itérations, 1 % de l'espace, pour atteindre 23 % du minimum. Le gain est très rapide pendant les vingt premières itérations, puis ralentit progressivement. Malgré l'aspect de l'espace, un algorithme relativement simple parvient à de bons résultats en peu d'itérations.

Insistons sur le fait que notre but ici n'est pas de présenter un algorithme très efficace, mais de valider notre approche de compilation itérative.



Ultrasparc $N = 400$ R10000 $N = 512$

FIG. 4.21 – *Efficacité de notre algorithme*

4.4.3.3 Résultats

Le TM1000 a pour vocation d'être un processeur enfoui. Les applications qu'ils fait tourner doivent être fortement optimisées et ce contexte autorise des temps de compilation relativement longs. Toutefois nous ne disposons que d'un simulateur et l'exécution d'un programme souffre d'un ralentissement d'un facteur 1000 environ : nous ne pouvons donc pas nous permettre d'explorer l'ensemble de l'espace des transformations. Le TM1000 constitue donc un cas idéal pour tester notre algorithme.

L'emploi du simulateur nous a contraint à réduire la dimension des boucles à $N = 64$ et $N = 128$. Notre politique d'allocation des registres étant actuellement incapable d'insérer du *spill code*, elle échoue quand une boucle est trop dépliée. Dans le cas du produit de matrices, cela se produit pour un dépliage de facteur 5, et nous devons ordonnancer le code après la phase d'allocation. Il en résulte une dégradation des performances telle que nous avons ignoré les dépliages de facteur supérieur. Notre nouvel espace contient donc $5 \times 200 = 1000$ points.

Dans cette situation, contrairement aux expériences menées pour les processeurs généraux, nous ne connaissons pas la valeur du minimum global. La figure 4.22 montre, pour chaque itération de notre algorithme, les performances mesurées pour le produit de matrices ramenées au minimum trouvé en 100 itérations. La droite horizontale indique les performances du programme original auquel aucune transformation n'est appliquée. Il apparaît que le nombre d'itérations nécessaires pour atteindre une certaine performance n'est pas proportionnel à la taille de l'espace global. Le comportement est toutefois identique à celui des expériences précédentes et l'algorithme gagne un facteur 6 sur les performances du programme original en une cinquantaine d'itérations.

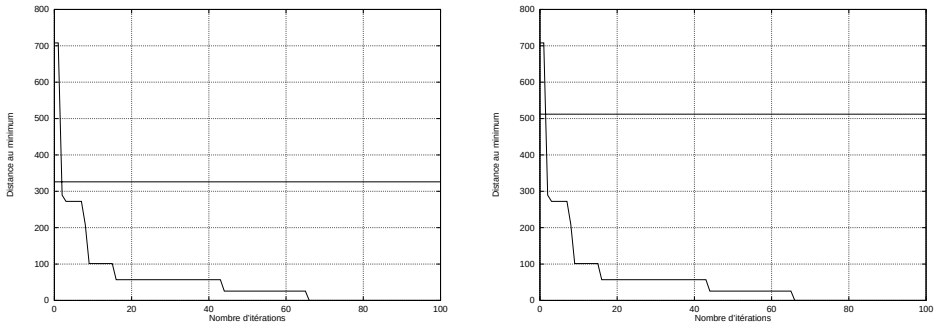


FIG. 4.22 – *Performance de l'algorithme sur le TM1000*

Dans [37], Coleman et McKinley proposent une méthode analytique pour déterminer la taille des pavés en fonction des paramètres du cache. Nous montrons dans [21] que, en présence d'autres transformations (dépliage et *padding*), la taille optimale des pavés est différente. Ainsi les paramètres optimaux des transformations sont différents selon qu'elles sont appliquées en séquence ou seules.

Notre approche ne fait appel à aucune analyse statique du code ni à aucune information sur les caractéristiques architecturales du processeur utilisé. Cela lui confère la possibilité d'être immédiatement adaptable à n'importe quelle modification de l'architecture ou des optimisations appliquées à bas niveau. L'inconvénient est de ne pas tirer parti des avancées dans le domaine de l'analyse de code. Pour intégrer ces nouveaux résultats à notre algorithme, nous pouvons utiliser les solutions des analyses statiques comme points de départ de notre recherche. Nous garantissons ainsi des performances au moins aussi bonnes à notre algorithme.

4.4.3.4 Travaux connexes

Nous avons vu que l'espace dans lequel doit évoluer l'algorithme n'a pas de «bonnes propriétés» qui permettraient d'appliquer des techniques classiques comme le gradient. L'intelligence artificielle a développé des algorithmes capables de s'adapter à de tels environnements [45] : systèmes experts, $A^\star$, etc. Les techniques les plus proches de

notre solution sont les algorithmes génétiques et le recuit simulé :

Recuit simulé : L'algorithme cherche le minimum global d'une fonction en évaluant itérativement des points de proche en proche. Il est autorisé à sélectionner un point dont la valeur est supérieure au point courant de façon à pouvoir sortir d'un minimum local, mais le seuil qu'il peut franchir baisse avec le temps [27, 40, 78].

Algorithmes génétiques : Ils simulent un phénomène d'évolution basé sur la sélection naturelle [40]. Une population est composée d'individus, chacun étant identifié par son code génétique (représenté par une chaîne de caractères). À partir d'une population initiale aléatoire, chaque individu est évalué selon un critère qui correspond à la fonction à optimiser. Les plus mauvais sont éliminés tandis que les meilleurs sont sélectionnés pour se reproduire. La reproduction consiste à fabriquer un nouveau code génétique en concaténant des fragments de codes des parents, à la manière des chromosomes pendant le *cross-over*. Des mutations sont introduites de façon aléatoire pour sortir des minima locaux.

Les algorithmes génétiques ont été utilisés par Nisbet [89] pour paralléliser des programmes.

4.5 Conclusion

Dans ce chapitre nous avons montré que l'interaction entre les diverses optimisations appliquées à un fragment de code a des répercussions sur les performances du code produit. Ces interactions sont profondes et ne peuvent généralement pas être capturées par un simple modèle analytique. Chaque application peut entraîner un comportement différent.

Nous avons proposé une approche itérative de la compilation qui permet d'explorer l'espace des optimisations et d'évaluer *a posteriori* l'impact des transformations appliquées. Nous avons implémenté quelques stratégies de compilation simple pour valider notre approche. Notre approche globale GCDS, présentée au chapitre précédent, s'intègre aussi naturellement dans ce cadre.

Nous n'avons pas abordé le problème du changement de structure d'un fragment de code au cours de l'optimisation, par exemple quand le nombre de boucles change après une distribution. Cet aspect a un impact sur les possibilités d'exploration d'une stratégie et mérite d'être étudié.

Conclusion

Les processeurs intègrent aujourd'hui plusieurs dizaines de millions de transistors et disposent de mécanismes toujours plus complexes. Parallèlement la recherche en compilation et optimisations développe des techniques de transformations de programmes pour exploiter ces nouveautés.

Nous sommes arrivés à un point où un grand nombre de techniques existent, chacune adaptée à l'amélioration d'un facteur de performance : comportement de la hiérarchie mémoire, remplissage du pipeline d'exécution, prédiction des branchements, etc. Ces transformations de code interagissent les unes avec les autres de telle sorte qu'il est souvent impossible d'estimer le gain apporté par une séquence d'optimisations.

La séquence d'optimisations qui va conduire à la meilleure performance dépend de l'application considérée. La déterminer demande d'explorer l'espace des transformations de code en se guidant avec des informations en provenance des différents niveaux. Un compilateur classique n'est pas capable de résoudre efficacement le problème de l'optimisation pour la performance qui requiert deux fonctionalités :

- il est nécessaire que certaines décisions prises par un module d'optimisation puissent être remises en cause par un autre, éventuellement une phase précédente doit être recommencée avec d'autres paramètres ;
- les différents modules doivent pouvoir se communiquer des informations sur les propriétés du code qu'ils ont produit.

Or la structure classique distingue nettement les transformations qui ont lieu avant la génération de code et celles qui ont lieu après et interdit toute rétroaction.

La possibilité d'explorer l'espace des optimisations implique une réorganisation de la structure des compilateurs. Nous montrons dans le chapitre 4 que cette capacité est pourtant essentielle à l'obtention de performances élevées. Nous avons proposé une approche itérative qui permet à l'algorithme d'optimisation d'évaluer les performances *a posteriori* et d'être guidé par ces mesures. Il peut ainsi prendre en compte les interactions réelles qui existent entre différentes transformations de code.

Les expériences que nous avons menées ont été rendues possibles grâces aux infrastructures logicielles que nous avons développées : Salto et SEA. Salto prend en charge la manipulation du code assembleur et permet au développeur de s'affranchir des tâches ingrates telles que l'analyse syntaxique de l'assembleur ou la construction du graphe de flot et de se concentrer sur l'aspect algorithmique, tandis que SEA fournit une abstraction du code assembleur et simplifie la mise au point de stratégies de compilation. Grâce à ces environnements, nous avons développé rapidement les prototypes

de stratégies qui nous ont permis de valider notre approche.

Nous sommes convaincus de la nécessité de disposer de tels outils pour mener à bien des projets de recherches similaires à ceux que nous avons entrepris. La diffusion de SALTO dans d'autres universités et l'intérêt de quelques entreprises nous conforte dans cette idée.

L'interaction entre les optimisations des différents fragments de code qui constituent le programme doit aussi être prise en compte, en particulier pour les systèmes enfouis. Notre approche GCDS explore des alternatives qui n'avaient pour l'instant pas été étudiées. L'impact sur les performances, notamment dans le contexte des systèmes enfouis, est significatif: le fait d'ignorer l'aspect global ne permet pas de tirer pleinement parti des ressources disponibles en ignorant le degré de liberté supplémentaire que procure leur distribution entre différents fragments.

Nous avons montré l'intérêt de cette approche itérative de la compilation à l'aide de quelques exemples de stratégies pour des boucles. L'espace des transformations est très vaste et ne peut être exploré entièrement. La suite des recherches dans ce domaine doit consister à développer des stratégies plus sophistiquées capables de s'adapter à un programme quelconque. Pour des raisons de temps, nous n'avons pas abordé le problème de la décomposition du programme original en fragments susceptibles d'être optimisés séparément. Cette décomposition influe sur les directions que peut prendre l'algorithme d'optimisations ainsi que sur les compromis globaux et doit aussi faire l'objet de travaux. ■

Bibliographie

[1] Aarts (Bas), Barreteau (Michel), Bodin (François), Brinkhaus (Peter), Chamski (Zbigniew), Charles (Henri-Pierre), Eisenbeis (Christine), Gurd (John R.), Hoogerbrugge (Jan), Hu (Ping), Jalby (William), Knijnenburg (Peter M.W.), O'Boyle (Michael F.P.), Rohou (Erven), Sakellariou (Rizos), Schepers (Henk), Seznec (André), Stöhr (Elena A.), Verhoeven (Marco) et Wijshoff (Harry A.G.). – Oceans: Optimizing compilers for embedded HPC applications. *In : Lecture Notes in Computer Science*. – Springer Verlag, août 1997.

[2] Adve (Sarita V.), Burger (Doug), Eigenmann (Rudolf), Rawsthorne (Alasdair), Smith (Michael D.), Gebotys (Catherine H.), Kandemir (Mahmut T.), Lilja (David J.), Choudhary (Alok N.), Fang (Jesse Z.) et Yew (Pen-Chung). – Theme feature: Changing interaction of compiler and architecture. *Computer*, vol. 30, n° 12, décembre 1997, p. 51–58.

[3] Aho (A. V.), Sethi (Ravi) et Ullman (J. D.). – *Compilers: Principles, Techniques and Tools*. – Addison-Wesley, 1985.

[4] Allan (Vicki H.), Jones (Reese B.), Lee (Randall M.) et Allan (Stephen J.). – Software pipelining. *ACM Computing Surveys*, vol. 27, septembre 1995, p. 367–432.

[5] Amme (Wolfram), Braun (Peter), Thomasset (François) et Zehendner (Eberhard). – Data dependence analysis of assembly code. *In : PACT'98*. – octobre 1998.

[6] Anderson (J.), Berc (L. M.), Dean (J.), Ghemawat (S.), Henziger (M. R.), Leung (S.), Sites (R. L.), Vandevoorde (M. T.), Waldspurger (C. A.) et Weihl (W. E.). – Continuous profiling: Where have all the cycles gone? *In : Proceedings of the 16^{th} Symposium on Operating Systems Principles*, p. 1–14. – octobre 1997.

[7] Bacon (David F.), Graham (Susan L.) et Sharp (Oliver J.). – Compiler transformations for high-performance computing. *ACM Computing Surveys*, vol. 26, décembre 1994, p. 345–420.

[8] Ball (Thomas) et Larus (James R.). – Branch prediction for free. *In : Conference on Programming Language Design and Implementation*, p. 300–313. – juin 1993.

[9] Ball (Thomas) et Larus (James R.). – Optimally profiling and tracing programs. *In : ACM Transactions on Programming Languages and Systems*, p. 1319–1360. – juillet 1994.

[10] Ball (Thomas), Mataga (Peter) et Sagiv (Mooly). – Edge profiling versus path profiling: The showdown. *In : Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 134–148. – San Diego, California, 19–21 janvier 1998.

[11] Banerjee (Utpal). – *Dependence analysis for supercomputing.* – Boston, MA, USA, Kluwer Academic, 1988, x + 155p.

[12] Barrado (C.), Labarta (J.), Ayguadé (E.) et Valero (M.). – Automatic generation of loop schedulings for VLIW. *In : Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, éd. par Bic (Lubomir), Böhm (Wim), Evripidou (Paraskevas) et Gaudiot (Jean-Luc). p. 306–309. – Limassol, Chypre, juin 27–29, 1995.

[13] Barreteau (M.), Charles (H-P), Jalby (W.), Eisenbeis (C.), Hu (P.), Bodin (F.), Rohou (E.), Seznec (A.), Stöhr (Elena) et Chamski (Z.). – *Advanced code optimization algorithms.* – Deliverable 2.2, ESPRIT Project OCEANS, novembre 1998.

[14] Barreteau (Michel), Bodin (François), Brinkhaus (Peter), Chamski (Zbigniew), Charles (Henri-Pierre), Eisenbeis (Christine), Gurd (John R.), Hoogerbrugge (Jan), Hu (Ping), Jalby (William), Knijnenburg (Peter M.W.), O'Boyle (Michael F.P.), Rohou (Erven), Sakellariou (Rizos), Seznec (André), Stöhr (Elena A.), Treffers (Menno) et Wijshoff (Harry A.G.). – Oceans: Optimizing compilers for embedded applications. – septembre 1998. À paraître.

[15] Bernstein (D.) et Rodeh (M.). – Global instruction scheduling for superscalar machines. *In : Conference on Programming Language Design and Implementation*, p. 241–255. – Toronto, Ontario, Canada, juin 1991.

[16] Berson (D.), Gupta (R.) et Soffa (M. L.). – Resource spackling: A framework for integrating register allocation in local and global schedulers. *In : International Conference on Parallel Architectures and Compilation Techniques.* – août 1994.

[17] Berson (David A.), Chang (Pohua), Gupta (Rajiv) et Soffa (Mary Lou). – Integrating program optimizations and transformations with the scheduling of instruction level parallelism. *In : Lecture Notes in Computer Science*, p. 207–221. – août 1996.

[18] Bik (Aart J. C.). – *MT1: A Prototype Restructuring Compiler.* – Rapport technique, Rijksuniversiteit te Leiden, octobre 1993.

[19] Bodin (F.), Beckman (P.), Gannon (D.) et Srinivas (J.G.S.). – Sage++: A class library for building Fortran and C++ restructuring tools. *In : Object-Oriented Numerics Conference.* – avril 1994.

[20] Bodin (F.), Chamski (Z.), Lelait (S.), Rohou (E.), Sawaya (A.), Seznec (A.) et Wang (J.). – Towards a retargetable framework for software pipelining. *In : Compilers for Parallel Computers*, éd. par Fritzson (Peter), p. 90–99. – Linköping, Suède, juin 1998.

[21] Bodin (F.), Kisuki (T.), Knijnenburg (P.M.W.), O'Boyle (M.F.P.) et Rohou (E.). – Iterative compilation in a non-linear optimisation space. *In : Workshop on Profile and Feedback-Directed Compilation, PACT'98.* – octobre 1998.

[22] Bodin (François). – Transformations de programmes pour l'amélioration de performance. – Document d'habilitation, IRISA/IFSIC, 1997.

[23] Bodin (François), Chamski (Zbigniew), Rohou (Erven) et Seznec (André). – *Functional Specification of* SALTO*: A Retargetable System for Asssembly Language Transformation and Optimization.* – Deliverable 1, ESPRIT Project OCEANS, janvier 1997.

[24] Bodin (François), Mével (Yann) et Quiniou (René). – A user level program transformation tool. *In : Proc. International Conference on Supercomputing.* – juillet 1998.

[25] Bradlee (David G.), Eggers (Susan J.) et Henry (Robert R.). – The Marion system for retargetable instruction scheduling. *In : Programming Languages and Systems.* – 1991.

[26] Bradlee (David G.), Henry (Robert R.) et Eggers (Susan J.). – Integrating register allocation and instruction scheduling for RISCs. *In : 4^{th} International Conference on Architecture Support for Programming Languages and Operating Systems*, p. 122–131. – avril 1991.

[27] Brooks (S. P.) et Morgan (B. J. T.). – Optimization using simulated annealing. *In : The Statistician*, p. 241–257. – 1995.

[28] Calder (Brad) et Grunwald (Dirk). – Reducing branch costs via branch alignment. *In : 6^{th} International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 242–251. – octobre 1994.

[29] Carr (Steve). – Combining optimization for cache and instruction-level parallelism. *In : PACT.* IEEE. – 1996.

[30] Case (Brian). – Philips hopes to displace DSPs with VLIW. *Microprocessor Report*, no1, décembre 1994, p. 12–15.

[31] Chang (Pohua P.), Mahlke (Scott A.), Chen (William Y.), Warter (Nancy J.) et Hwu (Wen-mei W.). – IMPACT: An architectural framework for multiple-instruction-issue processors. *In : International Symposium on Computer Architecture*, p. 266–275. – 1991.

[32] Charot (François), Le Fol (Gwendal) et Messé (Vincent). – Modélisation de processeurs spécialisés programmables pour l'adéquation application/architecture/compilateur. *In : Journées Adéquation Algorithme Architecture en traitement du signal et images*, p. 27–34. – janvier 1998.

[33] Chekuri (C.), Johnson (R.), Motwani (R.), Natarajan (B.), Rau (B. R.) et Schlansker (M.). – Profile-driven instruction level parallel scheduling with application to super blocks. *In : MICRO*, p. 58–67. – décembre 1996.

[34] Cmelik (Bob) et Keppel (David). – *Shade: A Fast Instruction-Set Simulator for Execution Profiling.* – Rapport technique, mai 1994.

[35] Cohn (Robert), Goodwin (David), Lowney (P. Geoffrey) et Rubin (Norman). – Spike: An optimizer for Alpha/NT executables. *In : The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, éd. par USENIX. p. 17–23. – Berkeley, CA, USA, août 1997.

[36] Cohn (Robert) et Lowney (P. Geoffrey). – Hot cold optimization of large Windows/NT applications. *In : Proceedings of the 29th Annual International Symposium on Microarchitecture*. IEEE Computer Society TC-MICRO and ACM SIGMICRO, p. 80–89. – Paris, France, décembre 2–4, 1996.

[37] Coleman (Stephanie) et McKinley (Kathryn S.). – Tile size selection using cache organization and data layout. *In : Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, p. 279–290. – La Jolla, California, 18–21 juin 1995.

[38] Corporation (LSI Logic). – *SPARC Architecture Manual (Version 7)*, 1990.

[39] Davidson (Jack W.) et Holler (Anne M.). – Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, vol. 18, n° 2, février 1992, p. 89–101.

[40] Davis (Lawrence). – *Genetic Algorithms and Simulated Annealing*. – Morgan Kaufmann Publishers, Inc., 1987.

[41] Debray (Saumya), Muth (Robert) et Weippert (Matthew). – Alias analysis of executable code. *In : Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 12–24. – San Diego, California, 19–21 janvier 1998.

[42] DEC. – *ATOM User Manual*, mars 1994.

[43] Ebcioğlu (Kemal), Groves (Randy D.), Kim (Ki-Chang), Silberman (Gabriel M.) et Ziv (Isaac). – VLIW compilation techniques in a superscalar environment. *In : Conference on Programming Language Design and Implementation*, p. 36–48. – Orlando, Florida, juin 20–24, 1994.

[44] Emmelmann (H.), Schroeer (F.-W.) et Landwehr (R.). – BEG - a generator for efficient back ends. *In : Conference on Programming Language Design and Implementation*. – juillet 1989.

[45] Farrenty (Henri) et Ghallab (Malik). – *Éléments d'intelligence artificielle*. – Hermès, 1990, 2e édition.

[46] Fisher (J.A.). – Trace scheduling: a technique for global microcode compaction. *In : IEEE Transactions on Computers*, p. 478–490. – juillet 1981.

[47] Fisher (Joseph A.) et Freudenberger (Stefan M.). – Predicting conditional branch directions from previous runs of a program. *In : Conference on Architecture Support for Programming Languages and Operating Systems*, p. 85–95. – 1992.

[48] Fourer (Robert) et Gregory (John W.). – Linear Programming FAQ. – World Wide Web `http://www-c.mcs.anl.gov/home/otc/Guide/faq/-linear-programming-faq.html`, Usenet sci.answers, FTP anonyme `/pub/-usenet/sci.answers/linear-programming-faq` sur rtfm.mit.edu., 1997.

[49] Gasperoni (Franco). – *Scheduling for horizontal systems : the VLIW paradigm in perspective*. – Thèse de PhD, New-York University, 1991.

[50] Ghiya (Rakesh) et Hendren (Laurie J.). – Putting pointer analysis to work. *In : Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 121–133. – San Diego, California, 19–21 janvier 1998.

[51] Gillies (David M.), Ju (Dz-ching Roy), Johnson (Richard) et Schlansker (Michael). – Global predicate analysis and its application to register allocation. *In : Proceedings of the 29th Annual International Symposium on Microarchitecture*. IEEE Computer Society TC-MICRO and ACM SIGMICRO, p. 114–125. – Paris, France, décembre 2–4, 1996.

[52] Glakowsky (Peter N.). – First Medei processors reach the market. *Microprocessor Report*, no1, janvier 1997, p. 10–15.

[53] Gloy (N.), Blackwell (T.), Smith (M. D.) et Calder (B.). – Procedure placement using temporal ordering information. *In : Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*. p. 303–313. – Los Alamitos, décembre 1–3 1997.

[54] Goodman (James R.) et Hsu (Wei-Chung). – Code scheduling and register allocation in large basic blocks. *In : International Conference on Supercomputing*, p. 442–452. – 1988.

[55] Govindarajan (R.), Altman (Erik R.) et Gao (Guang R.). – Minimizing register requirements under resource-constrained rate-optimal software pipelining. *In : 27th Annual International Symposium on Microarchitecture*, p. 85–94. – 1994.

[56] Graham (Susan L.), Kessler (Peter B.) et McKusick (Marshall K.). – gprof: A call graph execution profiler. *SIGPLAN Notices*, vol. 17, n° 6, juin 1982, p. 120–126. – *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[57] Gösmann (Klaus), Hafer (Christian), Lindmeier (Horst), Plankl (Josef) et Westerholz (Karl). – Code reorganization for instruction caches. *In : 26th Annual Hawaii International Conference on System Sciences*, p. 214–223. – 1993.

[58] Gupta (R.) et Soffa (M. L.). – Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, vol. 16, n° 4, avril 1990, p. 421–431.

[59] Gwennap (Linley). – Intel, HP make EPIC disclosure. *Microprocessor Report*, vol. 11, n° 14, octobre 1997, p. 1,6–9.

[60] Hadjiyiannis (George). – *ISDL: Instruction Set Description Language Version 1.0.* – MIT RLE, avril 1998.

[61] Hadjiyiannis (George), Hanono (Silvina) et Devadas (Srivinas). – ISDL: An instruction set description language for retargetability. *In : 34th Design Automation Conference*. ACM. – 1997.

[62] Hall (Mary Wolcott). – *Managing Interprocedural Optimization.* – Technical Report n° TR91-157, Rice University, avril 28, 1998.

[63] Hank (R. E.), Hwu (W. W.) et Rau (B. R.). – Region-based compilation: An introduction and motivation. *In : 28th Annual International Symposium on Microarchitecture*, p. 158–168. – 1995.

[64] Hank (Richard E.), Mahlke (Scott A.), Bringmann (Roger A.), Gyllenhaal (John C.) et Hwu (Wen-mei W.). – Superblock formation using static program analysis. *In : MICRO*, p. 247–256. – décembre 1993.

[65] Hanono (Silvina) et Devadas (Srivinas). – Instruction selection, resource allocation, and scheduling in the Aviv retargetable code generator. *In :* 35^{th} *Design Automation Conference*. ACM. – juin 1998.

[66] Hashemi (Amir H.) et Kaeli (David R.). – Efficient procedure mapping using cache line coloring. *In : Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*. p. 171–182. – New York, juin 15–18 1997.

[67] Hennessy (J. L.) et Patterson (D. A.). – *Computer Architecture: A Quantitative Approach*. – San Mateo, California, Morgan Kaufmann, 1996, second édition.

[68] Hwu (W. W.) et Chang (P. P.). – Inline function expansion for compiling c programs. *In : Conference on Programming Language Design and Implementation*. SIGPLAN, p. 246–257. – 1989.

[69] Hwu (Wen-mei W.), Mahlke (Scott A.), Chen (William Y.), Chang (Pohua P.), Warter (Nancy J.), Bringmann (Roger A.), Ouellette (Roland G.), Hank (Richard E.), Kiyohara (Tokuzo), Haab (Grant E.), Holm (John G.) et Lavery (Daniel M.). – The superblock: An effective technique for VLIW and superscalar compilation. *In : The Journal of Supercomputing*, p. 229–248. – mai 1993.

[70] Irlam (Gordon). – Spa package. – Disponible à l'adresse `ftp://chook.cs.adelaide.edu.au/pub/sparc/spa-1.0.tar.gz`, 1991.

[71] Johnson (Richard) et Schlansker (Michael). – Analysis techniques for predicated code. *In : Proceedings of the 29^{th} Annual International Symposium on Microarchitecture*. IEEE Computer Society TC-MICRO and ACM SIGMICRO, p. 100–113. – Paris, France, décembre 2–4, 1996.

[72] Kerns (Daniel R.) et Eggers (Susan J.). – Balanced scheduling: Instruction scheduling when memory latency is uncertain. *In : Conference on Programming Language Design and Implementation*, p. 278–289. – 1993.

[73] Knuth (D.) et Stevenson (F.). – Optimal measurement points for program frequency counts. *BIT*, vol. 13, 1973, p. 313–322.

[74] Kuck (D. J.), Kuhn (R. H.), Padua (D. A.), Leasure (B.) et Wolfe (M.). – Dependence graphs and compiler optimization. *In : Proceedings of the Eighth Symposium on the Principles of Programming Languages (POPL)*, p. 207–218. – janvier 1981.

[75] Lafage (Thierry), Seznec (André), Rohou (Erven) et Bodin (François). – *Code Cloning Tracing: A New Approach to Trace Collection*. – Rapport technique n° 1176, IRISA, mars 1998.

[76] Lam (Monica). – Software pipelining: an effective scheduling technique for VLIW machines. *In : Conference on Programming Language Design and Implementation*. ACM SIGPLAN, p. 318–328. – juin 1988.

[77] Larus (James R.) et Schnarr (Eric). – EEL: Machine-independant executable editing. *In : Conference on Programming Language Design and Implementation*. ACM SIGMETRICS, p. 291–300. – juin 1995.

[78] Le Maire (Valérie). – *Optimisation d'un arbre de décision par un algorithme de recuit simulé*. – Rapport technique, Irisa, avril 1988.

[79] Levine (John), Mason (Tony) et Brown (Doug). – *Lex and Yacc.* – 103 Morris Street, Suite A, Sebastopol, CA 95472, O'Reilly and Associates, Inc, 1992.

[80] Mahlke (Scott A.), Lin (David C.), Chen (William Y.), Hank (Richard E.) et Bringmann (Roger A.). – Effective compiler support for predicated execution using the hyperblock. *In : Proceedings of the 25th Annual International Symposium on Microarchitecture.* IEEE Computer Society TC-MICRO and ACM SIGMICRO, p. 45–54. – Portland, Oregon, décembre 1–4, 1992.

[81] McFarling (Scott). – Program optimization for instruction caches. *In : ASPLOS-III*, p. 183–191. – avril 1989.

[82] McFarling (Scott). – Procedure merging with instruction caches. *In : Conference on Programming Language Design and Implementation*, p. 71–79. – juin 1991.

[83] McFarling (Scott). – *Combining branch predictors.* – Rapport technique, DEC, 1993.

[84] Moreno (J. H.), Moudgill (M.), Ebcioğlu (K.), Altman (E.), Hall (C. B.), Miranda (R.), Chen (S.-K.) et Polyak (A.). – Simulation/evaluation environment for a VLIW processor architecture. *IBM Journal of Research and Development*, vol. 41, n° 3, 1997, p. 287–302.

[85] Mowbray (Thomas J.) et Ruh (William A.). – *Inside CORBA: Distributed Object Standards and Applications.* – Reading, MA, USA, Addison-Wesley, 1997, *The Addison-Wesley object technology series*, xxiii + 376p.

[86] Mowry (Todd C.), Lam (Monica S.) et Gupta (Anoop). – Design and evaluation of a compiler algorithm for prefetching. *In : Conference on Architecture Support for Programming Languages and Operating Systems*, p. 62–73. – octobre 1992.

[87] Mueller (Frank) et Whalley (David B.). – Avoiding conditional branches by code replication. *In : Conference on Programming Language Design and Implementation*, éd. par Wall (David W.). p. 56–66. – New York, NY, USA, juin 1995.

[88] Nicolau (Alex), Halambi (Ashok), Dutt (Nikil), Novack (Steve) et Hummel (Joe). – Retaining semantic information for improved code generation. *In : Code Generation for Embedded Processors.* – mars 1998.

[89] Nisbet (Andy). – GAPS : Genetic algorithm optimised parallelisation. *In : Workshop on Compilers for Parallel Computing*, p. 172–182. – Linköping, Suède, juin 1998.

[90] Norris (Cindy) et Pollock (Lori L.). – A scheduler-sensitive global register allocator. *In : Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, éd. par IEEE. p. 804–813. – IEEE Computer Society Press, 1993.

[91] Numerical C Extensions Group of ANSI X3J11. – Restricted pointers, août 1995.

[92] Object Management Group, Inc. – *The Common Object Request Broker : Architecture and Specification*, août 1997, 2.1 édition.

[93] Pettis (Karl) et Hansen (Robert C.). – Profile guided code positioning. *In : Conference on Programming Language Design and Implementation*, p. 16–27. – juin 1990.

[94] Pinter (Shlomit S.). – Register allocation with instruction scheduling: A new approach. *In : Proceedings of the Conference on Programming Language Design*

and Implementation, éd. par Cartwright (Robert). p. 248–257. – ACM Press, juin 1993.

[95] Proebsting (Todd A.) et Fraser (Christopher W.). – Detecting pipeline structural hazards quickly. *In : Annual Symposium on Principles of Programming Languages*, p. 280–286. – janvier 1994.

[96] Rau (B. R.). – Iterative modulo scheduling: An algorithm for software pipelining loops. *In : 27th International Symposium on Microarchitecture*, p. 63–74. – décembre 1994.

[97] Rau (B. R.), Lee (M.), Tirumalai (P. P.) et Schlansker (M. S.). – Register allocation for software pipelined loops. *In : Conference on Programming Language Design and Implementation*, p. 283–299. – 1992.

[98] Rau (B. Ramakrishna), Kathail (Vinod) et Gupta (Shail Aditya). – Machine-description driven compilers for VLIW processors. *In : Code Generation for Embedded Processors*. – mars 1998.

[99] Rau (B.R.) et Glaeser (C.D.). – Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *IEEE*, 1981, p. 183–198.

[100] Rohou (Erven), Bodin (François), Chamski (Zbigniew) et Seznec (André). – SALTO : un système pour la manipulation de code assembleur. *In : Adéquation Algorithme Architecture en traitement du signal et images*. CEA/Leti/CNRS/Isis. – janvier 1998.

[101] Rohou (Erven), Bodin (François) et Seznec (André). – SALTO*: System for Assembly Language Transformation and Optimization*. – Rapport de recherche n° 2980, Inria, septembre 1996.

[102] Rohou (Erven), Bodin (François) et Seznec (André). – SALTO: System for assembly language transformation and optimization. *In : Compilers for Parallel Computers*, éd. par Gerndt (Michael), p. 261–272. – décembre 1996.

[103] Romer (Ted), Voelker (Geoff), Lee (Dennis), Wolman (Alec), Wong (Wayne), Levy (Hank), Bershad (Brian) et Chen (J. Bradley). – Instrumentation and optimization of Win32/Intel executables using etch. *In : The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, éd. par USENIX. p. 1–7. – Berkeley, CA, USA, août 1997.

[104] Ruttenberg (John), Gao (G.), Stoutchinin (A.) et Lichtenstein (W.). – Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. *In : Conference on Programming Language Design and Implementation*. SIGPLAN, p. 1–11. – mai 1996.

[105] Schlansker (Michael), Conte (Thomas M.), Dehnert (James), Ebcioğlu (Kemal), Fang (Jesse Z.) et Thompson (Carol L.). – Theme feature: Compilers for instruction-level parallelism. *Computer*, vol. 30, n° 12, décembre 1997, p. 63–69.

[106] Smith (J.E.). – A study of branch prediction strategies. *In : Proceedings of the 8th Annual International Symposium on Computer Architecture*. – mai 1981.

[107] Smith (Michael D.). – *Tracing with pixie*. – Rapport technique n° CSL-TR-91-497, Stanford University, novembre 1991.

[108] Srivastava (Amitabh) et Eustace (Alan). – ATOM: a system for building customized program analysis tools. *In : Conference on Programming Language Design and Implementation*. SIGPLAN. – 1994.

[109] Srivastava (Amitabh) et Wall (David W.). – A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, vol. 1, n° 1, décembre 1992, p. 1–18.

[110] Stanford Compiler Group. – Suif compiler system. – Disponible à l'adresse `http://suif.stanford.edu`.

[111] Stanford SUIF Compiler Group. – *SUIF: A Parallelizing and Optimizing Research Compiler*. – Rapport technique n° CSL-TR-94-620, Computer Systems Laboratory, Stanford University, mai 1994.

[112] Toburen (Mark C.), Conte (Thomas M.) et Reilly (Matt). – *Instruction Scheduling for Low Power Dissipation in High Performance Microprocessors*. – Rapport technique, North Carolina State University, 1998.

[113] Torrellas (Josep), Xia (Chun) et Daigle (Russel). – Optimizing instruction cache performance for operating system intensive workloads. *In : High Performance Computer Architecture*, p. 360–369. – janvier 1995.

[114] Uhlig (Richard) et Mudge (Trevor). – *Trace-driven Memory Simulation : A Survey*. – Rapport technique, University of Michigan, 1995.

[115] Université de New York. – Trimaran. – Disponible à l'adresse `http://www.trimaran.org`.

[116] Wall (David W.). – Predicting program behavior using real or estimated profiles. *In : Conference on Programming Language Design and Implementation*, p. 59–70. – juin 1991.

[117] Wang (Jian), Eisenbeis (Christine), Jourdan (Martin) et Su (Bogong). – DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, vol. 22, n° 3, juin 1994, p. 351–373.

[118] Weaver (David L.) et Germond (Tom). – *The SPARC Architecture Manual*. – SPARC International, 1994.

[119] Wirth (Niklaus). – A plea for lean software. *Computer*, vol. 28, n° 2, février 1995, p. 64–68.

[120] Yeh (T.-Y.) et Patt (Y.N.). – Two-level adaptive branch prediction. *In : Proceedings of the 24^{th} International Symposium on Microarchitecture*. – novembre 1991.

[121] Young (Cliff) et Smith (Michael D.). – Improving the accuracy of static branch prediction using branch correlation. *In : Conference on Architecture Support for Programming Languages and Operating Systems*. – octobre 1994.

[122] Zhang (Xiaolan), Wang (Zheng), Gloy (Nicholas), Chen (J. Bradley) et Smith (Michael D.). – System support for automatic profiling and optimization. *In : Symposium on Operating Systems Principles*. – octobre 1997.

Annexe A

IL

Cette annexe illustre le langage IL que nous avons développé pour propager de l'information entre les différents modules d'optimisation. Il permet aussi de conserver une carte de correspondance entre les représentations du programme à différents niveaux et de maintenir un historique des transformations appliquées.

Le langage IL est présenté dans le paragraphe 4.3.3. Nous donnons ici la grammaire du langage et un exemple complet de description de la structure d'un programme assembleur.

A.1 Grammaire du langage IL

Fichier `.il`	→	en-tête corps
en-tête	→	def-mots-clés niveau
def-mots-clés	→	`Keywords` = { liste-mots-clés }
liste-mots-clés	→	ε | liste-mots-clés , mot-clé
niveau	→	`DefaultLevel` = niveau-par-défaut
corps	→	liste-propriétés
liste-propriétés	→	ε | propriété liste-propriétés
propriété	→	Référence |Référence . mot-clé := valeur
Reférence	→	#(type-objet niveau id contenu)
type-objet	→	`INST` |`BB` |`SS`
niveau	→	[chaîne] | ε
contenu	→	`,DESC=` { liste-ref } | ε
valeur	→	entier |chaîne |Référence |ensemble |<?>
ensemble	→	{ liste-valeur }
liste-valeur	→	liste-valeur , valeur |valeur

A.2 Exemple complet

A.2.1 Programme Fortran

```
      SUBROUTINE MxM (A,B,C,n)
      REAL*8    A,B,C
      DIMENSION A(n,n)
      DIMENSION B(n,n)
      DIMENSION C(n,n)
      INTEGER i,j,k
      DO i=1,n
         DO j=1,n
            A(i,j) = 0
            DO k=1,n
               A(i,j) = A(i,j) + B(i,k)*C(k,j)
            ENDDO
         ENDDO
      ENDDO
      END
```

A.2.2 Programme assembleur

```
(* This code was generated by MT1 Revision: 1.1 [TM-1 V1.0.3 S] *)
(* at Mon Aug 24 12:19:46 PM METDST 1998. *)

.reserve _mxm.i,4,"bss",4
.reserve _mxm.j,4,"bss",4
.reserve _mxm.k,4,"bss",4

.text

.global _mxm
_mxm:

        .instid 2
        iadd IF r1 r1 r0 -> r37;
        .instid 3
        ld32d IF r1 (0) r8 -> r38;
        .instid 4
        ijmpi IF r1 (_MXM._DT_1);

_MXM._DT_1:

        .instid 6
        igeq IF r1 r0 r38 -> r128;
        .instid 62
        igtr IF r1 r38 r0 -> r129;
```

```
        .instid 7
        ijmpi IF r128 (_MXM._DT_2);
        .instid 8
        iadd IF r129 r1 r0 -> r39;
        .instid 9
        ld32d IF r129 (0) r8 -> r40;
        .instid 10
        ijmpi IF r129 (_MXM._DT_3);

_MXM._DT_3:

        .instid 12
        igeq IF r1 r0 r40 -> r130;
        .instid 64
        igtr IF r1 r40 r0 -> r131;
        .instid 13
        ijmpi IF r130 (_MXM._DT_4);
        .instid 14
        isub IF r131 r37 r1 -> r132;
        .instid 15
        asli IF r131 (2) r132 -> r133;
        .instid 16
        isub IF r131 r39 r1 -> r134;
        .instid 17
        ld32d IF r131 (0) r8 -> r135;
        .instid 18
        asli IF r131 (2) r135 -> r136;
        .instid 19
        imul IF r131 r134 r136 -> r137;
        .instid 20
        iadd IF r131 r133 r137 -> r138;
        .instid 21
        iadd IF r131 r5 r138 -> r139;
        .instid 22
        h_st32d IF r131 (0) r0 r139;
        .instid 23
        iadd IF r131 r1 r0 -> r41;
        .instid 24
        iadd IF r131 r135 r0 -> r42;
        .instid 25
        ijmpi IF r131 (_MXM._DT_5);

_MXM._DT_5:

        .instid 52
        igtr IF r1 r42 r0 -> r140;
        .instid 29
        isub IF r1 r37 r1 -> r141;
        .instid 30
```

```
isub IF r1 r39 r1 -> r142;
.instid 31
ld32d IF r140 (0) r8 -> r143;
.instid 32
imul IF r1 r142 r143 -> r144;
.instid 33
iadd IF r1 r141 r144 -> r145;
.instid 34
ld32x IF r140 r5 r145 -> r146;
.instid 35
isub IF r1 r41 r1 -> r147;
.instid 36
imul IF r1 r147 r143 -> r148;
.instid 37
iadd IF r1 r141 r148 -> r149;
.instid 38
ld32x IF r140 r6 r149 -> r150;
.instid 39
iadd IF r1 r147 r144 -> r151;
.instid 40
ld32x IF r140 r7 r151 -> r152;
.instid 41
fmul IF r1 r150 r152 -> r153;
.instid 42
fadd IF r1 r146 r153 -> r154;
.instid 43
asli IF r1 (2) r141 -> r155;
.instid 44
asli IF r1 (2) r143 -> r156;
.instid 45
imul IF r1 r142 r156 -> r157;
.instid 46
iadd IF r1 r155 r157 -> r158;
.instid 47
iadd IF r1 r5 r158 -> r159;
.instid 48
h_st32d IF r140 (0) r154 r159;
.instid 49
iadd IF r140 r41 r1 -> r41;
.instid 50
isub IF r140 r42 r1 -> r42;
.instid 66
igeq IF r1 r0 r42 -> r160;
.instid 51
ijmpi IF r140 (_MXM._DT_5);
.instid 53
iadd IF r160 r39 r1 -> r39;
.instid 54
isub IF r160 r40 r1 -> r40;
```

```
        .instid 55
        ijmpi IF r160 (_MXM._DT_3);

_MXM._DT_4:

        .instid 57
        iadd IF r1 r37 r1 -> r37;
        .instid 58
        isub IF r1 r38 r1 -> r38;
        .instid 59
        ijmpi IF r1 (_MXM._DT_1);

_MXM._DT_2:

        .instid 61
        ijmpt IF r1 r1 r2;
```

A.2.3 Fichier IL

```
// This code was generated by MT1 Revision: 1.1 [TM-1 V1.0.3 S]
// at Mon Aug 24 12:19:46 PM METDST 1998.

Keywords =  SrcStatements, MemRAWdependsOn, MemWARdependsOn, MemWAWdependsOn,
             ToOptimize, Loop, iteratorRsrc, lowerBoundValue, upperBoundValue,
             strideValue, body, incrCode, loopBack, loopExit, loopBackTest,
             Entry

DefaultLevel = asm

#(BB 1, DESC =  #(INST 2), #(INST 3), #(INST 4) )
#(BB 2, DESC =  #(INST 6), #(INST 62), #(INST 7) )
#(BB 3, DESC =  #(INST 8), #(INST 9), #(INST 10) )
#(BB 4, DESC =  #(INST 12), #(INST 64), #(INST 13) )
#(BB 5, DESC =  #(INST 14), #(INST 15), #(INST 16), #(INST 17), #(INST 18),
  #(INST 19), #(INST 20), #(INST 21), #(INST 22), #(INST 23), #(INST 24),
  #(INST 25) )
#(BB 6, DESC =  #(INST 52), #(INST 29), #(INST 30), #(INST 31), #(INST 32),
  #(INST 33), #(INST 34), #(INST 35), #(INST 36), #(INST 37), #(INST 38),
  #(INST 39), #(INST 40), #(INST 41), #(INST 42), #(INST 43), #(INST 44),
  #(INST 45), #(INST 46), #(INST 47), #(INST 48), #(INST 49), #(INST 50),
  #(INST 66), #(INST 51) )
#(BB 7, DESC =  #(INST 53), #(INST 54), #(INST 55) )
#(BB 8, DESC =  #(INST 57), #(INST 58), #(INST 59) )
#(BB 9, DESC =  #(INST 61) )

#(INST 48).MemWARdependsOn :=  #(INST 34)

// Additional Loop Information
```

```
#(SS 2).Loop := true
#(SS 2).body := #(BB 2) , #(BB 3) , #(BB 4) , #(BB 5) , #(BB 6) , #(BB 7) ,
          #(BB 8)
#(SS 2).Entry := #(BB 2)
#(SS 2).iteratorRsrc := "r37"
#(SS 2).lowerBoundValue := 1
#(SS 2).strideValue := 1
#(SS 2).incrCode :=  #(INST 57)
#(SS 2).loopBack :=  #(INST 59)
#(SS 2).loopExit :=  #(INST 7)
#(SS 2).loopBackTest :=  #(INST 6)

#(SS 3).Loop := true
#(SS 3).body := #(BB 4) , #(BB 5) , #(BB 6) , #(BB 7)
#(SS 3).Entry := #(BB 4)
#(SS 3).iteratorRsrc := "r39"
#(SS 3).lowerBoundValue := 1
#(SS 3).strideValue := 1
#(SS 3).incrCode :=  #(INST 53)
#(SS 3).loopBack :=  #(INST 55)
#(SS 3).loopExit :=  #(INST 13)
#(SS 3).loopBackTest :=  #(INST 12)

#(SS 4).Loop := true
#(SS 4).body := #(BB 6)
#(SS 4).Entry := #(BB 6)
#(SS 4).iteratorRsrc := "r41"
#(SS 4).lowerBoundValue := 1
#(SS 4).strideValue := 1
#(SS 4).incrCode :=  #(INST 49)
#(SS 4).loopBack :=  #(INST 51)
#(SS 4).loopExit :=  #(INST 55)
#(SS 4).loopBackTest :=  #(INST 52)


// Subsection containing *all* basic blocks

#(SS 1, DESC =  #(BB 1), #(SS 2), #(BB 9)  )
#(SS 2, DESC =  #(BB 2), #(BB 3), #(SS 3), #(BB 8)  )
#(SS 3, DESC =  #(BB 4), #(BB 5), #(SS 4), #(BB 7)  )
#(SS 4, DESC =  #(BB 6)  )

#(SS 1).ToOptimize := true
```

Annexe B

Dépliage de boucle

Nous donnons ici le source Fortran des boucles et l'ensemble des chiffres correspondant à l'étude du paragraphe 4.4.2. Chacune des boucles `quan`, `reco`, `mkcenter`, `mkcenter-flat`, `MxM` et `fdct` est dépliée entre 2 et 16 fois. La colonne «Taille» indique la taille du corps de boucle produit en instructions VLIW (ou, de manière équivalente, en cycles). La colonne «Vitesse» est le rapport de la taille de la boucle par le facteur de dépliage s_n. a_n désigne l'accélération globale de la boucle et Δ_n l'accroissement par rapport à l'étape précédente. La dernière colonne indique quelle fraction du gain maximum est atteinte (100 % pour $n = 16$).

Dans chaque tableau, la dernière accélération qui dépasse le seuil des 10 % est indiquée en gras.

B.1 quan

```
      INTEGER FUNCTION quan (val,table,size)
      IMPLICIT NONE

      INTEGER val, size
      INTEGER table(*)

      INTEGER i

      quan = size
      DO i = 0,size-1
         IF (val .LT. table(i+1)) THEN
            quan = i
            GOTO 10
            ENDIF

      END DO

 10   RETURN

      END
```

n	Taille	Vitesse	Δ_n	a_n	
1	19	19,0		1,0	56 %
2	29	14,5	23,7 %	1,3	73 %
3	39	13,0	**10,3 %**	1,5	81 %
4	49	12,3	5,8 %	1,6	86 %
5	59	11,8	3,7 %	1,6	90 %
6	69	11,5	2,5 %	1,7	92 %
7	79	11,3	1,9 %	1,7	94 %
8	89	11,1	1,4 %	1,7	95 %
9	99	11,0	1,1 %	1,7	96 %
10	109	10,9	0,9 %	1,7	97 %
11	119	10,8	0,8 %	1,8	98 %
12	129	10,8	0,6 %	1,8	98 %
13	139	10,7	0,5 %	1,8	99 %
14	149	10,6	0,5 %	1,8	99 %
15	159	10,6	0,4 %	1,8	100 %
16	169	10,6	0,4 %	1,8	100 %

B.2 reco

```
subroutine reco (s, d, om, foo)
INTEGER   d(100), lx2, om(100), s(100)
INTEGER   foo(8)
INTEGER   i1, i2, i4, m, k

lx2 = 15
i1 = 0
i2 = 0
i4 = 0
DO k = 0, 63
   d(i1+1) = s(i4+1) * om(i1+1)
   i1 = i1 + 1
   m = i1 - (i1/8)*8
   i2 = i2 + lx2 * foo(m+1)
   i4 = i2 + m
ENDDO
END
```

n	Taille	Vitesse	Δ_n	a_n	
1	22	22,0		1,0	29 %
2	30	15,0	31,8 %	1,5	43 %
3	33	11,0	26,7 %	2,0	58 %
4	36	9,0	**18,2 %**	2,4	71 %
5	41	8,2	8,9 %	2,7	78 %
6	48	8,0	2,4 %	2,8	80 %
7	54	7,7	3,6 %	2,9	83 %
8	58	7,3	6,0 %	3,0	88 %
9	65	7,2	0,4 %	3,0	88 %
10	69	6,9	4,5 %	3,2	92 %
11	76	6,9	-0,1 %	3,2	92 %
12	80	6,7	3,5 %	3,3	96 %
13	87	6,7	-0,4 %	3,3	95 %
14	91	6,5	2,9 %	3,4	98 %
15	98	6,5	-0,5 %	3,4	98 %
16	102	6,4	2,4 %	3,5	100 %

B.3 mkcenter-flat

```
subroutine make_center(src, dst, width, height)
INTEGER     src(1000), dst(5000)
INTEGER     width, height

INTEGER   srcindex, dstindex, k, i, tmp, edge

edge = height/2
srcindex = 0
dstindex = 0
tmp = width*height-1
DO k = 0, tmp
   i = k - (k/width)*width
   dst(dstindex+i+1) = src(srcindex+i+1)
   IF (i .eq. width-1) THEN
      dstindex = dstindex + width + 2*edge
      srcindex = srcindex + width
   ENDIF
ENDDO
END
```

n	Taille	Vitesse	Δ_n	a_n	
1	43	43,0		1,0	41 %
2	59	29,5	31,4 %	1,5	60 %
3	75	25,0	**15,3 %**	1,7	71 %
4	91	22,8	9,0 %	1,9	78 %
5	107	21,4	5,9 %	2,0	83 %
6	123	20,5	4,2 %	2,1	86 %
7	139	19,9	3,1 %	2,2	89 %
8	155	19,4	2,4 %	2,2	91 %
9	171	19,0	1,9 %	2,3	93 %
10	187	18,7	1,6 %	2,3	95 %
11	203	18,5	1,3 %	2,3	96 %
12	219	18,3	1,1 %	2,4	97 %
13	235	18,1	0,9 %	2,4	98 %
14	251	17,9	0,8 %	2,4	99 %
15	267	17,8	0,7 %	2,4	99 %
16	283	17,7	0,6 %	2,4	100 %

B.4 mkcenter

```
subroutine make_center(src, dst, width, height, edge)
INTEGER     src(1000), dst(5000)
INTEGER     width, height, edge

INTEGER   srcindex, dstindex, i

srcindex = 0
dstindex = 0

   DO i = 0, width-1
      dst(dstindex+i+1) = src(srcindex+i+1)
   ENDDO
END
```

n	Taille	Vitesse	Δ_n	a_n	
1	7	7,0		1,0	21 %
2	7	3,5	50,0 %	2,0	43 %
3	8	2,7	23,8 %	2,6	56 %
4	9	2,3	15,6 %	3,1	67 %
5	10	2,0	**11,1 %**	3,5	75 %
6	13	2,2	-8,3 %	3,2	69 %
7	13	1,9	14,3 %	3,8	81 %
8	14	1,8	5,8 %	4,0	86 %
9	17	1,9	-7,9 %	3,7	79 %
10	18	1,8	4,7 %	3,9	83 %
11	18	1,6	9,1 %	4,3	92 %
12	19	1,6	3,2 %	4,4	95 %
13	20	1,5	2,8 %	4,6	98 %
14	22	1,6	-2,1 %	4,5	95 %
15	23	1,5	2,4 %	4,6	98 %
16	24	1,5	2,2 %	4,7	100 %

B.5 MxM

```
SUBROUTINE MxM (A, B, C, i, n)
DIMENSION A(n*n)
DIMENSION B(n*n)
DIMENSION C(n*n)
real*4    A,B,C
INTEGER i, jk, n, j, k

j = 1
k = 1
DO jk=1, n*n
   A(i*n+j) = A(i*n+j) + B(i*n+k)*C((k-1)*n+j)
   k = k+1
   IF (k .eq. n+1) THEN
      k  = 1
      j = j+1
   ENDIF
ENDDO
END
```

n	Taille	Vitesse	Δ_n	a_n	
1	26	26,0		1,0	16 %
2	27	13,5	92,6 %	1,9	30 %
3	34	11,3	19,1 %	2,3	36 %
4	35	8,8	29,5 %	3,0	46 %
5	36	7,2	**21,5 %**	3,6	56 %
6	43	7,2	0,5 %	3,6	57 %
7	44	6,3	14,0 %	4,1	65 %
8	45	5,6	11,7 %	4,6	72 %
9	46	5,1	**10,1 %**	5,1	79 %
10	53	5,3	-3,6 %	4,9	77 %
11	54	4,9	8,0 %	5,3	83 %
12	55	4,6	7,1 %	5,7	89 %
13	56	4,3	6,4 %	6,0	94 %
14	57	4,1	5,8 %	6,4	100 %
15	64	4,3	-4,6 %	6,1	95 %
16	65	4,1	5,0 %	6,4	100 %

B.6 fdct

```
subroutine fdct_softpipe(block, tmp, i, c)
integer block(*), tmp(*), i, c(*)
integer k, s1, jk, m, v1, v2, v3, j

k = 0
j = -1
s1 = 0
index = 0
v2 = 8*i
DO jk = 0, 63
   m = jk - (jk/8)*8
   IF (m .eq. 0) THEN
      v3 = 1
   ELSE
      v3 = 0
   ENDIF
   j = j + v3
   s1 = s1 - s1*v3
   v1 = c(index+1)
   index = index + 1
   s1 = s1 + v1 * block(v2+m+1)
   tmp(v2+j+1) = s1
ENDDO
END
```

n	Taille	Vitesse	Δ_n	a_n	
1	24	24,0		1,0	34 %
2	29	14,5	65,5 %	1,7	56 %
3	36	12,0	20,8 %	2,0	68 %
4	46	11,5	4,3 %	2,1	71 %
5	52	10,4	**10,6 %**	2,3	79 %
6	57	9,5	9,5 %	2,5	86 %
7	67	9,6	-0,7 %	2,5	86 %
8	74	9,3	3,5 %	2,6	89 %
9	80	8,9	4,1 %	2,7	92 %
10	87	8,7	2,2 %	2,8	94 %
11	97	8,8	-1,3 %	2,7	93 %
12	102	8,5	3,7 %	2,8	96 %
13	111	8,5	-0,5 %	2,8	96 %
14	118	8,4	1,3 %	2,8	97 %
15	124	8,3	2,0 %	2,9	99 %
16	131	8,2	1,0 %	2,9	100 %

Annexe C

Dépliage et blocage

Cette annexe présente les codes Fortran utilisés dans notre exemple de compilation itérative du paragraphe 4.4.3. Les trois programmes Fortran correspondent au produit de matrices original, après blocage, et après blocage et dépliage.

C.1 Code original

```
      SUBROUTINE MxM (A,B,C,n)

      DO j=1, n
         DO k=1, n
            DO ii=1, n
               R1 = a(ii, j)
               R1 = R1 + b(ii, k)*c(k, j)
               a(ii, j) = R1
            ENDDO
         ENDDO
      ENDDO
      END
```

C.2 Code bloqué

```
      SUBROUTINE MxM (A,B,C,n)

      DO j=1, n, 7
         DO k=1, n, 7
            IF ((j + 7 .le. n) .AND. (k + 7 .le. n)) THEN
               DO ii=1, n
                  DO jj=j, j+6
                     R1 = a(ii + 0,jj + 0)
                     DO kk=k,k + 6
                        R1 = R1 + b(ii + 0,kk)*c(kk,jj + 0)
                     ENDDO
                     a(ii + 0,jj + 0) = R1
                  ENDDO
               ENDDO
            ELSE
               ...
            ENDIF
         ENDDO
      ENDDO
      END
```

C.3 Code bloqué et déplié

```
      SUBROUTINE MxM (A,B,C,n)

      DO j=1, n, 7
         DO k=1, n, 7
            IF ((j + 7 .le. n) .AND. (k + 7 .le. n)) THEN
               DO ii=1, n-2, 2
                  DO jj=j, j + 6-2, 2
                     R1 = a(ii + 0,jj + 0)
                     R2 = a(ii + 1,jj + 0)
                     R3 = a(ii + 0,jj + 1)
                     R4 = a(ii + 1,jj + 1)
                     DO kk=k,k + 6
                        R1 = R1 + b(ii + 0,kk)*c(kk,jj + 0)
                        R2 = R2 + b(ii + 1,kk)*c(kk,jj + 0)
                        R3 = R3 + b(ii + 0,kk)*c(kk,jj + 1)
                        R4 = R4 + b(ii + 1,kk)*c(kk,jj + 1)
                     ENDDO
                     a(ii + 0,jj + 0) = R1
                     a(ii + 1,jj + 0) = R2
                     a(ii + 0,jj + 1) = R3
                     a(ii + 1,jj + 1) = R4
                  ENDDO
                  DO jj=jj, j+6
                     R1 = a(ii + 0,jj )
                     R2 = a(ii + 1,jj )
                     DO kk=k, k+6
                        R1 =  R1 + b(ii + 0,kk)*c(kk,jj)
                        R2 =  R2 + b(ii + 1,kk)*c(kk,jj)
                     ENDDO
                     a(ii + 0,jj ) = R1
                     a(ii + 1,jj ) = R2
                  ENDDO
               ENDDO
               DO ii=ii, n
                  DO jj=j, j+6
                     R1 = a(ii,jj)
                     DO kk=k, k+6
                        R1 = R1+ b(ii,kk)*c(kk,jj)
                     ENDDO
                     a(ii,jj) = R1
                  ENDDO
               ENDDO
            ELSE
               ...
            ENDIF
         ENDDO
      ENDDO
      END
```

Index

Résumé

La complexité croissante des processeurs a conduit au développement d'un grand nombre de transformations de code pour adapter l'organisation des calculs à l'architecture matérielle. La difficulté majeure à laquelle est confronté un compilateur consiste à déterminer la séquence de transformations qui va fournir la meilleure performance. Cette séquence dépend de l'application et du processeur considérés. L'interaction profonde entre les diverses transformations de code ne permet pas de trouver une solution statique.

Nous proposons une approche itérative de la compilation pour résoudre ce problème : chaque module d'optimisation peut remettre en cause les décisions prises par un autre module. Ces modules peuvent se communiquer des informations sur les propriétés du code qu'ils ont produit. Cette approche nécessite une refonte complète de la structure des compilateurs actuels.

La réalisation n'a été rendue possible que grâce aux infrastructures logicielles que nous avons développées : SALTO et SEA. Grâce à ces environnements nous avons pu développer rapidement des prototypes de stratégies de compilation.

Nous montrons aussi que l'analyse et l'optimisation ne doivent pas se contenter d'un comportement local à un fragment de code. Au contraire, le comportement global de l'application doit être considéré, en particulier pour les systèmes enfouis.