Méthodologie d'optimisation de programmes pour architectures complexes de processeurs

David Parello

UNIVERSITÉ DE PARIS-SUD
U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ DE PARIS XI ORSAY
SPÉCIALITÉ : *Informatique*

par

DAVID PARELLO

Sujet :

Méthodologie d'optimisation de programmes pour architectures complexes de processeurs

soutenue le vendredi 17 septembre 2004 devant le jury composé de :

M. OLIVIER TEMAM	Université de Paris-Sud	*Directeur*
M. FRANÇOIS BODIN	Université de Rennes 1	*Rapporteur*
M. PASCAL SAINRAT	Université Paul Sabatier, Toulouse III	*Rapporteur*
M. ALBERT COHEN	INRIA	*Examinateur*
M. JEAN-MARIE VERDUN	HP France	*Examinateur*

Remerciements

Je tiens à adresser mes remerciements à François Bodin et Pascal Sainrat pour avoir accepté d'être rapporteurs. Je remercie également les autres membres du jury : Albert Cohen, Jean-Marie Verdun et Olivier Temam.

Je remercie à nouveau Albert Cohen pour les précieuses connaissances dans le domaine de la compilation qu'il dissipe autour de lui. Je remercie à nouveau Jean-Marie Verdun pour sa disponibilité et les précieuses informations et connaissances qu'il m'a apportées. Je remercie Olivier Temam pour m'avoir enseigné non seulement la *recherche* mais également l'art de partager, diffuser et présenter son travail.

Je remercie les membres de l'équipe Avant-Vente de HP France pour m'avoir chaleureusement accueilli. Je remercie également tous les membres de l'équipe Alchemy pour leur convivialité.

Je remercie ma famille et mes amis pour m'avoir soutenu pendant la thèse. Je remercie tout particulièrement mon amie Anne-Lise.

Résumé

L'augmentation de la complexité des processeurs rend de plus en plus difficile l'intégration de modèles précis d'architectures dans les compilateurs. En conséquence, l'efficacité des compilateurs statiques décroît. Actuellement, les compilateurs statiques sont enrichis d'informations dynamiques sur le comportement de l'architecture à la manière des techniques d'optimisation basées sur les profils d'exécutions ou des techniques de re-compilation dynamique. Malheureusement, seules quelques informations élémentaires sur le comportement de l'architecture sont utilisées.

Dans cette thèse, nous montrons de quelle manière les interactions entre les différents composants d'une architecture rendent complexe le comportement des programmes et nous montrons qu'il est possible de capturer cette complexité pour en déduire les transformations à apporter aux programmes. Nous avons étudié une méthode plus systématique pour adresser le problème de la complexité. Nous proposons un processus itératif d'optimisation manuelle basé sur une analyse dynamique détaillée. Nous montrons expérimentalement l'efficacité de ce processus.

Cette approche présente potentiellement une stratégie pour guider de futurs environnements d'optimisation itératifs et propose, dans l'immédiat, un processus d'optimisation manuelle systématique pouvant être utilisé par des ingénieurs ou des chercheurs.

Table des matières

Introduction

L'augmentation de la performance des processeurs est due d'une part à l'avancée des techniques de gravure qui permettent d'accroître la vitesse de fonctionnement des processeurs et d'autre part à l'évolution de l'architecture qui intègre de nombreux mécanismes complexes. Ces mécanismes, comme par exemple la prédiction de branchement, l'exécution dans le désordre et la hiérarchie mémoire, ont pour objectif d'accélérer l'exécution du programme en exploitant au mieux le parallélisme des instructions et la localité des données. Les optimisations modifient les programmes afin qu'ils profitent pleinement de ces différents mécanismes.

Quel que soit le domaine, les exécutions des applications sont soumises à des contraintes donnant à l'optimisation de programmes toute son importance. Dans le domaine du calcul scientifique, les applications de modélisation et de simulation numériques réalisent de grands volumes de calculs. Aussi bien le secteur de la recherche que le secteur industriel ont besoin de modèles de plus en plus précis et de plus en plus grands, réclamant sans cesse d'importantes ressources de calculs. L'optimisation de programmes est donc cruciale pour ces applications afin de leur permettre d'utiliser pleinement ces ressources. La principale contrainte est donc le temps d'exécution. Les applications embarquées ont à la fois des contraintes de temps, des contraintes sur l'espace mémoire et des contraintes sur la consommation d'énergie. Ce domaine a également des besoins croissants notamment avec le rôle croissant des applications de traitement de l'image et du son. La recherche d'une utilisation optimale des ressources est donc primordiale dans le domaine de l'embarqué.

L'optimisation de programmes

Même si les contraintes de chaque domaine d'application sont différentes, les techniques d'optimisation sont très similaires. L'optimisation de programmes peut se faire de manière automatique ou manuelle.

L'optimisation automatique Elle est généralement réalisée par un compilateur ou bien un pré-processeur. Dans chacun des cas, elle consiste à effectuer une longue séquence de transformations parmi lesquelles certaines sont parfois répétées. A chaque étape de cette séquence, la phase d'optimisation utilise une analyse statique de la représentation intermédiaire dans le but de décider l'application ou non de la transformation et si oui avec quels paramètres. Les analyses statiques se basent sur des modèles de l'architecture et du programme pour en deviner leur comportement. Les optimisations dirigées par l'analyse dynamique apportent des informations supplémentaires que l'analyse statique ne peut pas obtenir pour appréhender plus précisément

le comportement des programmes.

L'optimisation manuelle Elle est généralement réalisée lorsque l'optimisation du compilateur n'est pas satisfaisante. Un utilisateur peut transformer le programme source ou bien intervenir sur les optimisations du compilateur au travers des options de compilations. Son champ d'actions est très vaste : il peut en effet modifier l'algorithme, les boucles et également apporter des modifications qui peuvent changer le comportement des optimisations du compilateur (par exemple, il peut agir sur la génération de code). Le programmeur peut utiliser des informations sur le comportement du programme lors de son exécution pour choisir les optimisations. Par l'exemple, il peut utiliser des outils de profilage. Généralement, l'optimisation manuelle est une tâche difficile qui est réservée à un petit nombre d'experts ayant une bonne connaissance de l'architecture du compilateur et des transformations de programmes.

Problématique

Les mécanismes, comme la prédiction de branchement, l'exécution dans le désordre et la hiérarchie mémoire augmentent la performance *crête* des architectures. Le comportement de ces mécanismes évolue durant l'exécution des programmes générant parfois des phénomènes complexes ayant des effets néfastes sur les performances. Par exemple, des conflits de mémoire cache peuvent fortement dégrader les performances. Il est difficile de modéliser ces phénomènes et leurs interactions et donc de comprendre précisément le comportement des programmes sur les architectures. Par conséquent, la compréhension du comportement devient de moins en moins précise, empêchant ainsi l'optimisation automatique comme manuelle de maintenir des *performances soutenues* proches de la *performance crête* des processeurs.

Contribution et Organisation

Dans le premier chapitre, nous présentons l'architecture du processeur Alpha-21264 que nous avons utilisé comme cible pour l'ensemble de nos études. Nous illustrons également les mécanismes complexes des processeurs superscalaires et détaillons des particularités d'exécution qui sont rarement prises en compte lors de l'optimisation de programmes. Ensuite, nous examinons l'optimisation de programmes.

Le second chapitre présente l'étude de l'optimisation d'un noyau de calcul sur le processeur Alpha-21264. Cette étude montre la complexité des phénomènes se produisant au sein d'une architecture superscalaire lors de l'exécution d'un programme. Elle montre que certains problèmes de performance peuvent être mis en évidence uniquement par une analyse dynamique détaillée et suggère donc l'adoption d'une telle analyse pour guider un processus d'optimisation efficace.

Le troisième chapitre présente un processus systématique d'optimisation manuelle basé sur l'analyse dynamique détaillée. L'identification des problèmes de performances basée sur une analyse dynamique détaillée est systématique et formalisée par un arbre de décision. Ce dernier permet de formaliser l'expérience empirique de l'optimisation manuelle. La nature itérative du processus permet de construire des séquences d'optimisations et permet ainsi d'adresser plusieurs

problèmes de performances. L'application de ce processus à un ensemble de programmes de la suite SpecFP2000 montre son efficacité.

Finalement, nous présentons les conclusions et les perspectives de ces travaux.

En annexe, nous présentons les journaux d'optimisation de 3 des 11 programmes optimisés. Ces journaux détaillent l'optimisation itérative réalisée en utilisant le processus d'optimisation présenté dans le chapitre 3.

Chapitre 1

Architecture et Optimisation

Nous avons brièvement présenté l'optimisation manuelle et l'optimisation automatique. Dans la section 1.1, nous décrivons l'architecture d'un processeur superscalaire complexe. La section 1.2 présente l'optimisation de programmes. Elle se concentre plus particulièrement sur les méthodes de sélection des optimisations.

1.1 Complexité des architectures superscalaires

Les processeurs superscalaires possèdent des mécanismes complexes pour exploiter le parallélisme d'instructions. Ces mécanismes sont capables, d'une part d'exécuter des instructions indépendantes dans le désordre pour masquer les latences et d'autre part de prédire le comportement des instructions de branchement pour exécuter spéculativement de longues séquences d'instructions et enfin ils possèdent une hiérarchie mémoire complexe pour exploiter dynamiquement la réutilisation des données et des instructions.

Le processeur Alpha-21264 est un processeur superscalaire à exécution dans le désordre possédant de tels mécanismes ; nous l'avons utilisé comme architecture cible pour nos expériences durant cette thèse. Nous donnons une description détaillée de son architecture dans les sous-sections suivantes.

1.1.1 Architecture générale du processeur Alpha-21264 (EV68)

L'architecture générale du processeur Alpha-21264 est décrite par la figure 1.1. Les composants *Ebox* et *Fbox* contiennent respectivement 4 unités fonctionnelles entières et 2 unités fonctionnelles flottantes. Le composant *Ibox* contient les différents mécanismes relatifs au *pipeline*, il est capable de charger 4 instructions depuis le cache d'instructions et de lancer jusqu'à 6 instructions vers les unités fonctionnelles à chaque cycle. Il gère l'exécution dans le désordre des instructions. Les caches d'instructions (*Icache*) et de données (*Dcache*) ont chacun une taille de 64 Koctets, une associativité d'ordre 2 et des lignes de 64 octets. Le second niveau de cache a également des lignes de 64 octets mais il a une taille de 8 Moctets et il est à correspondance directe. Le composant *Mbox* contrôle les accès au cache de données et assure une exécution correcte de toutes les opérations mémoires. Il contient une queue des opérations de lecture et une queue des opérations d'écriture de 32 entrées chacune (*Load/Store queue*), une table des adresses

des opérations mémoires en échec (*MAF Miss Address File*) et un cache de translation d'adresses (TLB) de 128 entrées complètement associatif mémorisant les adresses des pages dont la taille est de 8 Koctets ou des adresses de groupes de 8, de 64 ou de 512 pages de 8Koctets. Les registres du processeur sont dupliqués : 2 x 80 registres entiers et 2 x 72 registres flottants.

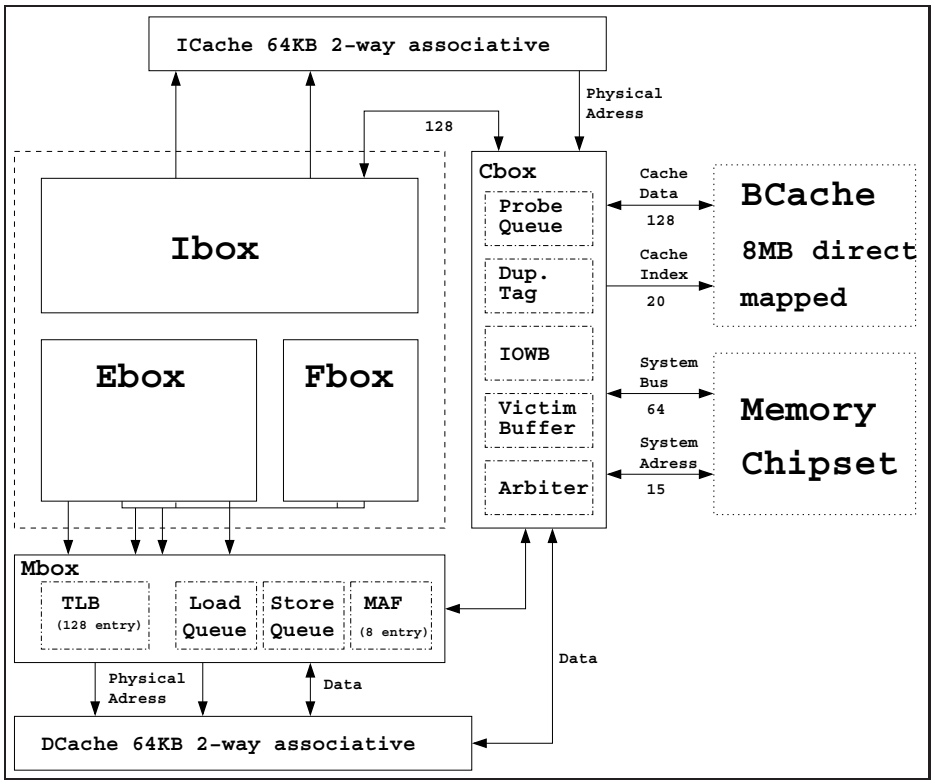


Fig. 1.1 – *Architecture générale du processeur Alpha-21264/EV68*

1.1.2 Pipeline du processeur Alpha-21264 (EV68)

Le pipeline du processeur comporte 8 étages qui sont représentés sur la figure 1.2 et dont le fonctionnement est décrit dans les paragraphes suivants.

Etage 0, Chargement des instructions (*fetch*) Le prédicteur de branchement combine un prédicteur local et un prédicteur global en utilisant un prédicteur de sélection.

L'étage charge jusqu'à 4 instructions alignées depuis le cache d'instructions. Les tables de prédiction sont accédées durant ce cycle. S'il y a plus d'un branchement parmi les instructions chargées et que le premier branchement est prédit non pris alors le prédicteur de branchement prédit les branchements suivants à raison d'un par cycle.

Chaque ligne du cache d'instructions contient un champ de prédiction de ligne. Ce champ indique le prochain numéro de ligne du cache d'instructions à charger. Même si les tables de prédictions de branchement sont accédées durant ce cycle, le résultat de la prédiction n'est connu qu'au cycle suivant. Le champ de prédiction de ligne permet donc d'éviter l'insertion d'une bulle dans le pipeline à chaque branchement. Le champ de prédiction de ligne pointe initialement vers

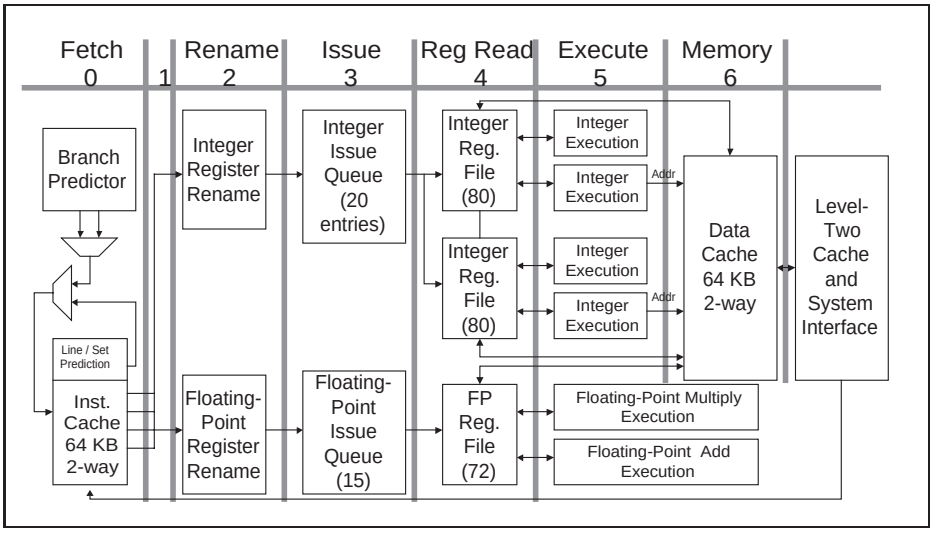


Fig. 1.2 – *Pipeline du processeur Alpha-21264/EV68*

la ligne suivante, lorsqu'un branchement est pris, le champ est mis à jour.

Etage 1, Placement des instructions (*slot*) A ce cycle, le résultat de la prédiction de branchement est disponible et la prédiction de ligne peut être vérifiée.

Les instructions chargées au cycle précédent sont distribuées sur les pipelines (flottant et entier).

Etage 2, Renommage des registres (*rename*) Durant ce cycle, le renommage de registres est effectué. Les instructions se voient attribuer un numéro unique (*inum*) d'identification pour la durée pendant laquelle elles seront en vol dans le pipeline. Ces numéros permettent d'identifier les instructions et l'ordre du programme. Les instructions sont considérées comme étant "en vol" (*inflight*) entre l'étage de renommage et l'étage de terminaison des instructions. Les instructions sont placées dans les queues d'instructions flottante et entière.

Etage 3, Lancement des instructions (*issue*) La queue de lancement entière peut lancer jusqu'à 4 instructions par cycle et la queue de lancement flottante peut lancer jusqu'à 2 instructions par cycle. Les instructions ne sont supprimées des queues que 2 cycles après leur lancement.

Etage 4, Lecture des registres (*reg*) Les instructions, qui viennent d'être lancées, lisent leurs opérandes dans les bancs de registres et reçoivent les renvois anticipés (*bypass*).

Etage 5, Exécution des instructions (*execute*) Les unités fonctionnelles commencent les calculs, l'addition et la multiplication flottantes ont une latence de 4 cycles, la division flottante a une latence entre 9 et 15 cycles et la racine carrée a une latence entre 15 et 33 cycles. La plupart des opérations arithmétiques entières ont une latence de 1 cycle sauf la multiplication entière qui a une latence de 7 cycles.

Etage 6, Accès mémoires (*memory*) Dans cet étage, les opérations mémoires accèdent au cache de données et au cache de traduction d'adresses (TLB). Les opérations de lectures accèdent aux tables d'étiquettes et de données alors que les opérations d'écriture n'accèdent qu'à la table d'étiquettes, la donnée est écrite dans la queue des instructions d'écriture et ne sera écrite dans le cache que lorsque l'instruction sera retirée.

Etage 7 (non représenté), Terminaison des instructions (*retire*) Dans cet étage, les instructions sont retirées dans l'ordre du programme. Le processeur peut retirer jusqu'à 11 instructions par cycles.

1.1.3 Cas particuliers d'exécutions

Lancement spéculatif des instructions

Le pipeline possède des mécanismes de renvois anticipés (*bypass*). Une instruction $i2$ dépendante d'une instruction $i1$ est donc lancée dans l'étage de lecture des registres pendant le cycle précédant la fin de l'exécution de l'instruction $i1$. Lorsqu'une instruction $i2$ dépend d'une instruction de lecture mémoire $i1$, elle est lancée spéculativement : si $i1$ fait un succès dans le cache alors l'instruction $i2$ est retirée de la queue de lancement 2 cycles après son lancement. Si $i1$ fait un défaut de cache alors l'instruction $i2$ est annulée et reste dans la queue de lancement depuis laquelle elle sera relancée.

L'exécution des instructions de la queue de lancement entière est annulée si celles-ci ont été lancées dans la fenêtre de spéculation d'une opération de lecture entière, même si elles ne dépendent pas de l'instruction de lecture. Cette fenêtre de spéculation est de 2 cycles, la table 1.1 montre la fenêtre de spéculation entière dans laquelle deux instructions $i2$ et $i3$ sont lancées. Un compteur, fonctionnant en mode saturé, est incrémenté chaque fois qu'une opération de lecture entière fait un succès dans le cache et est décrémenté à chaque défaut. Lorsque le bit de poids fort de ce compteur est à zéro, la latence des lectures entières passe de 3 à 5 cycles et la fenêtre de spéculation est ainsi supprimée. Les opérations de préchargement dans le registre $R31$ ne créent pas de fenêtre de spéculation.

Cycles	1	2	3	4	5	6	7	8
i1 (lecture entière)	Q	R	E	D	B			
i2				Q	R			
i3					Q			

Symboles	Descriptions
Q	Queue de lancement
R	Lecture des registres
E	Exécution
D	Accès au cache
B	Accès au bus

Tab. 1.1 – *Fenêtre de spéculation entière : cycles 4 à 5*

La fenêtre de spéculation des instructions flottantes n'est que d'un cycle. Si une instruction $i1$ se trouve dans la fenêtre de spéculation d'une opération de lecture flottante faisant un défaut de cache alors cette instruction ($i1$) est annulée uniquement si elle dépend de l'opération mémoire. La table 1.2 montre la fenêtre de spéculation flottante de 1 cycle dans laquelle l'instruction $i2$ est lancée.

Cycles	1	2	3	4	5	6	7	8
i1 (lecture flottante)	Q	R	E	D	B			
i2					Q			
i3								

Symboles	Descriptions
Q	Queue de lancement
R	Lecture des registres
E	Exécution
D	Accès au cache
B	Accès au bus

Tab. 1.2 – *Fenêtre de spéculation flottante : cycles 5*

Exécution des opérations mémoires

Il existe des situations dans lesquelles l'exécution d'une opération mémoire nécessite d'être retardée. Ces situations dépendent des adresses des opérations mémoires, elles ne sont donc détectées qu'au moment de l'exécution. Lorsqu'une telle situation est détectée, l'opération mémoire responsable de la situation et toutes les instructions plus jeunes sont supprimées du pipeline et rechargées depuis l'étage de chargement d'instructions. Ce mécanisme est appelé un *replay trap*.

L'exécution des instructions se fait dans le désordre mais l'architecture garantit une cohérence mémoire et maintient un ordre d'exécution entre certaines opérations mémoires. L'ordre d'exécution est maintenu dans les deux cas suivants :

- lorsque les opérations mémoires s'effectuent sur la même adresse (quelles que soient les opérations),
- lorsque les opérations mémoires sont des opérations d'écriture sur des adresses différentes.

Les entrées dans les queues des opérations de lecture et d'écriture sont allouées dans l'ordre du programme. Ces queues sont des tampons de ré-ordonnancement (*reorder buffers*) pour les opérations mémoires.

Opérations mémoires ordonnées Lorsqu'une opération mémoire est exécutée, son adresse est calculée et comparée avec les adresses des opérations se trouvant dans les queues des opérations de lecture et d'écriture. Si une opération mémoire est en cours sur la même adresse, l'exécution est annulée à partir de l'opération la plus jeune. Dans le cas où l'adresse d'une opération d'écriture est identique à l'adresse d'une opération de lecture plus jeune, se trouvant dans la queue de lectures, l'opération de lecture est annulée. Cet événement est appelé une purge d'ordonnancement d'écriture-lecture (*store-load order trap*). Pour éviter que cet événement ne se reproduise trop souvent, une table de 1024 entrées indexée par le compteur de programme, mémorise dans 1 bit si cet événement s'est déjà produit. Si une opération de lecture se trouve dans la queue de lancement et que le bit correspondant est à 1 alors le lancement de l'opération est retardé jusqu'à ce que toutes les opérations mémoires d'écriture plus anciennes soient lancées. La table des bits d'attente est remise à zéro tous les 16384 cycles.

Opérations mémoires en conflits La mémoire cache de données ne peut garantir l'exécution correcte d'une opération de lecture et d'écriture dans le même ensemble. Toute nouvelle opération mémoire qui entre en conflit avec une opération dans les queues des opérations mémoires est relancée.

Opérations mémoires et ressources Le mécanisme de purge des opérations mémoires est utilisé pour relancer les opérations lorsque les queues des opérations mémoires sont pleines ou

lorsque le tampon des opérations mémoires en échec est plein.

1.2 Optimisation de programmes

Pour réaliser une optimisation de programmes, il est nécessaire de réaliser trois étapes :
- **sélectionner** une transformation et la partie de programme sur laquelle il faut l'appliquer,
- **vérifier** que la transformation est légale, c'est-à-dire qu'elle ne modifie pas la sémantique du programme,
- et enfin **transformer** le programme.

La sous-section suivante présente un aperçu des transformations que l'on peut appliquer aux programmes.

1.2.1 Transformations de programmes

Les transformations peuvent être appliquées à différents niveaux dans le programme :

- Au niveau de l'*instruction du langage source* (*statement*) : les transformations n'affectent que les instructions du langage machine correspondant à l'instruction du langage source.
- Au niveau du *bloc de base* : les transformations affectent uniquement les instructions d'un même bloc de base (un bloc de base est formé d'instructions parmi lesquelles il n'y a qu'une seule instruction de contrôle.)
- Au niveau des *nids de boucles* : les transformations affectent les instructions appartenant au nid de boucles.
- Au niveau des *procédures* : les transformations affectent toutes les instructions d'une procédure. Ces transformations sont aussi appelées des transformations *globales* ou *intra-procédurales*.
- Au niveau du *programme complet* : les transformations affectent plusieurs procédures ; elles sont appelées transformations *inter-procédurales*.

Plus le niveau d'une transformation est élevé, comme c'est le cas pour les transformations inter-procédurales, plus les trois étapes nécessaires pour réaliser la transformation (sélection, vérification et transformation) sont difficiles à mettre en œuvre. En revanche, plus le niveau d'une transformation est élevé et plus ses gains de performance sont potentiellement importants.

Des ouvrages de références présentent les transformations de programmes de manière détaillée et exhaustive [AhSeUl86, Mu97, AlKe01]. Bacon et al. [BaGrSh94] présentent les bénéfices et les défauts de nombreuses transformations au niveau des nids de boucles. Les différents niveaux d'applications des transformations peuvent servir de classement. Cependant de plus en plus de transformations sont réalisées à plusieurs niveaux, par exemple la propagation de constantes est réalisée au niveau des blocs de bases et au niveau inter-procédurale. Le classement n'a donc plus un intérêt majeur.

Les optimisations peuvent être groupées en 3 grandes catégories :

Les transformations de simplifications Elles sont généralement assez indépendantes des autres optimisations et leur effet sur la performance est le plus souvent positif. Il est donc possible de sélectionner ces optimisations de manière systématique pour l'ensemble du programme. Parmi

ces transformations de programmes, on trouve la propagation de constantes, le remplacement de sous-expressions, les simplifications algébriques, l'élimination de code mort, la factorisation des invariants de boucles. Muchnick [Mu97] décrit de manière très détaillée toutes ces transformations.

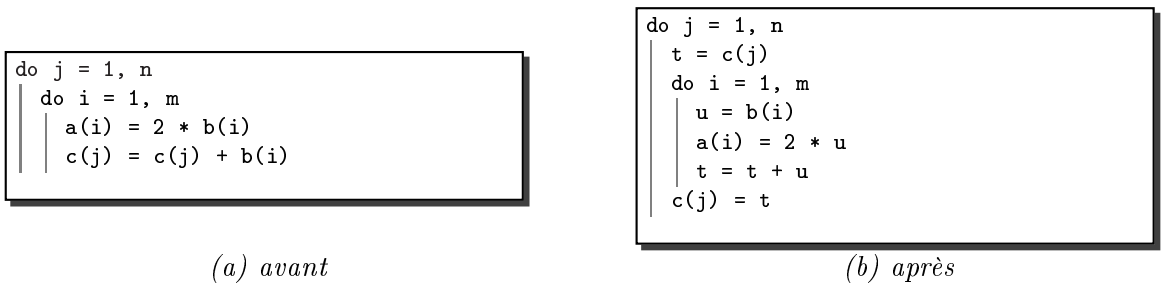
Les transformations spécifiques à la génération de code L'allocation de registres et le réordonnancement d'instructions sont deux optimisations qui ne doivent être effectuées qu'une fois les simplifications faites et la structure du programme fixée.

L'allocation de registres détermine les variables temporaires du programme qui vont être mémorisées dans les registres logiques de la machine pour minimiser le nombre des instructions de lecture et des instructions d'écriture mémoire. Lorsqu'il n'y a plus suffisamment de registres disponibles, les variables temporaires sont mémorisées dans la pile par du code de débordement (*Spill code*) [BrCoKe89].

Le réordonnancement d'instructions est une optimisation uniquement dédiée au parallélisme d'instructions. Elle réordonne les instructions pour maximiser l'utilisation des ressources du processeur. Les techniques basées sur le réordonnancement par liste (*list scheduling*) [GiMu86] sont très efficaces pour optimiser l'ordre des instructions dans un bloc de base.

L'allocation de registres et le réordonnancement des instructions sont corrélés. Le réordonnancement des instructions modifie la durée de vie des registres et a un impact direct sur l'allocation des registres [Pi93].

La promotion de scalaires n'est pas une transformation spécifique à la génération de code, cependant elle est fortement liée à l'allocation de registres. Elle consiste à identifier les multiples accès à une même référence mémoire pour placer la valeur de celle-ci dans un registre. La figure 1.3 montre la promotion des scalaires `b(i)` et `c(j)`.



(a) avant *(b) après*

Fig. 1.3 – *Promotion de scalaires*

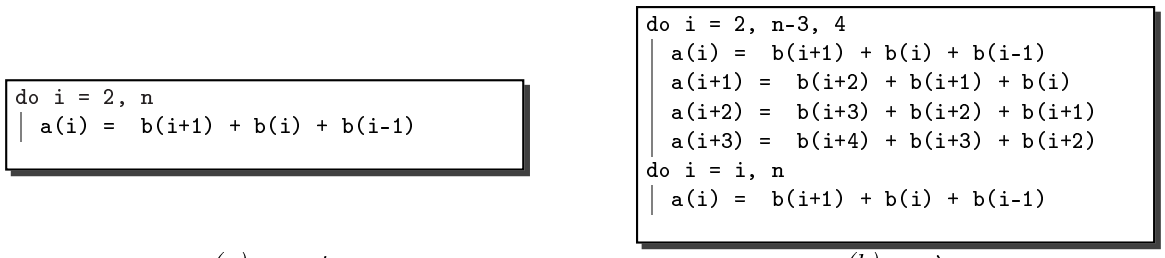
La promotion des scalaires augmente le nombre de registres utilisés, par conséquent, promouvoir agressivement les scalaires peut générer du code de débordement et peut ainsi dégrader les performances. La promotion des scalaires [CaKe94] doit donc tenir compte de la *pression* qu'elle exerce sur les registres.

Les transformations spécifiques aux structures de contrôles et aux structures de données Les transformations faisant partie de ce groupe ont généralement un impact important sur les optimisations des deux groupes précédents et plus particulièrement sur l'allocation de registres et le réordonnancement des instructions. Ce groupe est formé des transformations

relatives à la structure de procédure, celles relatives aux nids de boucles et celles relatives aux structures de données.

Expansion de code Cette transformation consiste à remplacer un appel à une procédure par le code de celle-ci. Cette transformation est généralement réalisée lorsque le corps de la procédure est de petite taille et si l'appel est souvent exécuté. La taille de la procédure peut être connue statiquement, en revanche, le nombre d'exécutions de l'appel peut dépendre du jeu de données.

Déroulage de boucles Dérouler une boucle k fois consiste à recopier le corps de la boucle k fois et multiplier le pas de l'itérateur par k. La transformation réduit le nombre d'instructions exécutées relatives au branchement et éventuellement le nombre d'instructions exécutées relatives aux calculs d'adresse des références mémoires. Cette instruction a un impact important sur le réordonnancement des instructions et l'allocation de registres. Dans le cas idéal, c'est-à-dire lorsqu'il n'y a pas de dépendances entre les itérations, le déroulage de boucle permet au réordonnancement d'instructions de trouver plus de parallélisme. Ce réordonnancement augmente le nombre de variables temporaires en vie simultanément ce qui peut permettre de profiter des registres potentiellement disponibles mais cela peut également obliger l'allocation de registres à générer du code de débordement. Dans le même temps, le déroulage de boucles augmente la taille du programme, cela peut avoir un impact sur le nombre de défauts du cache d'instructions. Le déroulage de boucles peut aussi favoriser la promotion de scalaires, par exemple dans la boucle déroulée de la figure 1.4, les références `b(i)`, `b(i+1)`, `b(i+2)` et `b(i+3)` peuvent profiter de la promotion de scalaires. Le nombre d'itérations déroulées doit donc être judicieusement choisi afin d'obtenir le meilleur compromis entre parallélisme, utilisation des registres, réduction des instructions exécutées et défauts du cache d'instructions.



(a) avant *(b) après*

Fig. 1.4 – *Déroulage de boucles*

Permutation de boucles La permutation de boucles consiste à permuter deux boucles appartenant à un même nid. Cette optimisation est généralement utilisée pour améliorer la localité mais ses effets sont variés. Considérons l'exemple de la figure 1.5, supposons que le tableau `b` est rangé en mémoire par colonne. La localité spatiale des références au tableau `b` est mal exploitée. La permutation de boucles permet de l'améliorer. Cet exemple, montre également que la permutation à un impact sur la promotion des scalaires et sur le parallélisme d'instructions. Supposons cette fois-ci que le code original est le code de droite *(b)* sur la figure, la permutation de boucles favorise la promotion des scalaires du tableau `a`. En revanche, cette permutation rend toutes les itérations de la boucle interne `i` dépendantes. Sur un exemple aussi simple, il n'est pas

facile de sélectionner une permutation d'autant plus que le comportement du programme dépend des paramètres `m,n` et des tailles des tableaux `a` et `b`.

```
do j = 1, n
| do i = 1, m
| | a(j) = a(j) + b(j,i)
```

(a) avant

```
do i = 1, m
| do j = 1, n
| | a(j) = a(j) + b(j,i)
```

(b) après

Fig. 1.5 – *Permutation de boucles*

Fusion et fission de boucles La fusion et la fission de boucles sont souvent utilisées pour améliorer la localité et le parallélisme d'instructions. Ces deux optimisations sont opposées mais ont pourtant les mêmes objectifs. La fusion consiste à regrouper les opérations effectuées dans deux boucles séparées dans une seule boucle. La fission est l'inverse de la fusion, elle sépare des calculs effectués dans une boucle en deux boucles séparées. La figure 1.6 montre un exemple de fusion ou bien de fission de boucles. Le premier bénéfice apporté par la fusion est la réduction du nombre d'instructions exécutées relatives aux instructions de branchement et de calcul d'adresses. La fusion permet d'améliorer la réutilisation des données et peut favoriser l'application de la promotion des scalaires et d'optimisations de simplifications comme la suppression des tableaux temporaires. Elle permet aussi d'augmenter la diversité des opérations d'une même boucle et d'améliorer ainsi l'utilisation des ressources. Dans le cas où l'architecture cible du programme de la figure 1.6 possède une unité fonctionnelle d'addition et une unité fonctionnelle de multiplication, la fusion améliore le parallélisme. Cependant, la fusion peut également avoir des effets opposées aux améliorations que nous venons de citer. Dans l'exemple, le bénéfice de l'amélioration de la localité sur le tableau `a` peut être inhibé par des conflits de cache entre les tableaux `c` et `d`. La fusion a également un impact sur le réordonnancement des instructions car elle peut augmenter les dépendances dans le corps de la boucle. La fission de boucle est donc utilisée pour réduire les conflits de cache et les dépendances.

```
do i = 1, n
| a(i) = b(i) + c(i)
do i = 1, n
| d(i) = a(i) * d(i)
```

(a) avant fusion ou après fission

```
do i = 1, n
| a(i) = b(i) + c(i)
| d(i) = a(i) * d(i)
```

(b) après fusion ou avant fission

Fig. 1.6 – *Fusion et fission de boucles*

Décalage d'itérations Le décalage d'itérations généralise l'optimisation connue sous le nom de pipeline logiciel en permettant de décaler des groupes complets d'opérations dans l'espace d'itération. La figure 1.7 montre un exemple de décalage d'itérations où des calculs intermédiaires sont anticipés en étant décalés sur les deux itérations précédentes. Le décalage d'itérations augmente le parallélisme d'instructions en augmentant la durée de vie des variables temporaires et donc la pression sur les registres.

Blocage de boucles Le blocage de boucles consiste à découper l'espace d'itérations d'un nid de boucles pour le parcourir différemment. Cette transformation est habituellement utilisée

```
do i = 1, n
| a(i) = a(i)-k*(b(i)+c(i))*(d(i)+e(i))
```

(a) avant

```
t3 = b(1) + c(1) * d(1) + e(1)
t1 = b(2) + c(2)
t2 = d(2) + e(2)
do i = 1, n-2
| a(i) = a(i) - k * t3
| t3 = t1 * t2
| t1 = b(i+2) + c(i+2)
| t2 = d(i+2) + e(i+2)
a(n-1) = a(n-1) - k * t3
a(n) = a(n) - k * (t1 * t2)
```

(b) après

Fig. 1.7 – *Décalage d'itérations*

pour améliorer la localité. La figure 1.8 montre un exemple de blocage de boucles sur un produit de matrices et la figure 1.9 représente les accès aux matrices correspondantes. Cette optimisation peut être réalisée pour des niveaux différents de la hiérarchie mémoire et de manière hiérarchique [NaJuLa94, CaFeHu95]. Dans l'exemple du produit de matrices, les paramètres `Tj` et `Tk` doivent être choisis de manière à ce que le bloc `Tj*Tk` de la matrice `a` et la colonne `Tk` de la matrice `b` tiennent dans le niveau de cache considéré.

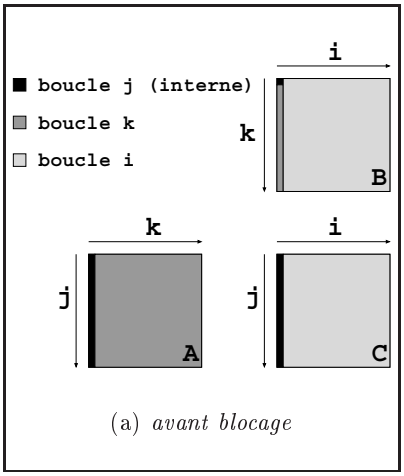
```
do i=1,Ni
| do k=1,Nk
| | R=b(k,i)
| | do j=1,Nj
| | | c(j,i)=c(j,i)+R*a(j,k)
```

(a) avant blocage

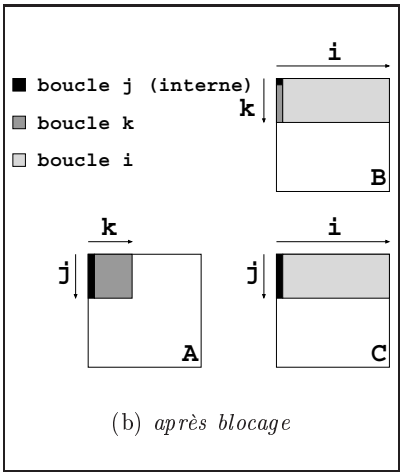
```
do kk=1,Nk,Tk
| do jj=1,Nj,Tj
| | do i=1,Ni
| | | do k=kk, MIN(kk+Tk-1,Nk)
| | | | R=b(k,i)
| | | | do j=jj, MIN(jj+Tj-1,Nj)
| | | | | c(j,i)=c(j,i)+R*a(j,k)
```

(b) après blocage

Fig. 1.8 – *Exemple d'un blocage de boucle sur un produit de matrices*



(a) *avant blocage*



(b) *après blocage*

Fig. 1.9 – *Ordre d'accès aux matrices*

Réarrangement de données Le réarrangement de données consiste à modifier les déclarations de données et éventuellement les accès aux données afin que le placement des données en mémoire permette au programme d'exploiter une meilleure localité. La figure 1.10 montre deux types de réarrangement de données. Le premier modifie la structure du tableau `b` en inversant ses dimensions et modifie les références correspondantes. Ce genre de réarrangement peut être plus complexe lorsque, par exemple, les accès à une matrice se font en diagonale. Le second est l'insertion d'une variable fantôme `d` qui permet d'éviter des conflits de cache lorsque les tailles de matrices sont des multiples d'une puissance de 2 ; cette transformation est appelée *padding*.

```
double precision a(m,n), b(n,m)
...
do j = 1, n
  do i = 1, m
    a(i,j) = a(i,j) * b(j,i)
```

(a) avant

```
double precision a(m,n), d(8),b(m,n)
...
do j = 1, n
  do i = 1, m
    a(i,j) = a(i,j) * b(i,j)
```

(b) après

Fig. 1.10 – *Réarrangement des données*

1.2.2 Optimisation automatique

Les optimisations automatiques sont réalisées par les compilateurs et les préprocesseurs qui doivent suivre l'évolution des architectures et s'adapter à leur complexité croissante.

Pour réaliser une optimisation automatiquement, un compilateur doit réaliser les trois étapes nécessaires : la sélection, la validation et la transformation. Nous nous intéressons plus particulièrement à l'étape de sélection. Elle nécessite une analyse du programme et une modélisation de l'architecture pour évaluer le comportement du programme sur celle-ci. Cette évaluation doit permettre de déterminer si une transformation doit être appliquée, c'est-à-dire si elle améliore la performance.

Modélisation de l'architecture et analyse du programme

Les transformations de simplifications ne nécessitent pas de modèle de l'architecture parce qu'elles sont généralement indépendantes de l'architecture. Les transformations relatives à la génération de code sont, elles aussi, systématiquement appliquées. Cependant, leurs algorithmes nécessitent un modèle précis de l'architecture interne. L'algorithme de réordonnancement des instructions doit posséder une représentation des unités fonctionnelles, connaître à la fois la latence de chaque instruction, le nombre d'instructions chargées depuis le cache d'instructions par cycle, le nombre d'instructions lancées à l'exécution par cycle, les tailles des queues des opérations mémoires et les différents conflits que peuvent rencontrer certaines instructions dans le pipeline, etc.

Les transformations relatives aux structures de contrôles et aux structures de données nécessitent des modèles complexes. Ces transformations sont utilisées principalement pour améliorer la localité, les premiers travaux se sont donc focalisés sur la modélisation de la hiérarchie mémoire. L'analyse du programme qui doit être associée au modèle de l'architecture s'est inspirée de l'analyse des dépendances plus particulièrement des *vecteurs de distances*. Les premiers résultats

conséquents en analyse dédiée à la sélection sont ceux de Wolf et Lam [WoLa91] qui ont formalisé deux concepts : la *réutilisation* et la *localité*. Cette formalisation leur a permis d'automatiser des transformations uni-modulaires avec un algorithme favorisant la réutilisation. Le modèle de hiérarchie mémoire implicitement utilisé est très abstrait car il s'agit d'un espace mémoire sans fonction de placement.

Nous allons illustrer ci-dessous, à l'aide des optimisations pour la mémoire, comment des modèles de plus en plus précis de l'architecture ont été utilisés.

Analyses et modélisations de la hiérarchie mémoire Lam, Rothberg et Wolf [LaRoWo91] ont mis en évidence les problèmes liés au comportement du cache sur les programmes utilisant le blocage de boucles. Les défauts de caches sont de trois types : les défauts de démarrage (*compulsory misses*), les défauts de conflits (*conflict misses*), et les défauts de capacité (*capacity misses*). Ils ont montré la difficulté du choix des paramètres (Tj et Tk) pour réduire les conflits dans le cache et ont étudié le comportement du cache en faisant varier la taille des blocs et la taille des matrices. Ils présentent un algorithme permettant de calculer la taille de bloc rectangulaire maximale n'engendrant pas de conflit de cache sur lui-même (*self interference*). Le cache considéré pour l'algorithme est un cache à correspondance directe. Le modèle utilisé pour guider l'algorithme est très simple puisqu'il utilise uniquement la taille du cache.

Coleman et McKinley [CoKi95] présentent un algorithme basé sur la division euclidienne pour calculer plusieurs tailles de blocs rectangulaires sans conflit de cache entre eux. Ils enrichissent leur algorithme d'une fonction de coût qui propose de sélectionner les tailles de blocs minimisant les interférences entre les différents blocs et entre les différentes variables. Le cache considéré pour l'algorithme est toujours un cache à correspondance directe mais le modèle utilisé pour guider l'algorithme prend en considération la taille du cache et la taille d'une ligne du cache.

Rivera et Tseng [RiTs99] comparent différents algorithmes de blocage de boucles. Ils simplifient l'algorithme de Coleman et McKinley par un algorithme récursif. Ils proposent de combiner le réarrangement des données (*padding*) avec les algorithmes de blocage pour résoudre les problèmes de conflits de cache rencontrés pour certaines tailles de matrices. Le cache considéré pour tous les algorithmes reste un cache à correspondance directe et les modèles utilisent la taille du cache et de la ligne.

Navarro, Juan et Lang [NaJuLa94] généralisent un algorithme de blocage pour une hiérarchie mémoire à m niveaux prenant ainsi en compte les effets du blocage sur le cache de traductions d'adresses (TLB). L'algorithme réalise le blocage successivement sur les différents niveaux de la hiérarchie mémoire. La hiérarchie mémoire considérée est un ensemble de caches à correspondance directe modélisés par leurs tailles et celles de leur ligne.

Mitchell et al. [MiCaFe97] revisitent cet algorithme de blocage pour une hiérarchie mémoire en utilisant une fonction de coût global pour l'ensemble de la hiérarchie.

Ghosh, Martonosi et Malik [GhMaMa99] décrivent une méthode générale pour mettre en équations les relations précises entre les indices des nids de boucles, les tailles de tableaux, leurs adresses de bases, et les paramètres du cache. Ces équations appelées *CME* pour *Cache Miss Equations* permettent de déterminer précisément le comportement du cache et peuvent ainsi guider l'optimisation des nids de boucles. L'architecture considérée est un cache alloué en écriture avec une associativité quelconque et une politique de remplacement de type *LRU*. Les *CME* peuvent être étendues à plusieurs niveaux de caches mais le temps de calcul devient

prohibitif.

Chatterjee et al. [ChPaHaLe01] ont développé un modèle pour caractériser le comportement exact des nids de boucles sur une hiérarchie mémoire. Ils utilisent l'arithmétique de Presburger [We97] pour identifier différents types de défauts de caches et pour déterminer l'état du cache à la fin des boucles. La hiérarchie mémoire considérée possède deux niveaux de cache alloué en écriture avec une associativité quelconque.

Les architectures deviennent de plus en plus complexes, par conséquent, les modèles d'architectures utilisés par les heuristiques des compilateurs sont de moins en moins précis et de plus en plus difficiles à construire et à intégrer dans les compilateurs.

Monsifrot, Bodin et Quiniou [MoBoQu02] ont montré qu'il était possible de générer une heuristique à l'aide d'une méthode d'apprentissage pour le déroulage de boucle. Ils utilisent une méthode d'apprentissage basée sur les arbres de décision. Les paramètres d'apprentissage qui caractérisent les boucles sont extraits statiquement du programme. Pour le déroulage de boucles, la méthode utilise 6 paramètres : le nombre d'instructions, le nombre des opérations arithmétiques, le nombre minimum d'itérations, le nombre d'accès aux tableaux, le nombre d'éléments de tableau réutilisés et le nombre d'instructions conditionnelles. Les boucles ayant des paramètres identiques appartiennent à la même classe. Le processus d'apprentissage applique le déroulage à un ensemble de boucles et identifie les classes bénéficiant de cette optimisation. Ils utilisent un outil de classification générant des arbres de décision obliques. L'intégration des heuristiques dans le compilateur se fait par une implémentation des arbres de décision générés. Le temps moyen des exécutions des programmes dont le déroulage est guidé par l'heuristique d'apprentissage est inférieur de 4% au temps moyen des exécutions des programmes dont le déroulage est guidé par l'heuristique du compilateur *g77*. En appliquant la méthode d'apprentissage avec les mêmes paramètres sur deux architectures différentes, ils démontrent que la méthode s'adapte à l'architecture cible.

Stephenson et al. [StAmMaOR03] proposent un algorithme génétique permettant de construire des fonctions de priorités utilisées par les heuristiques. L'algorithme génétique génère directement la fonction de priorité. Ils montrent le fonctionnement de l'algorithme pour construire les fonctions de priorités des heuristiques pour trois optimisations : la construction d'hyper-blocs, l'allocation des registres et le préchargement des données. Contrairement aux travaux précédents, les paramètres d'apprentissage ne sont pas uniquement statiques, ils utilisent également des informations dynamiques obtenues par profilage. Par exemple, pour la construction d'hyper-blocs, ils utilisent comme paramètre la fréquence d'exécutions des chemins.

Le compromis des méthodes d'apprentissage est le choix des paramètres. En effet, la taille de l'ensemble des programmes d'apprentissage ou de la population (et par voie de conséquence le temps d'apprentissage) croît de manière exponentielle avec le nombre de paramètres. Les méthodes d'apprentissage simplifient la construction des compilateurs, il reste cependant des efforts à fournir pour identifier les paramètres d'apprentissage pertinents et pour configurer le processus d'apprentissage.

Importance de l'ordre de transformations

L'analyse statique et la modélisation des architectures ne sont pas les seules difficultés que rencontrent les compilateurs. La phase d'optimisation d'un compilateur est scindée en sous-

phases dans lesquelles sont effectuées plusieurs optimisations. La plupart du temps l'ordre suivant lequel sont appliquées les transformations est figé. Allen et Kennedy ainsi que Muchnick[AlKe01, Mu97] décrivent les structures générales des compilateurs et les optimisations de chaque phase de manière détaillée. Par exemple, la structure du compilateur *Open Research Compiler* [ORC] est découpée en 5 phases réalisant toutes les optimisations :

- `PreOPT` (*Pre-Optimizations*) : cette phase réalise des pré-optimisations simples telles l'élimination de code mort et réalise aussi les analyses statiques qui vont être utilisées par les phases suivantes.
- `LNO` (*Loop Nest Optimizations*) : cette phase contient toutes les transformations sur les tableaux et les nids de boucles.
- `WOPT` (*WHIRL or global Optimizations*) : cette phase effectue des optimisations générales comme l'élimination d'expressions redondantes, des optimisations sur les pointeurs et sur le graphe de contrôle ainsi que des optimisations de simplifications arithmétiques.
- `CG` (*Code Generation*) : cette phase réalise toutes les optimisations relatives à la génération de code.

Généralement, les séquences d'optimisations réalisées dans chaque phase sont figées. Seules quelques transformations peuvent être répétées et permutées, par exemple dans le compilateur ORC, seules la fusion et la fission de boucles peuvent être permutées et appelées plusieurs fois.

L'importance des compositions de transformations a été montrée par de nombreux travaux. Rivera et Tseng [RiTs98, RiTs98+] ont proposé des algorithmes de réarrangement des données (*Padding*) pour minimiser les interférences dans le cache. Ils ont ensuite étudié le comportement du cache vis-à-vis de différentes transformations de programmes et ils ont commencé à étudier les combinaisons d'optimisations comme la fusion de boucles et le réarrangement des données pour augmenter la réutilisation tout en limitant les conflits de cache. Ils ont étendu ces combinaisons d'optimisations pour des hiérarchies mémoires à plusieurs niveaux de caches [RiTs99].

Whitfield et Soffa [WhSo90] proposent un environnement pour étudier les interactions entre les optimisations. L'environnement permet d'exprimer les pré-conditions et post-conditions des transformations. Si la pré-condition d'une transformation t_2 est incluse dans la post-condition d'une transformation t_1 alors t_1 autorise t_2. L'étude des conditions d'un ensemble de transformations permet de dériver un ordre qui favorise l'application des transformations.

Triantafyllis et al. [TrVaVaAu03] ont récemment proposé une structure de compilateur qui explore simultanément plusieurs séquences d'optimisations dirigées par différentes heuristiques. Ils proposent de sélectionner, *a posteriori*, la meilleure séquence d'optimisations à l'aide d'un prédicteur de performance travaillant sur le code généré.

Zhao et al. [ZhCaWh02] ont présenté un environnement de compilation interactif utilisé pour l'optimisation de programmes pour *l'embarqué* (VISTA). Cet environnement permet à l'utilisateur de contrôler les parties de programmes sur lesquelles ils appliquent des séquences d'optimisations, de modifier manuellement les transformations de programmes, de défaire des transformations de programmes déjà réalisées et de visualiser la représentation intermédiaire du programme en cours d'optimisation.

Kulkarni et al. [KuZhMo03] ont utilisé l'environnement de compilation interactif VISTA et ont intégré un algorithme génétique pour rechercher des séquences d'optimisations. Ils ont ensuite proposé des algorithmes pour accélérer le temps de recherche des algorithmes génétiques [KuHiHi04].

Almagor et al. [AlCoGr04] ont étudié l'espace des séquences d'optimisations afin de caractériser celui-ci et de déterminer le coût de construction de séquences d'optimisations adaptées. Ils proposent trois algorithmes de recherche dans l'espace des séquences d'optimisations.

Optimisation dirigée par l'analyse dynamique

L'analyse dynamique permet d'améliorer le processus de sélection des optimisations en intégrant des informations sur le comportement du programme sur l'architecture. En outre, lors de la compilation d'un programme, de nombreux paramètres sont inconnus car ils dépendent du jeu de données utilisé par le programme. Ainsi, le nombre d'itérations des boucles, la taille des tableaux, le comportement des branchements ne sont connus que lors de l'exécution. Différentes techniques [DeHiWa97, PerfMon3, OProfile, VTune] sont utilisées pour collecter ces informations lors de l'exécution ; elles sont ensuite utilisées par les compilateurs pour guider leurs optimisations.

Cohn et Lowney [CoLo99] décrivent les optimisations dirigées par l'analyse dynamique dans le compilateur Alpha. Ils obtiennent un gain de 17% par rapport aux optimisations guidées statiquement. Les optimisations dirigées par l'analyse dynamique sont généralement des transformations pour améliorer le parallélisme d'instructions. Elles visent généralement à améliorer le comportement des branchements.

La principale difficulté de l'optimisation dirigée par l'analyse dynamique est qu'elle rend les optimisations réalisées sur le programme très dépendantes du jeu de données utilisé à la compilation. Pour adapter les optimisations à différents jeux de données, Bala, Duesterwald et Banerjia [BaDuBa00] ont proposé un système permettant d'optimiser les programmes durant leurs exécutions. Le système collecte des informations sur l'exécution en cours d'un programme et optimise les parties de programmes les plus coûteuses en temps. La principale difficulté est de gérer le surcoût de l'optimisation qui a lieu sur le même processeur que le programme exécuté. Voss et Eigenmann [VoEi00, VoEi01] propose un environnement similaire mais découplé, c'est-à-dire un environnement où les optimisations sont réalisées sur un processeur différent de celui exécutant le programme. Ils proposent un langage permettant de définir des heuristiques d'optimisations dynamiques.

Kistler et Franz [KiFr03] présentent l'étude d'un système permettant de réaliser des optimisations lors du chargement du programme et lors de son exécution. Ils proposent, comme les autres travaux sur l'optimisation dynamique, des optimisations pour améliorer le parallélisme d'instructions mais ils proposent également une optimisation pour améliorer l'utilisation de la hiérarchie mémoire basée sur le réarrangement des données.

Kisuki et al [KiKnOB00, KiKnOB00+] ont introduit la compilation itérative et son utilisation pour rechercher simultanément les paramètres d'une combinaison de transformations (déroulage, blocage de boucles et réarrangement des données). Le principe de la compilation itérative est, pour sélectionner la meilleure combinaison de paramètres, de répéter la compilation et l'exécution du programme avec des paramètres différents. Ils ont exploré l'utilisation de différents algorithmes pour parcourir l'espace de recherche. Ils ont également étudié l'apport d'un modèle de cache pour réduire l'espace de recherche [KKGO00]. Malheureusement, l'espace de recherche croît de manière exponentielle avec le nombre de transformations utilisées. Le temps de recherche devient prohibitif pour un nombre important de transformations.

1.2.3 Optimisation manuelle

Il existe peu de travaux académiques sur l'optimisation manuelle, en revanche, de nombreux outils ont été développés pour que celle-ci soit pratiquée.

Les outils de profilage [DeHiWa97, PerfMon3, OProfile, VTune] utilisent des compteurs matériels. Les compteurs matériels comptent les différents événements se produisant dans une architecture lors de l'exécution d'un programme comme par exemple le nombre de défauts de cache. Les capacités des outils de profilages dépendent des architectures qui mettent à disposition du programmeur plus ou moins d'informations. Lors de l'exécution d'un programme, chaque fois qu'un événement se produit le compteur correspondant est incrémenté. Lorsque ce compteur déborde, une interruption apparaît, le numéro du compteur et l'adresse de l'instruction ayant provoqué le débordement sont transmis au système qui collecte les informations. Plus une instruction provoque un événement, plus la probabilité que cette instruction fasse déborder le compteur est importante. Après l'exécution du programme, l'information collectée par le système est une distribution statistique des événements sur les instructions du programme. En fonction des architectures, l'attribution d'un événement à une instruction est plus ou moins précise. En effet, les processeurs à exécution dans le désordre comme les processeurs *EV6, EV7, Opteron* ou *Pentium* ont parfois des difficultés à identifier l'instruction provoquant un événement. En revanche, les processeurs à exécution dans l'ordre comme les processeurs *EV5* ou *Itanium* attribuent les événements précisément aux instructions qui les provoquent. Les processeurs les plus récents, comme l'*Itanium2* [IPF2], permettent de mesurer jusqu'à 4 compteurs simultanément et plus de 300 événements différents.

L'optimisation manuelle de programmes est une tâche difficile car elle nécessite une connaissance précise de l'architecture des machines et des transformations de programmes. Cette double connaissance doit permettre de trouver les opportunités de transformations et de construire des séquences de transformations. L'objectif des travaux présentés ci-dessous est de faciliter et d'accélérer le processus manuel d'optimisation.

Le projet *MHAOTEU* [AbToAn00] propose un ensemble organisé d'outils d'analyse (statiques et dynamiques) et d'optimisation pour les hiérarchies mémoires. Les outils sont accessibles par l'utilisateur à travers une interface unique. Elle est connectée à un serveur qui communiquent avec les outils d'analyses et de transformations. Le serveur enregistre toutes les informations relatives aux analyses dans une base de données. Parmi les outils d'analyses dynamiques proposés pour la hiérarchie mémoire, les outils de visualisation de mémoire cache (*CVT*) [Wa02] permettent à l'utilisateur de visualiser les phénomènes de conflits. Cet outil instrumente le programme de l'utilisateur pour que l'exécution de celui-ci transmette les opérations mémoires à un simulateur de cache pas à pas. Une interface graphique montre le placement des données dans le cache.

Monsifrot et Bodin [MoBo01] proposent un environnement interactif d'optimisation. L'environnement *CAHT* est basé sur le raisonnement à partir de *cas*. Il permet de rechercher, dans les parties critiques d'un programme, les opportunités d'optimisation (dont la légalité n'est pas garantie) et de les proposer à l'utilisateur. Ce dernier a la responsabilité de vérifier et de valider le programme optimisé. Une base de connaissances mémorise les couples *(cas,transformation)*. Les *cas* sont des conjonctions de propriétés de boucles ou des reconnaissances de structures de calculs. Les propriétés, au nombre de 31, sont extraites par une analyse statique des nids de boucles. Les *cas* ont été construits à partir des guides d'optimisation. L'environnement est évolutif car l'utilisateur peut définir de nouveaux *cas* et ajouter des transformations de programmes.

La construction des *cas* est un travail d'expert qui nécessite une bonne pratique de l'optimisation de programmes. Cette automatisation de la recherche des *cas* est complémentaire de nos travaux, qui sont focalisés sur la sélection rapide des transformations et l'ordre de composition des transformations.

L'environnement *CAHT* présente un point commun majeur avec le processus d'optimisation que nous proposons dans le chapitre 3 : ils mettent à profit l'expertise empirique de l'optimisation manuelle en la formalisant et permettent son évolution, En revanche, les principales différences apparaissent dans l'identification des cas. Effectivement, l'identification est basée sur une analyse statique des programmes pour *CAHT* alors qu'elle est basée sur une analyse dynamique pour le processus présenté. Enfin contrairement à *CAHT*, le processus que nous proposons est itératif.

1.3 Synthèse

Les architectures ont évolué de manière incrémentale en intégrant des mécanismes de plus en plus complexes. L'association de ces mécanismes et l'évolution de la technologie ont considérablement augmenté la performance *crête* des processeurs. Les optimisations des compilateurs sont cruciales pour permettre aux programmes d'exploiter une partie de cette performance (la performance *soutenue*). L'augmentation de la complexité des architectures rend très difficile le maintien d'une performance *soutenue* peu éloignée de la performance *crête* des processeurs.

Les techniques classiques d'optimisation automatique utilisent des analyses statiques et des modèles simplifiés des architectures pour prédire le comportement des programmes.

Ces techniques rencontrent deux problèmes majeurs :
- d'une part certains paramètres des programmes ne sont connus qu'au moment de l'exécution ce qui restreint le pouvoir de l'analyse statique,
- d'autre part, les architectures devenant de plus en plus complexes, les modèles d'architectures ont de plus en plus de difficultés à représenter fidèlement le comportement des composants.

Les optimisations guidées par l'analyse dynamique pallient le manque d'information des analyses statiques. Les informations extraites de l'analyse dynamique précisent généralement le comportement du programme vis-à-vis d'un seul composant de l'architecture, par exemple le cache d'instructions. Malheureusement, la majeure partie de la prédiction du comportement des programmes reste à la charge des modèles d'architectures.

Les méthodes d'apprentissage et les algorithmes génétiques ont montré des résultats intéressants pour automatiser la création des heuristiques. Malheureusement, ces méthodes sont coûteuses en temps d'apprentissage et ne sont envisageables qu'individuellement pour chaque optimisation. La complexité de ces méthodes ne permet pas de les utiliser pour créer des heuristiques prenant en compte des ensembles d'optimisations.

Les techniques d'optimisation automatiques doivent affronter une difficulté supplémentaire qu'elles utilisent ou non des analyses dynamiques. Cette difficulté est le choix de l'ordre d'application des transformations. En effet, le plus souvent cet ordre est figé. Peu de travaux ont étudié les interactions entre les optimisations, notamment l'impact des compositions des transformations sur la performance. Comme pour les heuristiques, les méthodes d'apprentissage sont des moyens *obscurs* de construire de nouvelles séquences d'optimisations. Mêmes si ces méthodes obtiennent des résultats, elles manquent d'informations détaillées sur le comportement des programmes pour

que l'apprentissage soit pertinent.

Les travaux sur l'optimisation manuelle se sont focalisés sur des thèmes différents mais complémentaires qui sont l'analyse dynamique et la formalisation de l'expertise empirique.

L'efficacité d'un processus d'optimisation peut être accrue par une amélioration de la compréhension du comportement des programmes sur les architectures complexes. La tendance suivie par l'optimisation automatique est l'apport de plus d'informations issues de l'analyse dynamique des programmes mais soit cet apport est encore très insuffisant, soit le temps de convergence des heuristiques de recherche est long.

Dans cette thèse, nous allons d'abord mettre en évidence l'importance de disposer des informations détaillées sur le comportement du programme sur l'architecture. Puis, nous verrons comment nous pouvons mettre à profit une expertise manuelle d'optimisation de programmes pour construire une heuristique rapide d'optimisation itérative de programmes.

Chapitre 2

Optimisation et Analyse dynamique détaillée

2.1 Introduction

La complexité des processeurs ne cessent d'augmenter et il devient de plus en plus difficile pour les compilateurs de générer des programmes "performants". En effet, les compilateurs utilisent des modèles d'architectures simplifiés leurs permettant de sélectionner et de paramétrer des optimisations dans des contraintes de temps relativement raisonnable pour les utilisateurs. Il existe de nombreux travaux sur les transformations de programmes [BaGrSh94, LaRoWo91, ChPaHaLe01, Ca96, McTe96, CoKi95, KiKnOB00, NaJuLa94] qui se focalisent le plus souvent sur un ou deux composants de l'architecture, en revanche, seul un petit nombre d'entre eux étudient l'optimisation pour plusieurs composants de l'architecture simultanément et leurs éventuelles interactions.

Le but de ce chapitre est de montrer qu'il est possible et nécessaire d'utiliser une analyse dynamique très détaillée pour guider les optimisations et ainsi d'obtenir de "bonnes performances". Pour cela, nous avons choisi d'optimiser un noyau de calcul très étudié pour lequel de nombreux travaux d'optimisation ont été réalisés : le produit de matrices. Nous avons extrait un sous-ensemble de transformations de programmes de la littérature et de l'industrie et nous avons appliqué ces transformations en fonction d'une analyse dynamique très précise. L'analyse dynamique a été obtenue à l'aide du simulateur cycle à cycle du processeur Alpha-21264 (EV68).

La section 2.2 présente l'environnement de travail dans lequel nous avons réalisé cette étude. Dans la section 2.3 seront détaillées l'analyse et l'optimisation du noyau de calcul. La section 2.4 apporte les conclusions et discussions.

2.2 Environnement de travail

Le processeur cible est un Alpha-21264 (EV68) cadencé à 1GHz et à exécution dans le désordre avec une mémoire cache de second niveau de 8 Moctets. Son architecture est décrite par la figure 1.1 dans le chapitre 1. Le système d'exploitation est Tru64-Unix 5.A qui utilise des pages mémoires de 8 Koctects.

Pour chaque expérience réalisée sur le produit de matrices, nous avons mesuré les temps d'exécutions pour un intervalle de tailles de matrices centré autour de 1060 éléments. Tous les gains en temps d'exécution indiqués dans cette section sont donc calculés à l'aide de cette moyenne.

2.3 Optimisation du produit de matrices basée sur une analyse dynamique détaillée

Notre travail d'optimisation a été guidé par la performance mais surtout par le comportement du programme sur chaque composant de l'architecture. L'analyse dynamique détaillée fournie par le simulateur cycle à cycle du processeur Alpha-21264 nous a permis de comprendre les phénomènes qui se produisent sur chaque composant de l'architecture. Pour chacun de ces phénomènes, nous avons testé plusieurs optimisations et sélectionné celles qui apportaient le plus de performance. Le résultat de ces expériences est une suite de versions du produit de matrices optimisées. La table 2.1 montre, pour chaque étape d'optimisation, le gain obtenu par rapport au code initial. Elle montre également le gain obtenu avec le niveau d'optimisation maximum du compilateur Fortran (V5.4) (`-O5`) et le gain obtenu avec l'association du préprocesseur KAP [Kap] (`-O5 + KAP`).

Etape	**Transformation**	**gain**
Etape 0	Programme initial	1.00
Etape 1	Blocage sur 2 dimensions pour cache de niveau 1	2.48
Etape 2	Blocage sur 3 dimensions pour TLB	2.62
Etape 3	Blocage sur 3 dimensions + permutation de boucle pour queue d'écriture	3.11
Etape 4	Blocage sur 3 dimensions + permutation de boucle + déroulage pour ILP	3.71
Etape 5	Blocage sur 3 dimensions + permutation de boucle + déroulage + blocage pour les registres	9.90
Etape 6	Blocage sur 2 dimensions pour cache de niveau 1 + copie pour TLB + permutation de boucle + déroulage de boucle + blocage pour les registres	8.43
Etape 7	Blocage sur 2 dimensions + copie + permutation de boucle + déroulage de boucle + blocage pour registres + préchargement pour latence	12.25
Etape 8	Blocage sur 2 dimensions + copie + permutation de boucle + déroulage de boucle + blocage pour registres + préchargement pour latence + Blocage sur 3 dimensions pour cache de niveau 2	12.75
Etape 9	Blocage sur 2 dimensions + copie + permutation de boucle + déroulage de boucle + blocage pour registres + préchargements pour latence + Blocage sur 3 dimensions pour cache de niveau 2 + optimisation spécifique pour l'architecture	**13.56**
-O5		3.26
-O5 + KAP		3.37

Tab. 2.1 – *Gains moyens pour des matrices de l'ordre de* 10^6 *éléments*

Le programme initial (Etape 0) est présenté dans la figure 2.1. Chaque version a été compilée

avec les options suivantes : `-O2 -unroll 1 -nopipeline` afin d'appliquer les optimisations utiles à la génération de code et d'inhiber les optimisations spécifiques à l'architecture et les transformations sur les boucles. Ces options de compilation évitent également que les optimisations manuelles ne soient détruites par les optimisations du compilateur.

```
do i=1,Ni
  do k=1,Nk
    R=b(k,i)
    do j=1,Nj
      c(j,i)=c(j,i)+R*a(j,k)
```

Fig. 2.1 – *Etape 0 : Programme initial*

Dans les paragraphes suivants, nous étudions chaque composant de l'architecture. Au début de chaque paragraphe, nous nommons le composant cible et nous donnons le gain obtenu par l'optimisation ainsi que sa contribution au gain final en pourcentage.

Cache de niveau 1 (Etape 1, gain=2,48, contribution=12%). De nombreux travaux de recherches sur l'optimisation de programmes de calculs scientifiques, et plus particulièrement sur le produit de matrices, se sont focalisés sur les mémoires caches. En effet, ces programmes utilisent généralement de grands espaces de données, réalisant par conséquent une utilisation intensive de la mémoire. Dans un premier temps, nous avons observé que le taux d'échecs du premier niveau de cache (Dcache pour l'Alpha) était de 32%.

Le blocage de boucle [WoLa91, LaRoWo91, CoKi95] est la transformation de programmes la plus connue pour améliorer l'utilisation des mémoires caches. Rothberg et al. [LaRoWo91] ont montré que la taille du bloc doit être choisie de telle manière que les conflits de cache soient minimisés. Les performances des techniques de sélection des tailles de blocs basées sur l'organisation du cache et des données [CoKi95] sont moins bonnes que les performances des techniques de sélection basées sur le temps d'exécution global [KiKnOB00]. Cette dernière technique basée sur le temps d'exécution prend en considération de manière implicite l'ensemble de l'architecture. Nous avons bloqué sur deux dimensions la matrice A. Nous avons fait une recherche exhaustive afin de trouver une taille de bloc qui minimise le temps d'exécution et nous avons trouvé $Tk = Tj = 33$. Même si l'optimisation a considérablement réduit le taux de défaut de premier niveau de cache (réduction de 95%), le résultat se traduit uniquement par une contribution au gain final de 12%.

```
do kk=1,Nk,Tk
  do jj=1,Nj,Tj
    do i=1,Ni
      do k=kk, MIN(kk+Tk-1,Nk)
        R=b(k,i)
        do j=jj, MIN(jj+Tj-1,Nj)
          c(j,i)=c(j,i)+R*a(j,k)
```

Fig. 2.2 – *Etape 1 : Blocage sur 2 dimensions*

Cache de niveau 1 + TLB (Etape 2, gain=2,62, contribution=1%). Plusieurs travaux [MiCaFe97] ont montré que le blocage de boucle pour le cache peut avoir un effet néfaste sur le

cache de traduction d'adresse (TLB). La figure 2.3a représente l'ordre d'accès aux matrices pour le produit de matrices de l'étape 0 avant le blocage et la figure 2.3b représente l'ordre d'accès après le blocage. Rappelons-nous qu'en fortran les éléments des tableaux sont rangés en mémoire suivant la première dimension (la plus à gauche), ici les éléments des matrices sont rangés suivant les colonnes. Supposons que la taille d'une colonne des matrices soit égale à la taille d'une page mémoire (8Koctets sur le processeur Alpha). Dans la version non-bloquée, pour chaque itération de i, une entrée du cache de translation d'adresse est utilisée pour chacune des colonnes des matrices B et C. Ensuite, l'entrée de la colonne de B est réutilisée au cours de l'itération k et l'entrée de la colonne de C est réutilisée au cours de l'itération j. Dans la version bloquée, les itérations k et j sont beaucoup plus courtes, les entrées de TLB ne sont donc que partiellement utilisées lors de ces itérations. La taille de l'itération i étant grande, la probabilité que les entrées de TLB contenant les colonnes de B et C soient purgées avant les prochaines itérations de kk et jj est importante. Nous avons observé une augmentation du taux d'échecs de TLB de 4,98e-4 à 5,08e-4. Ce taux est relativement faible mais l'impact des défauts de TLB sur la performance est important.

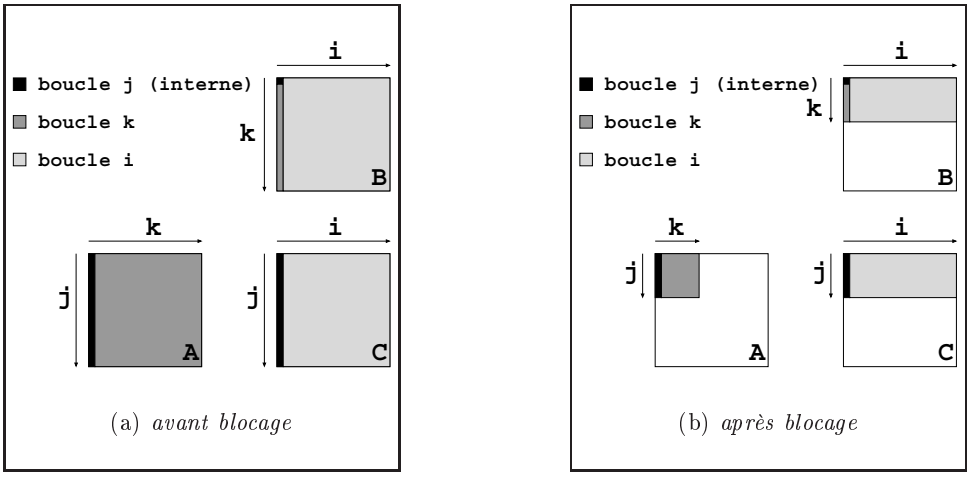


Fig. 2.3 – *Ordre d'accès aux matrices*

Afin de ne pas perdre le bénéfice du blocage pour le cache, nous avons bloqué la troisième dimension i en faisant en sorte que le nombre d'entrées de TLB utilisées par les blocs de B et de C soit inférieur au nombre total d'entrées de TLB. Le programme correspondant est présenté dans la figure 2.4. Le blocage sur les trois dimensions a réduit le taux d'échec de TLB de 5,08e-4 à 2,08e-5. Cette optimisation a augmenté le nombre de défauts de cache de premier niveau mais cela n'est pas suffisant pour compenser le bénéfice obtenu par la réduction des défauts de TLB. Le gain a donc augmenté, il est de : 2,62.

Cache de niveau 1 + TLB + queue d'écriture (Etape 3, gain=3,11, contribution=4%). Généralement, les opérations mémoires d'écriture (de rangement) ne sont pas considérées comme étant prioritaires pour les optimisations. En effet les opérations d'écriture sont envoyées à la mémoire et aucun calcul ne dépend de leurs exécutions (contrairement aux opérations de lecture ou de chargement). Cependant, plusieurs composants de l'architecture peuvent dégrader les performances du processeur à cause des opérations d'écriture, en particulier la queue des opérations d'écriture. Nous avons observé ici un phénomène différent. Dans les processeurs superscalaires, on rencontre des cas où des opérations mémoires doivent être annulées et leurs exécutions doivent

```
do ii=1,Ni,Ti
  do kk=1,Nk,Tk
    do jj=1,Nj,Tj
      do i=ii, MIN(ii+Ti-1,Ni)
        do k=kk, MIN(kk+Tk-1,Nk)
          R=b(k,i)
          do j=jj, MIN(jj+Tj-1,Nj)
            c(j,i)=c(j,i)+R*a(j,k)
```

Fig. 2.4 – *Etape 2 : Blocage sur 3 dimensions*

être retardées. Dans le processeur Alpha, il y a plusieurs cas où deux opérations mémoires ne peuvent pas être exécutées simultanément. Un exemple serait lorsqu'une opération d'écriture et une opération de lecture doivent accéder au même ensemble dans le cache. Comme le montre l'exemple de la figure 2.5, si l'opération de lecture Sj doit accéder à un ensemble identique à celui de l'opération d'écriture Si alors elle est purgée du pipeline avec toutes les opérations Sk plus jeunes que Sj ($k > j$). Ce type d'événements est appelé purge de conflit mémoire.

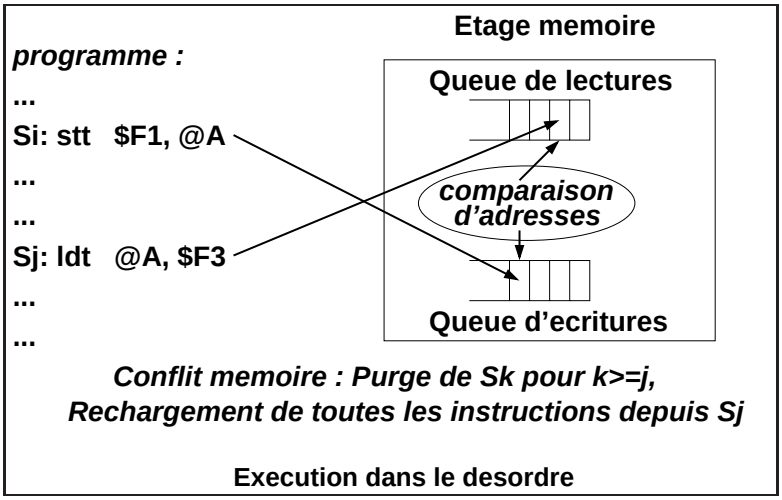


Fig. 2.5 – *Conflit mémoire*

Pour le programme optimisé de l'étape 2, nous avons observé un nombre important de purges dues à des conflits mémoires qui étaient souvent produits par l'opération d'écriture et les deux opérations de lecture de la boucle interne (j). Afin de réduire la probabilité d'avoir des conflits et donc des purges du pipeline, nous avons réduit le nombre d'opérations d'écriture. Nous avons, pour cela, permuté les boucles k et j ainsi montrées dans la figure 2.6. L'ordre d'accès aux matrices est représenté dans la figure 2.7.

La permutation des boucles permet de mémoriser $c(j, i)$ dans un registre et de diviser ainsi le nombre d'opérations d'écriture par la taille du bloc Tk (33). Le nombre de purges dues à des conflits mémoires a ainsi été divisé par 24 et le gain est passé de 2,62 à 3,11.

Cache de niveau 1 + TLB + queue d'écritures + ILP (Etape 4, gain=3,71, contribution=5%). Les registres sont souvent une ressource critique et les corps de boucles importants produisent généralement des instructions de débordements (lectures/écritures), appelées aussi

```
do ii=1,Ni,Ti
  do kk=1,Nk,Tk
    do jj=1,Nj,Tj
      do i=ii, MIN(ii+Ti-1,Ni)
        do j=jj, MIN(jj+Tj-1,Nj)
          R=c(j,i)
          do k=kk, MIN(kk+Tk-1,Nk)
            R=R+b(k,i)*a(j,k)
          c(j,i)=R
```

Fig. 2.6 – *Etape 3 : Permutation de boucle pour la queue des écritures*

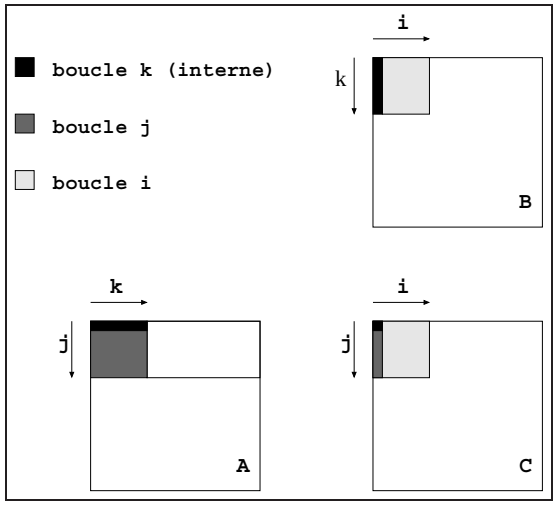


Fig. 2.7 – *Ordre d'accès aux matrices après blocage sur 3 dimensions*

spill code. Dans le cas du produit de matrices, le corps de la boucle est relativement petit et plusieurs registres restent donc disponibles. Les compilateurs utilisent généralement ces registres disponibles pour augmenter le parallélisme d'instructions (ILP) en utilisant le déroulage de boucles ou bien le pipeline logiciel. Afin d'augmenter le parallélisme d'instructions, nous avons déroulé 8 fois la boucle interne et obtenu un gain de 3,71.

Cache de niveau 1 + TLB + queue d'écritures + ILP + Registres (Etape 5, gain=9,90, contribution=49%). Durant les étapes précédentes, nous avons réduit les nombres de défauts du cache de données et du cache de traduction d'adresses (TLB). Nous avons ainsi réduit la latence mémoire moyenne. Cependant, les performances du programme optimisé sont toujours limitées par la mémoire. Effectivement, le rapport du nombre de calculs sur le nombre d'opérations mémoires est faible, il est approximativement égal à 1. Or, les registres (premier niveau de la hiérarchie mémoire) peuvent être utilisés pour réduire le nombre d'accès mémoires. Après l'optimisation précédente (le déroulage de boucle pour augmenter le parallélisme d'instructions), nous avons observé qu'il restait des registres disponibles que nous pouvons utiliser pour réduire le nombre d'accès mémoires. Le blocage pour les registres peut se faire par un simple déroulage de boucle externe [CaGu97]. La figure 2.8 montre le même programme que celui de la figure 2.6 pour lequel la boucle externe j a été déroulée deux fois.

La référence à $b(k, i)$ peut être placée dans un registre, supprimant ainsi un accès mémoire. De

```
do ii=1,Ni,Ti
  do kk=1,Nk,Tk
    do jj=1,Nj,Tj
      do i=ii, MIN(ii+Ti-1,Ni)
        do j=jj, MIN(jj+Tj-1,Nj), 2
          Rc0=c(j,i)
          Rc1=c(j+1,i)
          do k=kk, MIN(kk+Tk-1,Nk)
            Rb0=b(k,i)
            Rc0=Rc0+Rb0*a(j,k)
            Rc1=Rc1+Rb0*a(j+1,k)
          c(j,i)=R
```

Fig. 2.8 – *Etape 4 : Blocage de registres*

la même manière, nous pouvons dérouler la boucle i. Nous avons recherché, de manière exhaustive, la meilleure façon de dérouler les boucles et nous avons trouvé un déroulage de 3-4-8, c'est-à-dire 3 fois pour la boucle i, 4 fois pour la boucle j et 8 fois pour la boucle interne k. Rappelons que le déroulage de la boucle interne ne favorise pas la réutilisation des registres mais qu'il favorise plutôt le parallélisme d'instructions. Nous avons observé une diminution du nombre des lectures mémoires de 69% et le rapport du nombre de calculs sur le nombre d'opérations mémoires a augmenté de 300%. Par conséquent, l'utilisation des unités fonctionnelles est beaucoup plus importante. Le gain passe de 3,71 à 9,90 soit une contribution au gain final de 49%.

En considérant les registres comme un niveau de la hiérarchie mémoire supplémentaire, il est possible d'améliorer la bande-passante mémoire du programme. Il n'est pas facile de bien exploiter les registres physiques d'un processeur superscalaire à exécution dans le désordre car ses registres sont cachés par le mécanisme de renommage dynamique [SmSo95]. En revanche, certaines architectures comme l'architecture EPIC de l'itanium [Epic] exposent de nombreux registres logiques au programmeur et au compilateur.

Interaction entre cache de niveau 1, TLB, queue d'écritures et Registres (Etape 6). L'importance du gain obtenu lors de la précédente étape suggère de prêter plus d'attention et de privilégier le blocage de registres. Dans ce paragraphe, nous révisons les optimisations jusqu'ici appliquées afin de favoriser le blocage de registres. Nous allons voir comment l'interaction entre plusieurs composants de l'architecture peut influencer les optimisations individuellement et la séquence des optimisations.

Dans le paragraphe précédent, nous avions amélioré la réutilisation des registres pour les références $a(j,k)$ et $b(k,i)$ mais nous n'avions pas mentionné la référence $c(j,i)$. Le déroulage de boucle n'améliore pas la réutilisation de cette référence, c'est la taille de la boucle k (Tk) qui détermine la réutilisation de $c(j,i)$. Plus la longueur de l'itération k est grande, plus le nombre des lectures/écritures des références $c(j,i)$ diminue. Or, nous utilisons une taille d'itération k relativement petite ($Tk = 33$), nous pouvons donc augmenter la réutilisation de registre sur la référence $c(j,i)$ en augmentant Tk. Lorsque nous augmentons Tk, nous augmentons la taille des blocs des matrices A et B. Nous avons fait plusieurs tests qui nous ont montré qu'il n'était pas avantageux de perdre la réutilisation du cache de premier niveau au profit de la réutilisation de registres. Nous devons donc faire en sorte que les blocs ($Tk*Ti$) de la matrice B et ($Tj*Tk$) de la matrice A tiennent dans le cache. D'autre part, nous devons conserver la réutilisation de registres

pour les matrices A et B obtenu par le déroulage des boucles externes (3-4-8), c'est-à-dire que nous devons avoir $Ti \geq 3$ et $Tj \geq 4$. Nous avons donc choisi de maximiser Tk en utilisant des tailles telles que les blocs de la matrice B ($Tk * 3$) et de la matrice A ($4 * Tk$) soient inclus dans le cache de premier niveau. Ces blocs étant très étroits, nous avons obtenu une valeur pour Tk supérieure à Nk. Le programme correspondant présenté dans la figure 2.9 conserve le blocage de la boucle k avec le paramètre Tk. On remarque que le blocage des boucles i et j a disparu puisqu'il a été remplacé par le blocage de la boucle k associé au déroulage des boucles externes i et j.

```
do kk=1,Nk,Tk
  do i=1,Ni,3
    do j=1,Nj,4
      Rc0=c(j,i)
      Rc1=c(j+1,i)
      ...
      do k=kk, MIN(kk+Tk-1,Nk)
        Rb0=b(k,i)
        Rb1=b(k,i+1)
        Rb2=b(k,i+2)
        Ra0=a(j,k)
        Ra1=a(j+1,k)
        Ra2=a(j+2,k)
        Ra3=a(j+3,k)
        Rc0=Rc0+Rb0*Ra0
        Rc1=Rc1+Rb0*Ra1
        Rc2=Rc2+Rb0*Ra2
        ...
      c(j,i)=Rc0
      c(j+1,i)=Rc1
      ...
```

Fig. 2.9 – *Compromis entre réutilisation registres/cache*

En observant la performance de ce programme, nous pouvons constater une chute de performance de 48% due à une augmentation du nombre de défauts du cache de traduction d'adresses (TLB) qui ont été multipliés par 780. La valeur de Tk est telle que les références $a(j,k)$ purgent le TLB lors des itérations de la boucle interne k. Pour éviter de purger le TLB sans réduire Tk, nous avons choisi de copier les blocs des matrices A et B dans des espaces mémoires contigus. Nous avons donc un tableau de $4*Tk$ par $Nj/4$ éléments pour copier les blocs de la matrice A et un tableau de $3*Tk$ par $Ni/3$ éléments pour copier les blocs de la matrice B. La copie des blocs de A se fait en parcourant A suivant les colonnes (une colonne du bloc de A est copiée dans les 4 premières lignes du tableau de copie). Pour les blocs de B, les 4 premières colonnes du bloc sont copiées dans la première colonne du tableau de copie. Le schéma de la copie est représenté avec le schéma d'accès aux matrices dans la figure 2.10, le programme correspondant est représenté dans la figure 2.11.

La copie nous a permis de réduire le nombre de défauts du cache de traduction d'adresses de 98%. La copie permet également de réduire les défauts de cache dus à des conflits [TeGrJa93].

Enfin, nous avons effectué plusieurs tests de déroulage (nous avons simultanément fait varier le déroulage externe sur les boucles i, j et le déroulage interne sur la boucle k). Dans la section précédente, le meilleur facteur de déroulage était de 3-4-8. Suite à une nouvelle recherche ex-

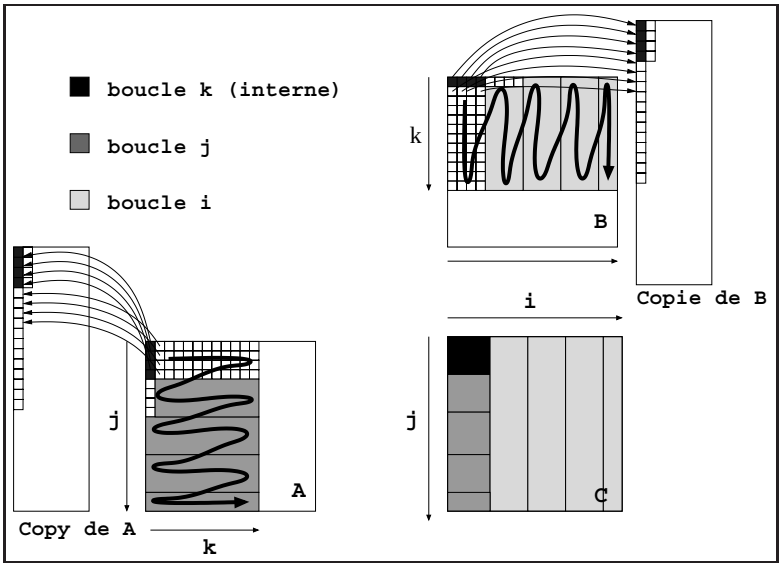


Fig. 2.10 – *Schéma d'accès aux matrices avec copies*

```
do kk=1,Nk,Tk
 | do i=1,Ni,4
C| | Copie du bloc de la matrice B
 | | ...
 | | do j=1,Nj,4
C| | | Copie du bloc de la matrice A
 | | ...
 | | | Rc0=c(j,i)
 | | | Rc1=c(j+1,i)
 | | | ...
 | | | do k=kk, MIN(kk+Tk-1,Nk)
C| | | | Code de la boucle interne travaillant sur les copies
 | | | | ...
 | | | c(j,i)=Rc0
 | | | c(j+1,i)=Rc1
 | | | ...
```

Fig. 2.11 – *Copie des blocs des matrices A et B pour réduire les défauts de TLB*

haustive, nous avons observé que le meilleur facteur de déroulage n'était plus 3-4-8 mais 4-4-2 car la valeur de Tk favorise la réutilisation des registres.

Finalement, le gain obtenu par cette suite d'optimisations est seulement de 8,43 (étape 6) contre 9,90 pour l'étape 5. Cette perte est due essentiellement à une dégradation de l'utilisation du cache pour les matrices A et C et au coût de la copie. Toutefois, le paragraphe suivant va nous montrer une optimisation pour laquelle le programme de l'étape 6 est plus adapté que le programme de l'étape 5. D'autre part, cette même optimisation permet également de réduire le coût de la copie (au final, nous obtiendrons un meilleur gain avec l'étape 6). Une approche similaire est utilisée pour le produit de matrices dans la librairie mathématiques optimisée manuellement pour le processeur Alpha [Cxml].

Cache de niveau 1 + TLB + queue d'écritures + ILP + Registres + Préchargement (Etape 7, gain=12,25, contribution=19%). Le préchargement est une optimisation qui consiste à précharger une donnée dans le cache afin que l'opération de lecture associée accède au cache avec la latence la plus courte. Dans certaines architectures, il existe des mécanismes de préchargement dynamiques. Pour le processeur Alpha ce mécanisme n'existe pas, en revanche des instructions de lecture particulières ont été prévues pour effectuer un préchargement logiciel.

Nous avons appliqué le préchargement logiciel sur le programme initial (étape 0) du produit de matrices (nous avons préchargé toutes les matrices). Nous avons ainsi obtenu un gain de 1,33 car la mauvaise utilisation des caches et des registres génère un trafic mémoire important ne laissant pas suffisamment de bande-passante pour le préchargement. Nous avons également appliqué l'optimisation de préchargement logiciel sur le programme de l'étape 5 et nous avons obtenu un gain de 1,05 par rapport au temps d'exécution du programme de l'étape 5. Ce faible gain est dû à la taille des blocs Tk. En effet, l'itération de la boucle interne k est petite ($Tk = 33$), elle lit 4 lignes de cache (chaque ligne contient 8 éléments de 8 octets) ce qui laisse peu d'opportunités de préchargement (moins de 4 car la distance de préchargement doit être supérieur à une itération pour être efficace).

Dans le programme de l'étape 6, nous utilisons une taille d'itération bien plus grande ($Tk = Nk$) qui permet au préchargement d'être beaucoup plus efficace. Après avoir réduit la bande-passante mémoire utilisée avec le blocage pour les registres et le cache de traduction d'adresses (TLB), la réduction de la latence mémoire avec le préchargement permet de cacher le coût de la copie dans cette version du produit de matrices. Nous avons appliqué le préchargement à plusieurs endroits dans le programme : à la fois dans les copies des blocs des matrices A et B pour masquer la latence entre le second niveau de cache et la mémoire et aussi dans la boucle interne k pour le chargement de la copie de A afin de masquer la latence entre le premier et le second niveau de cache. Le programme de la figure 2.12 représente l'insertion du préchargement en considérant que celui-ci puisse être effectué au niveau du programme source. Lorsque nous avons réalisé ces optimisations, les directives de compilation du compilateur Alpha pour insérer des instructions de préchargement n'étaient pas encore supportées. Nous avons donc inséré les instructions de préchargement au niveau assembleur. Les instructions de préchargement assembleur sont des instructions de lecture dont le registre de destination est le registre nul du processeur Alpha (f31) :

```
ldl f31, 16*8(r5) / préchargement de a(j,k)
```

La distance de préchargement est le nombre d'itérations entre l'itération pendant laquelle la donnée est préchargée et l'itération durant laquelle la donnée est effectivement lue. Cette distance doit être choisie suffisamment grande pour cacher la latence mémoire souhaitée mais cette distance ne doit pas être trop grande pour éviter que les données préchargées ne soient éjectées du cache avant d'être utilisées. Après plusieurs tests, nous avons trouvé une distance de préchargement de 2 itérations.

Finalement, le préchargement appliqué au programme de l'étape 6 permet d'obtenir un gain de 12,25 alors que le préchargement appliqué au programme de l'étape 5 n'obtient qu'un gain de 10,37.

Cache de niveau 1 + TLB + queue d'écritures + ILP + Registres + Préchargement + Cache de niveau 2 (Etape 8, gain=12,75, contribution=4%). Les algorithmes de blocage

```
do kk=1,Nk,Tk
 do i=1,Ni,4
C  Copie du bloc de la matrice B
   Préchargement(b(k+4,i))
   ...
   do j=1,Nj,4
C    Copie du bloc de la matrice A
     Préchargement(a(j+4,k))
     ...
     Rc0=c(j,i)
     Rc1=c(j+1,i)
     ...
     do k=kk, MIN(kk+Tk-1,Nk)
C      Code de la boucle interne travaillant sur les copies
       Préchargement(a(j,k+4))
       ...
     c(j,i)=Rc0
     c(j+1,i)=Rc1
     ...
```

Fig. 2.12 – *Etape 7 : préchargements*

de boucles se focalisent généralement sur le cache de premier niveau. Cependant, les pénalités engendrées par des défauts de caches des niveaux inférieurs sont souvent beaucoup plus importantes que celles du premier niveau. Par exemple, pour le système que nous utilisons, un accès au cache de second niveau (Bcache) coûte 12 cycles processeur alors qu'un accès mémoire coûte 140 cycles processeur. D'autre part, de récents travaux [SrLe98] ont montré que la latence de plusieurs défauts de cache de premier niveau peut être masquée par le réordonnancement dynamique des instructions dans les processeurs à exécution dans le désordre. Cela se traduit par de faibles gains de performances lors de l'optimisation pour le cache de premier niveau.

L'optimisation de blocage de boucles pour une hiérarchie de mémoires caches complète est particulièrement difficile : il est nécessaire de trouver le bon compromis entre les différents niveaux de cache [NaJuLa94, CaFeHu95]. En effet le blocage pour un niveau particulier engendre souvent des dégradations sur les autres niveaux de la hiérarchie mémoire.

Dans la version du programme optimisé de l'étape 6, nous utilisons une taille de bloc de $4*Tk$ pour les matrices A et B car le facteur de déroulage est 4-4-2. La première itération $i = 1$ utilise le premier bloc de la matrice B durant laquelle l'itération j parcourt les blocs de A. A la fin de l'itération i, le bloc de B ne sera plus utilisé. Lors de l'itération i suivante, le second bloc de la matrice B est utilisé et les blocs de la matrice A réutilisés (voir figure 2.10). Mais la distance de réutilisation des blocs de A est trop grande pour que ceux-ci soit encore dans le cache de second niveau. En effet, $Tk = Nk$ et $Ni = Nj = Nk = 1060$ et la taille du cache de second niveau de l'Alpha est de 8 Moctets. Nous avons bloqué les boucles i et j afin de réduire la distance de réutilisation des matrices A et B. Le choix des tailles de blocs est fait de telle sorte que le bloc de la matrice A ($Tj * Tk$) et le bloc de la matrice B ($Tk * Ti$) tiennent dans le second niveau de cache. Le programme correspondant est représenté sur la figure 2.13. Il est très similaire au programme réalisant le blocage sur 3 dimensions utilisé pour le cache de premier niveau et le cache de traduction d'adresses (TLB), cependant, les tailles de blocs (Ti et Tj) ciblent, ici, le second niveau de cache. Généralement, le choix des tailles des blocs est très important lorsque l'associativé du cache est faible, il permet de minimiser les conflits dans le cache [CoKi95, RiTs99].

Même si le cache de second niveau du processeur Alpha est à correspondance directe, les conflits sont déjà minimisés car le cache contiendra les copies des blocs et non pas les blocs eux-mêmes (les copies sont déclarées dans des espaces contigus en mémoire).

```
do ii=1,Ni, Ti
  do kk=1,Nk,Tk
C   Copie du bloc de la matrice B (Tk * Ti)
    Préchargement(b(k+4,i))
    ...
    do jj=1,Nj, Tj
C     Copie du bloc de la matrice A (Tj * Tk)
      Préchargement(a(j+4,k))
      ...
      do i=ii,MIN(ii+Ti-1,Ni), 4
        do j=jj,MIN(jj+Tj-1,Nj), 4
          Rc0=c(j,i)
          Rc1=c(j+1,i)
          ...
          do k=kk, MIN(kk+Tk-1,Nk), 2
C           Code de la boucle interne travaillant sur les copies
            Préchargement(a(j,k+4))
            ...
          c(j,i)=Rc0
          c(j+1,i)=Rc1
```

Fig. 2.13 – *Etape 8 : blocage des boucles i et j pour le cache de second niveau*

Le blocage pour le second niveau de cache a permis de réduire les défauts de cache de second niveau de 11%. Dans le même temps, les défauts de cache du premier niveau ont augmenté de 9% à 10%, mais le gain reste positif et il est de 12,75.

Cache de niveau 1 + TLB + queue d'écritures + ILP + Registres + Préchargement + Cache de niveau 2 + Architecture (Etape 9, gain=13,56, contribution=7%). Les composants de l'architecture pour lesquels nous avons optimisé jusqu'ici sont des composants "classiques", communs à de nombreuses architectures. Les architectures de processeurs possèdent de nombreuses caractéristiques spécifiques qui reflètent les compromis faits par les architectes lors de la conception. Parfois, ces caractéristiques propres à chaque architecture peuvent avoir un impact significatif sur la performance.

L'architecture du processeur Alpha possède deux pipelines : un pour les instructions entières et un pour les instructions flottantes. Chaque pipeline possède ses propres unités fonctionnelles et leurs queues d'instructions : la queue d'instructions entières et la queue d'instructions flottantes. La figure 2.14 représente le pipeline du processeur Alpha. Lorsque les queues d'instructions contiennent de nombreuses instructions, il se peut que tous les registres physiques du processeur soient utilisés, toute nouvelle instruction doit alors attendre qu'un registre se libère. Dans ce cas, le chargement des instructions doit être suspendu. Les étages de chargements (Fetch), de placement (Slot), et de renommage (Map) sont donc suspendus. Cependant, l'étage de renommage ne peut pas redémarrer immédiatement après la libération des registres : il doit attendre 2 cycles supplémentaires de suspension. L'étage de renommage peut également être suspendu lorsqu'une des deux queues d'instructions (entières ou flottantes) déborde. Dans le cas du produit de matrices, l'étage de renommage est régulièrement suspendu par un débordement de la

queue d'instruction flottantes. Pour éviter les suspensions de l'étage de renommage, nous devons diminuer le flux d'instructions flottantes. La solution, que nous avons choisie, est d'insérer des instructions fantômes (NOP). Ces instructions ne modifient pas les calculs du programme car elles ne s'exécutent pas : elles ne traversent qu'une partie du pipeline, de l'étage de chargement jusqu'à l'étage de renommage, ensuite elles sont retirées.

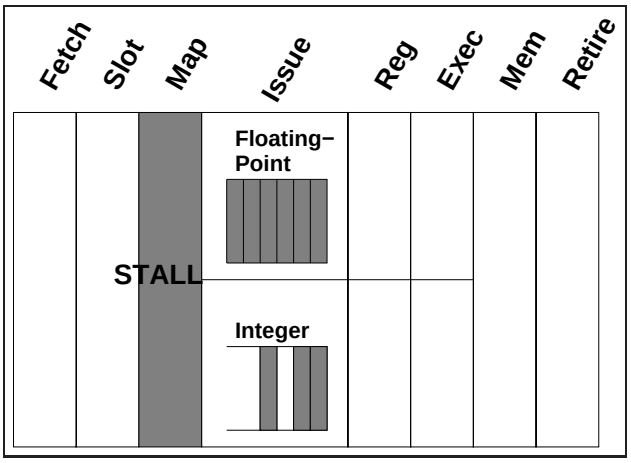


FIG. 2.14 – *Architecture du pipeline du processeur Alpha-21264*

Nous avons inséré ces instructions fantômes au niveau assembleur comme pour les instructions de préchargement et comme cela est indiqué dans la figure 2.15. Le nombre de suspensions du pipeline dues aux débordements de la queue d'instructions flottantes a diminué de 60%. Le gain final est de 13,56 par rapport au programme initial.

```
add   f18,f12,f18
mul   f14,f26,f12
ld    f8 , 0*SIZE(r3)
add   f17,f11,f17
-------------------------------
mul   f14,f27,f11
ld    f19, 0*SIZE(r5)
...
...
-------------------------------
...
```

```
add   f18,f12,f18
nop
mul   f14,f26,f12
ld    f8 , 0*SIZE(r3)
-------------------------------
add   f17,f11,f17
nop
mul   f14,f27,f11
ld    f19, 0*SIZE(r5)
-------------------------------
...
```

FIG. 2.15 – *Insertion des instructions fantômes*

2.4 Conclusions

Dans ce chapitre, nous avons montré la complexité du comportement de programmes aussi simples que le produit de matrices. Cette complexité est liée à la complexité de l'architecture du processeur et aux nombreuses interactions entre les différents composants. Nous avons également montré qu'une analyse dynamique très précise peut aider à comprendre les phénomènes complexes qui se produisent lors d'une exécution. L'analyse dynamique peut ainsi guider un processus d'optimisation. Cette étude montre également que les processus d'optimisation ne doivent pas

uniquement se focaliser sur les transformations de programmes les plus *populaires* comme le blocage pour les caches, dont la contribution au gain final est seulement de 12% ici, mais ils doivent prendre en compte tous les composants de l'architecture.

Chapitre 3

Processus d'optimisation systématique

Dans le chapitre 1, nous avons vu que la complexité des architectures augmentant, les compilateurs ont beaucoup de mal à prendre les bonnes décisions lors des phases d'optimisation. Nous avons vu également que l'information obtenue par l'analyse statique n'est pas suffisante pour en déduire le comportement des programmes. Le chapitre 2 a montré que l'analyse dynamique détaillée permet d'obtenir des informations essentielles pour prendre les bonnes décisions lors du processus d'optimisation. Dans ce chapitre, nous présentons un processus d'optimisation basé sur une analyse dynamique détaillée. Ce processus est guidé par un arbre de décision. Chaque branche de l'arbre est une suite d'étapes d'analyses et de décisions basées sur des métriques d'analyses dynamiques. Nous appelons ces métriques des *anomalies de performance*. Chaque feuille de l'arbre identifie un problème de performance dominant, pour lequel nous suggérons une ou plusieurs optimisations. Une itération du processus consiste à réaliser une optimisation, à éventuellement l'adapter et à la généraliser à d'autres parties du programme. Progressivement, le processus crée des séquences ou des compositions de transformations de programmes et continue jusqu'à ce que les optimisations n'apportent plus suffisamment de gain de performance.

3.1 Identification des problèmes de performance sur une architecture complexe

Le processus d'optimisation est guidé par l'arbre de décision. La décision prise à chaque nœud est un choix systématique basé sur des métriques issues de l'analyse dynamique. Chaque décision est une étape dans l'identification du problème de performance dominant. Notre motivation principale était de trouver *la cause exacte de toutes les dégradations de performance* observées lors de l'exécution d'un programme et ainsi de trouver une optimisation adaptée. Dans un processeur superscalaire à exécution dans le désordre comme le processeur Alpha-21264, il n'est pas facile de trouver la cause exacte d'une *dégradation de performance*. Par exemple, lorsqu'une station de réservation pleine cause le gel du pipeline, il faut remonter la trace d'exécution des instructions se trouvant dans les stations de réservation pour trouver la cause exacte : il se peut qu'au départ une lecture mémoire fasse un défaut de cache, que celui-ci ralentisse l'exécution de plusieurs opérations arithmétiques, qui elles-mêmes ralentissent un calcul d'adresse, le calcul d'adresse ralentit l'exécution d'autres opérations mémoires pour enfin geler le pipeline. Les instructions fautives peuvent donc avoir quitté le processeur de nombreux cycles avant que la *dégradation*

de performance ne soit observée. D'autre part, les causes exactes de la même *dégradation de performance* peuvent être partagées entre de multiples événements.

Plutôt que de rechercher les causes exactes à l'instant où une *dégradation de performance* apparaît, nous avons choisi de relever tous les événements pouvant être à l'origine d'une *dégradation de performance* au moment même où ils apparaissent. Nous avons appelé ces différents événements des *anomalies de performance* et nous les combinons afin de déterminer, parmi plusieurs causes, la cause dominante (la plus *responsable* de la *dégradation de performance*). Lorsque nous relevons une anomalie lors de l'exécution d'une instruction, nous analysons uniquement les instructions parentes dans le graphe de flot de données, c'est-à-dire les instructions fournissant les opérandes. Par exemple, dans la figure 3.1 si une anomalie est révélée pour l'instruction `S6`, nous limitons notre analyse aux instructions `S4` et `S5`.

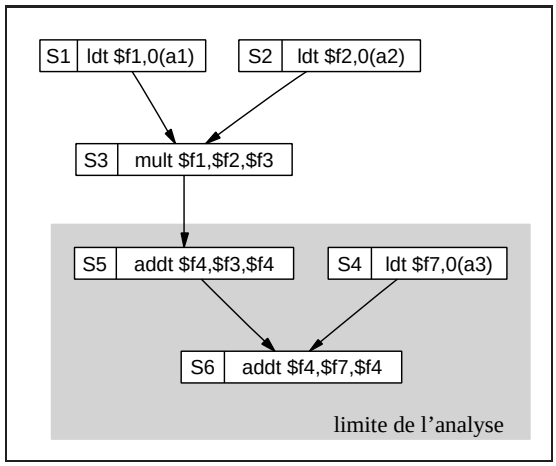


Fig. 3.1 – *Exemple de graphe de flot de données*

Pour sélectionner la transformation de programmes adaptée, il est nécessaire de caractériser au mieux la nature de la dégradation de performance. Le point de départ de l'analyse est basé sur l'observation de l'écart entre la performance soutenue et la performance crête. La performance crête est atteinte lorsque toutes les unités fonctionnelles du processeur sont utilisées à cent pour cent, c'est-à-dire à chaque cycle de l'exécution. Nous observerons donc une *dégradation de performance* à chaque fois qu'une unité fonctionnelle ne sera pas utilisée.

Si toutes les unités fonctionnelles sont utilisées à cent pour cent, alors le programme est limité par son nombre d'opérations. Dans le cas contraire, il y a une perte de performance qui peut être due à trois causes :

- une mauvaise utilisation de la hiérarchie mémoire,
- un manque de parallélisme,
- une mauvaise utilisation de composants spécifiques à l'architecture,
- ou bien encore, les trois à la fois.

Pour déterminer la cause principale, il faut observer des *Anomalies de performance* relatives aux unités fonctionnelles, aux queues de lancement associées (ou stations de réservations), à la nature des opérandes des instructions dans les queues de lancement, ou encore des *Anomalies de performance* relatives aux composants spécifiques de l'architecture.

Le raisonnement permettant d'identifier la cause dominante de la dégradation de performance est décrit par l'arbre de décision représenté dans la figure 3.2.

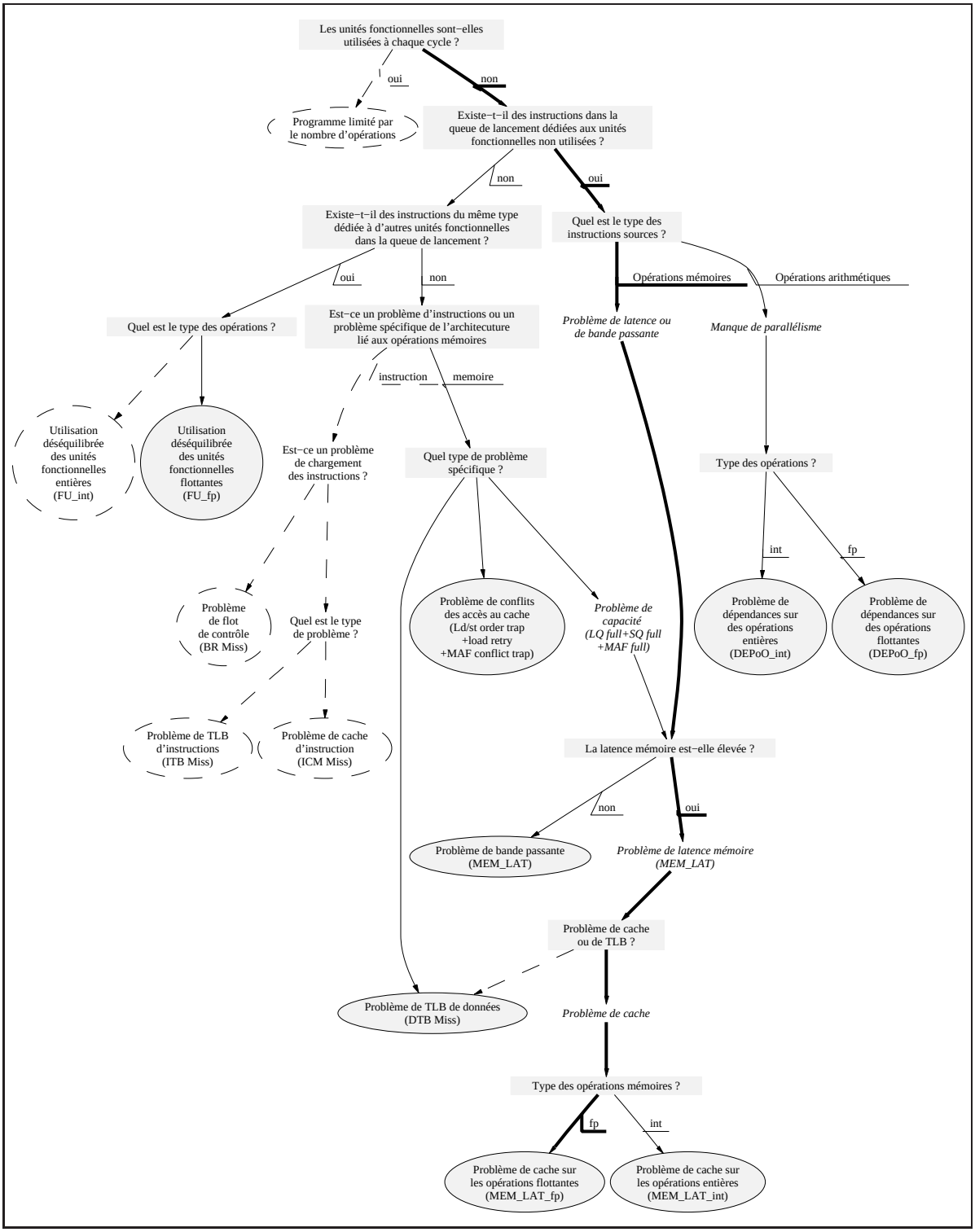


Fig. 3.2 – *Arbre de décision*

La branche soulignée par les flèches en gras donne un exemple de raisonnement identifiant un problème de *latence mémoire*. A la racine de l'arbre, nous constatons que les unités fonctionnelles ne sont pas utilisées à chaque cycle. Il peut y avoir deux explications à cela : la première est qu'au

cycle où l'unité fonctionnelle n'est pas utilisée, il n'y a aucune instruction dédiée à cette unité dans la queue de lancement. La seconde explication est qu'au cycle où l'unité fonctionnelle n'est pas utilisée, il y a bien une instruction dédiée à cette unité dans la queue de lancement mais au moins l'un de ses deux opérandes n'est pas prêt. Nous avons donc une *anomalie de performance* qui compte le nombre de cycle où une unité fonctionnelle n'est pas utilisée et où il n'y a pas d'instruction dédiée dans la queue de lancement, et une autre *anomalie de performance* qui, chaque fois qu'une unité fonctionnelle n'est pas utilisée, compte le nombre d'opérandes non prêts appartenant aux instructions dédiées à cette unité, dans la queue de lancement. Nous verrons plus loin dans la section 3.2 comment comparer des anomalies de performance ayant des probabilités d'occurrence et des impacts sur la performance différents. Mais supposons ici que l'anomalie de performance dominante soit la seconde (opérandes non prêts), dans ce cas la cause dominante peut être soit un manque de parallélisme, soit un problème mémoire. Pour distinguer ces deux cas, l'anomalie de performance est raffinée en deux sous-anomalies, c'est-à-dire que chaque fois qu'une unité fonctionnelle n'est pas utilisée et qu'une instruction dédiée se trouve dans la queue de lancement, nous observons la nature des instructions devant fournir les opérandes non prêts. Si ces instructions sont majoritairement des opérations arithmétiques alors la cause dominante sera un manque de parallélisme. Si, au contraire, ces instructions sont en majorité des opérations de lecture mémoire alors la cause dominante sera un problème mémoire. Supposons, dans notre exemple que nous ayons affaire à un problème mémoire, nous devons alors déterminer si la cause dominante est un problème de *bande passante* ou bien un problème de *latence*. Pour cela, nous combinons l'anomalie précédente avec une nouvelle anomalie : la latence mémoire. Si la latence mémoire est élevée alors le problème est identifié comme étant un problème de latence sinon il est identifié comme étant un problème de bande passante. L'identification du problème de performance a pour objectif de trouver la transformation de programmes la plus adaptée.

Pour collecter les anomalies de performance, nous observons le comportement des instructions sur l'architecture à chaque cycle. Considérons l'exemple de la figure 3.1 et supposons que l'exécution de celui-ci illustre l'exemple du problème de performance décrit ci-dessus. Pour un cycle donné, l'unité fonctionnelle d'addition n'est pas utilisée et l'instruction `S6` est dans la queue de lancement. Son opérande `f4` est prêt alors que son opérande `f7` ne l'est pas. L'instruction `S4` est l'instruction source de `f7`, elle est en cours d'exécution. Cette instruction est une lecture mémoire, nous incrémentons donc le compteur d'anomalies de type *dépendances sur une opération mémoire*. Indépendamment et parallèlement, nous mesurons la latence de l'opération de lecture mémoire. Si ce cas de figure se présente souvent alors l'anomalie de dépendances sur des opérations mémoires et l'anomalie latence mémoire seront élevées, nous en déduirons un problème de *latence mémoire*. Au contraire, si la latence mémoire est faible alors nous en déduirons un problème de *bande passante*.

3.2 Normalisation des anomalies

La table 3.1 présente la hiérarchie des anomalies utilisées dans le processus d'identification formalisé par l'arbre de décision. Les quatre principales catégories d'anomalies sont :

- les anomalies de dépendances (`DEP`),
- les anomalies d'utilisation des unités fonctionnelles (`FU`),
- la latence mémoire (`MEM`),

– les purges de pipeline dues à des comportements spécifiques à l'architecture des composants (TRAPS).

Statistiques	Descriptions	m_a	σ_a
DEP	**Les anomalies de dépendances** (DEP_fp + DEP_int)	**7.38**	**5.08**
DEP_fp	DEPoO_fp + DEPoM_fp		
DEPoO_fp	Nombre d'opérandes, en cours de calcul, des instructions (se trouvant dans la queue de lancement) dédiées à une unité fonctionnelle flottante non utilisée.		
DEPoM_fp	Nombre d'opérandes, en cours de chargement (par des opérations de lecture mémoire), des instructions (se trouvant dans la queue de lancement) dédiées à une unité fonctionnelle flottante non utilisée.		
DEP_int	DEPoO_int + DEPoM_int		
DEPoO_int	Nombre d'opérandes, en cours de calcul dans les unités fonctionnelles, des instructions (se trouvant dans la queue de lancement) dédiées à une unité fonctionnelle entière non utilisée.		
DEPoM_int	Nombre d'opérandes, en cours de chargement (par des opérations de lecture mémoire), des instructions (se trouvant dans la queue de lancement) dédiées à une untié fonctionnelle entière non utilisée.		
FU	**Anomalies d'utilisation des unités fonctionnelles** (FU_fp + FU_int)	**2.09**	**0.54**
FU_fp	Nombre de fois où une unité fonctionnelle flottante n'est pas utilisée sans qu'il y ait une seule instruction dédiée dans la queue de lancement		
FU_int	Nombre de fois où une unité fonctionnelle entière n'est pas utilisée sans qu'il y ait une seule instruction dédiée dans la queue de lancement		
MEM_LAT	**Latence mémoire de toutes les opérations de lecture**	**12.84**	**8.13**
MEM_LAT_fp	Latence des opérations de lecture flottantes		
MEM_LAT_int	Latence des opérations de lecture entières		
Traps	**Anomalies spécifiques aux comportements de certains composants de l'architecture** (ITraps + DTraps)	**0.005**	**0.041**
ITraps	BR Miss + IC Miss + ITB Miss		
BR Miss	Nombre de mauvaises prédictions de branchements		
IC Miss	Nombre de défauts dans le cache d'instructions		
ITB Miss	Nombre de défauts dans le cache de traductions d'adresses d'instructions		
DTraps	DTB Miss + Cache conflict + LSQ/MAF full		
DTB Miss	Nombre de défauts dans le cache de traductions d'adresses de données		
Cache conflict	Nombre de conflits d'accès au cache dus à une opération de lecture et d'écriture qui accèdent à la même ligne de cache		
LSQ/MAF full	Nombre de fois où la queue d'instructions de lecture, la queue d'écriture, ou bien la queue de opérations en cours d'échec dans le cache sont pleines et provoquent un gel de pipeline.		

Tab. 3.1 – *Anomalies utilisées pour identifier les problèmes de performance*

Il est important de noter que les anomalies de dépendance sont comptabilisées uniquement si des unités fonctionnelles ne sont pas utilisées et si les opérandes sont en cours de calcul. Cela permet de comptabiliser un problème de dépendance uniquement au moment où il se produit. En effet, si on compte tous les opérandes non prêts, y compris ceux devant être fournis par des instructions se trouvant dans la queue de lancement, alors l'anomalie est polluée. Une dépendance entre deux instructions dans la queue de lancement ne doit pas être comptabilisée car il se peut que, pendant les cycles d'exécution de l'instruction devant fournir l'opérande, toutes les unités fonctionnelles soient utilisées.

Pour déterminer la cause dominante d'une dégradation de performance, nous devons pouvoir comparer les différentes anomalies entre elles. Nous avons donc normalisé les anomalies. La

première normalisation est simple et permet de comparer une même anomalie entre des parties différentes de programmes. Les métriques absolues des anomalies sont divisées par le nombre de cycles sauf pour les anomalies de latence mémoire qui sont divisées par le nombre de lectures mémoires. Nous noterons ν_a la métrique normalisée d'une anomalie a. La seconde normalisation permet de comparer deux anomalies différentes pour une même partie de programmes. Nous voulons également savoir si une anomalie est élevée ou pas. Pour cela, nous avons collecté les anomalies pour l'ensemble des programmes de la suite SpecFP2000 et nous avons défini une métrique normalisée ν_a^{norm} de la façon suivante :

$$\nu_a^{norm} = \frac{\nu_a - m_a}{\sigma_a}$$

où m_a et σ_a sont respectivement la moyenne et l'écart type de la métrique v_a sur l'ensemble des programmes de la suite SpecFP2000[1]. Par conséquent, pour une anomalie a donnée, la moyenne de la métrique normalisée ν_a^{norm} sur l'ensemble des programmes est égale à 0 et son écart type est égal à 1. D'une manière plus intuitive, cela signifie que nous utilisons l'ensemble des programmes de la suite comme point de référence des anomalies.

Nous utilisons la normalisation uniquement au début de l'arbre pour comparer des anomalies qui n'ont aucune relation entre elles et qui sont difficilement comparables en valeur absolue. Par exemple, nous comparons des anomalies de type dépendance (`DEP`) avec des anomalies de type unités fonctionnelles (`FU`) ou bien encore avec des anomalies de type latence mémoire (`MEM`). Dans la suite de l'arbre, nous utilisons directement les valeurs absolues des anomalies. Par exemple, les anomalies de dépendances sur la mémoire ou de dépendances sur les opérations arithmétiques peuvent être comparées en valeur absolue. La moyenne et l'écart-type des métriques ν_a des anomalies sont donnés dans la table 3.1.

3.3 Construction de l'arbre de décision

L'arbre de décision a deux objectifs :
- formaliser l'analyse d'un *humain expert* permettant d'identifier les problèmes de performance à l'aide d'informations dynamiques détaillées,
- mémoriser l'association, faite par l'*humain expert*, entre les problèmes de performance identifiés par l'analyse et les transformations de programmes utilisées.

Dans la sous-section suivante, nous décrivons la démarche que nous avons suivi pour construire l'arbre de décision de la figure 3.2.

3.3.1 Construction empirique

Pour construire l'arbre, nous avons optimisé manuellement un ensemble de programmes pour une architecture spécifique. Nous avons utilisé 12 programmes de calculs flottants parmi les programmes de la suite *SpecFP2000*. L'architecture cible est le processeur Alpha-21264 superscalaire d'ordre 4 à exécution dans le désordre. Nous avons utilisé le simulateur cycle à cycle du processeur Alpha pour collecter un grand nombre d'informations sur le comportement des programmes. Nous avons optimisé manuellement un premier programme (applu) à l'aide de ces informations.

[1] Les métriques sont collectées par la simulation des programmes compilés avec les options des *SpecFP2000 Bases*

Pour ce premier programme, le travail d'optimisation n'était pas réellement structuré : l'objectif était d'obtenir une séquence d'optimisations offrant les meilleures performances possibles. Ensuite, nous avons inversement examiné cette séquence d'optimisations pour en extraire l'analyse des informations du simulateur permettant d'obtenir cette séquence. Cette analyse a ensuite été formalisée sous la forme d'un arbre de décision. L'optimisation de chaque nouveau programme nous a permis de raffiner et d'étendre l'arbre : pour chaque nouvelle analyse, nous avons construit une nouvelle branche. Le but de la formalisation de l'analyse est de pouvoir réutiliser les optimisations associées aux analyses (feuilles des branches de l'arbre). Pour que la réutilisation soit possible, il est nécessaire de trouver un compromis dans le niveau de détail de l'analyse. En effet, si l'analyse n'est pas suffisamment précise alors les branches de l'arbre n'identifieront pas les problèmes de manière suffisamment précise pour donner les bonnes solutions (optimisations). En revanche, si l'analyse est trop détaillée alors chaque feuille de l'arbre correspondra à un problème bien particulier dont la solution proposée est spécifique à chaque programme optimisé. Dans ce cas, la réutilisation devient impossible. Pour que l'association entre un problème de performance et une optimisation soit la plus indépendante possible de la structure des programmes, nous formalisons uniquement l'analyse dynamique permettant d'identifier un problème de performance sur un composant de l'architecture. Contrairement à ce qui est fait dans les compilateurs, nous ne formalisons pas l'analyse statique que nous effectuons sur les programmes pour réaliser les transformations.

Nous avons poursuivi l'optimisation des autres programmes en systématisant la démarche et le processus que nous décrivons plus loin dans la section 3.4. La formalisation de l'analyse basée sur les informations dynamiques détaillées lors de l'optimisation des programmes est représentée dans l'arbre de la figure 3.2. Les branches en pointillés représentent des problèmes que nous avons identifiés mais auxquels nous n'avons pas été confrontés. Ce sont donc des problèmes pour lesquels nous n'avons pas fait d'optimisations, par exemple, les problèmes de cache d'instructions.

3.4 Processus d'optimisation

Les processus d'optimisation itératifs proposés jusqu'à aujourd'hui [KiKnOB00] sont basés sur la recherche des paramètres d'un nombre restreint de transformations. Si l'on désire étendre ces processus itératifs à des nombres importants de transformations alors l'espace de recherche des paramètres croît de manière exponentielle et le temps de recherche devient dissuasif. Notre approche de l'optimisation itérative n'est pas conçue pour rechercher la performance optimale mais plutôt pour assurer une progression de performance sur chaque composant de l'architecture au fur et à mesure des itérations. Nous avons observé expérimentalement que cette approche apporte des gains de performance significatifs.

3.4.1 Description du processus

Le processus d'optimisation itératif que nous avons utilisé est relativement simple. Chaque itération du processus se décompose de la façon suivante :

1. collecter les *Anomalies de performance*,
2. utiliser l'arbre de décision afin d'identifier le problème de performance dominant,

3. localiser, dans le programme source (ou éventuellement dans le programme assembleur), la partie de code où le problème de performance est le plus présent,
4. sélectionner, parmi les optimisations associées à chaque problème de performance, l'optimisation qui améliore au mieux le problème localement,
5. effectuer des itérations d'adaptations,
6. généraliser les optimisations à d'autres parties du programme,
7. recommencer une nouvelle itération.

Collecte des Anomalies de performance Nous avons modifié le code source du simulateur du processeur Alpha en insérant des sondes permettant de compter tous les événements listés dans la table 3.1 ainsi que le nombre de cycles et le nombre d'opérations mémoires (nécessaire pour la normalisation des anomalies). Avant de démarrer le processus d'optimisation pour chaque programme, nous avons effectué un profil en temps. Ces profils nous ont permis d'identifier l'ensemble des procédures dont le temps cumulé correspond à environ 90% du temps total de chaque exécution. Le temps de simulation d'un programme complet de la suite SpecFP2000 pouvant atteindre plusieurs mois, nous avons simulé uniquement les procédures sélectionnées suivant les profils et nous avons limité chaque simulation à 4 milliards d'instructions exécutées. Le processus d'optimisation est fortement séquentiel : les simulations pour la collecte des anomalies ne peuvent pas être effectuées en parallèle.

Utilisation de l'arbre de décision Une fois les anomalies de performance collectées, l'utilisation de l'arbre de décision est totalement systématique : les réponses aux questions posées dans l'arbre de décision sont les vérifications d'équations ou d'inéquations dont les variables sont les anomalies de performance. Par exemple, la réponse à la première question :

"les unités fonctionnelles sont-elles utilisées à chaque cycle ?"

est "non" si l'inéquation suivante est vérifiée :

$$\texttt{DEP} + \texttt{FU} > 0$$

La figure 3.3 représente l'arbre de décision précédemment décrit avec les équations et les inéquations sur les arcs permettant d'identifier le problème de performance de manière systématique.

Localisation dans le programme source Après l'identification d'un problème dominant, il est nécessaire de localiser ce problème, c'est-à-dire trouver la région responsable. Il est donc nécessaire que la collecte des anomalies de performance conserve l'association entre une anomalie et une instruction. Lors de la simulation, nous associons le plus précisément possible les anomalies aux instructions assembleurs en cours d'exécution :

- les anomalies de dépendance (`DEP`) sont associées aux instructions se trouvant dans la queue de lancement pour lesquelles les opérandes ne sont pas prêtes.
- les anomalies d'utilisation des unités fonctionnelles (`FU`) ne peuvent pas être associées à une instruction puisqu'elle révèle un manque d'instruction dédiée à une unité fonctionnelle. Cette anomalie a donc été associée au bloc de base des instructions en cours d'exécution,

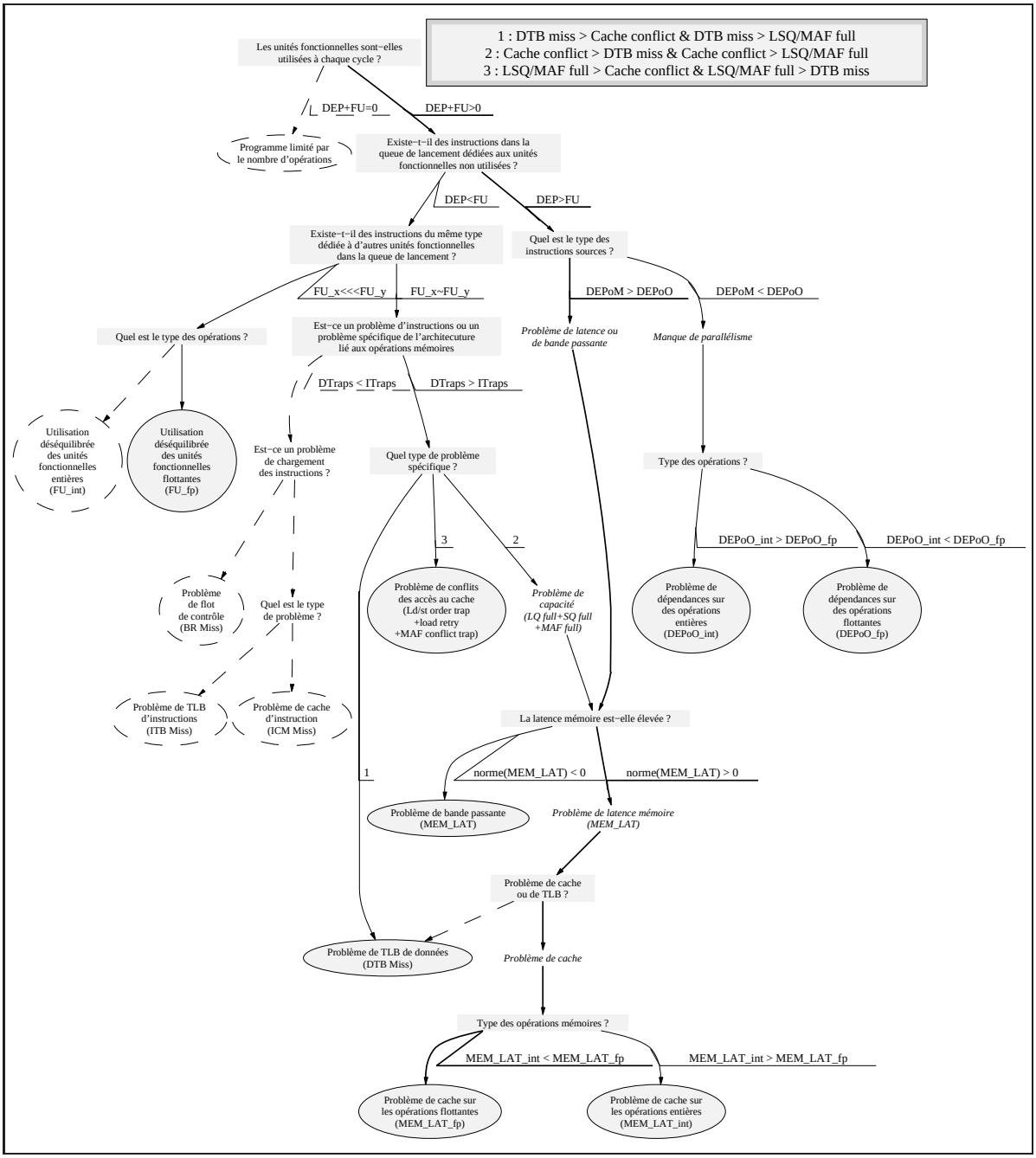


Fig. 3.3 – *Arbre de décision systématique*

c'est-à-dire au bloc de base de l'instruction la plus proche de l'anomalie : la première instruction trouvée soit dans les queues de lancement, soit dans les étages précédents du pipeline.

– les anomalies de latence mémoire (`MEM_LAT`) sont associées aux instructions assembleurs réalisant les chargements mémoires. La latence des instructions de préchargement n'est pas comptabilisée dans les anomalies de latence mémoire.
– les anomalies spécifiques à l'architecture (`Traps`) sont comptabilisées suivant les instructions assembleurs.

Sélection d'une optimisation C'est la partie du processus qui demande le plus de travail à l'utilisateur. Mais c'est également celle qui laisse le plus de liberté. Il doit choisir une optimisation adaptée à la partie de programme identifiée lors de la localisation. L'utilisateur doit vérifier la légalité et doit valider ses optimisations.

Lors de la construction de l'arbre de décision, les utilisateurs peuvent suggérer des optimisations adaptées et les rajouter aux feuilles de l'arbre.

Itérations d'adaptation Les itérations d'adaptation sont quasiment identiques aux itérations générales du processus. Elles diffèrent uniquement au niveau de la phase d'identification du problème de performance. Lors de ces itérations, le processus ne se focalise pas sur le problème de performance dominant mais se focalise uniquement sur des augmentations de problèmes en observant les augmentations d'anomalies de performance. Les itérations d'adaptation permettent d'ajuster le paramétrage d'une optimisation ou bien de composer des transformations de programmes afin d'améliorer finement le comportement local du programme sur l'ensemble des composants de l'architecture.

Considérons, par exemple, le programme APPLU dont la première itération a nécessité une adaptation. Les anomalies de performance et l'arbre de décision nous ont permis d'identifier un problème de dépendances sur des opérations arithmétiques flottantes localisé dans la procédure `blts`, plus précisément sur l'opération de soustraction `v(m,i,j,k)-omega` représentée sur la figure 3.4. Cette soustraction dépend de 4 multiplications et de 2 additions. Nous avons bru-

```
do k = 2, nz-1
  do j = 2, ny-1
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v(m,i,j,k) = v(m,i,j,k) - omega *
             ( ldz(m,l,i,j,k) * v(l,i,j,k-1)
             + ldy(m,l,i,j,k) * v(l,i,j-1,k)
             + ldx(m,l,i,j,k) * v(l,i-1,j,k)
```

Fig. 3.4 – *Dépendance sur les instructions de la procédure* `blts`

talement optimisé pour réduire au maximum le problème de dépendances : nous avons scindé le nid de boucle (`m,l`) en deux en scindant l'instruction pour séparer les multiplications du calcul. De plus, nous avons effectué un décalage d'itération (*shifting*) du second nid créé par la fission de boucles. La partie de programme résultante est montrée sur la figure 3.5. Après cette optimisation, nous avons procédé à une itération d'adaptation : nous avons collecté de nouvelles anomalies et nous avons observé la réduction des anomalies de dépendances sur des opérations arithmétiques flottantes. En revanche, nous avons constaté une augmentation des anomalies de dépendances sur des opérations arithmétiques entières localisées sur les instructions de rangements en mémoire des tableaux temporaires dans le second nid de boucle. Nous avons donc choisi de réduire les instructions de rangements mémoires en modifiant légèrement la fission de boucle : nous avons regroupé les additions et les multiplications dans le même nid de boucle pour réduire le nombre de tableaux temporaires. Le programme correspondant est représenté sur la figure 3.6

La table 3.2 montre l'évolution des anomalies de dépendances entières (`DEP_int`) et flottantes

```
do k = 2, nz-1
  do j = 2, ny-1
    ...
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v( m, i, j, k ) =  v( m, i, j, k )
            - omega * (  tmpldz(m,l)
                       + tmpldy(m,l)
                       + tmpldx(m,l) )
      do m = 1, 5
        do l = 1,5
          tmpldx(m,l) = ldx( m, l, i+1, j, k )
                      * v( l, i-1+1, j, k )
          tmpldy(m,l) = ldy( m, l, i+1, j, k )
                      * v( l, i+1, j-1, k )
          tmpldz(m,l) = ldz( m, l, i+1, j, k )
                      * v( l, i+1, j, k-1 )
    ...
```

Fig. 3.5 – *Fission de boucle et décalage d'itération (procédure* `blts`*)*

```
do k = 2, nz-1
  do j = 2, ny-1
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v( m, i, j, k ) =  v( m, i, j, k )
              - omega * tmpldxyz(m,l)
      do m = 1, 5
        do l = 1,5
          tmpldx = ldx( m, l, i+1, j, k )
                 * v( l, i-1+1, j, k )
          tmpldy = ldy( m, l, i+1, j, k )
                 * v( l, i+1, j-1, k )
          tmpldz = ldz( m, l, i+1, j, k )
              * v( l, i+1, j, k-1 )
          tmpldxyz(m,l) = tmpldx + tmpldy + tmpldz
```

Fig. 3.6 – *Modification de la fission de boucle (procédure* `blts`*)*

(`DEP_fp`), en nombre d'anomalies par cycle, au cours des différentes optimisations.

Anomalies	DEP_fp	DEP_int
original	5.50	2.49
itération 1	3.78	3.65
adaptation 1	4.03	3.31

Tab. 3.2 – *Évolution des anomalies de dépendances.*

Les itérations d'adaptation peuvent agir récursivement les unes sur les autres, le processus doit donc fixer une limite au nombre d'adaptations par itération. Nous nous sommes limités à deux ou trois adaptations par itération.

Généralisation Le paragraphe 3.4.1 décrit la localisation des anomalies et donc des problèmes de performance. Lors de la construction de l'arbre de décision et de l'optimisation des différents programmes, nous avons observé à plusieurs reprises qu'un même problème de performance pouvait être localisé dans des parties de programmes distantes. La raison est que plusieurs parties d'un même programme peuvent parfois avoir la même structure, le même comportement et donc les mêmes problèmes de performance. Nous avons ajouté une sixième étape à la fin du processus d'optimisation : la *généralisation*. Elle consiste à rechercher et à localiser des parties de programmes avec des problèmes de performance identiques à ceux de l'itération en cours afin d'appliquer systématiquement les mêmes optimisations. La généralisation peut reproduire la transformation de l'optimisation principale, celle d'une des adaptations, ou bien encore plusieurs simultanément.

Cette étape de généralisation peut parfois être suivie d'étapes d'adaptation. En effet, même si la généralisation s'applique sur des parties de programmes similaires, elles ne sont pas identiques. Le comportement des parties du programme optimisées par la généralisation peut parfois être différent des autres parties du programme optimisées et peut donc nécessiter la réalisation d'étapes d'adaptation.

Exemple APPLU Pour illustrer le fonctionnement de l'étape de généralisation, nous considérons une nouvelle fois l'exemple du programme APPLU. Lors de la première itération, nous avions identifié un problème de dépendances sur des opérations arithmétiques localisé sur le nid de boucle de la procédure `blts`. La localisation d'autres instructions exprimant le même problème de performance donne des instructions ayant une structure très similaire appartenant à la procédure `buts` (figure 3.7).

```
do k = 2, nz-1
  do j = 2, ny-1
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v(m,i,j,k) = v(m,i,j,k) - omega *
             ( ldz(m,l,i,j,k) * v(l,i,j,k-1)
             + ldy(m,l,i,j,k) * v(l,i,j-1,k)
             + ldx(m,l,i,j,k) * v(l,i-1,j,k)
```

procédure `blts`

```
do k = nz-1, 2
  do j = 2, ny-1, 2
    do i = 2, nx-1, 2
      do m = 1, 5
      tv(m) = 0.0d+00
        do l = 1, 5
          tv(m) = t(m) - omega *
              ( udz(m,l,i,j,k) * v(l,i,j,k+1)
              + udy(m,l,i,j,k) * v(l,i,j+1,k)
              + udx(m,l,i,j,k) * v(l,i+1,j,k)
```

procédure `buts`

Fig. 3.7 – *Généralisation des optimisations de `blts` sur `buts`*

Supposons que l'on utilise le processus d'optimisation sans généralisation. Si un problème de performance dominant est réparti sur deux parties de programmes distinctes alors l'optimisation de la première partie peut changer le problème dominant. Si le processus d'optimisation n'échoue pas et s'il est conduit suffisamment longtemps alors le problème de performance localisé sur la seconde partie de programmes pourra redevenir le problème dominant. La généralisation permet d'exploiter au mieux le bénéfice des optimisations qui peut être parfois partiellement perdu par le processus sans généralisation.

Avant la généralisation, une étape de validation de l'optimisation est nécessaire pour vérifier si elle a été bénéfique. Cette validation consiste à exécuter le nouveau programme et à collecter les

Contributions de la généralisation	
EQUAKE	9%
WUPWISE	41%
APPLU	38%
MGRID	6%
GALGEL	8%
Moyenne	20%

Tab. 3.3 – *Contribution de la généralisation*

anomalies afin de constater la réduction du problème de performance. Dans le cas du programme WUPWISE, l'application du processus sans généralisation a réduit le temps de 232 secondes à 135 secondes ; la généralisation à réduit le temps d'exécution à 80 secondes.

La table 3.3 montre la contribution (en pourcentage) de la généralisation au gain final.

3.4.2 Décomposition des problèmes de performance

Comme nous l'avons expliqué précédemment dans la section 3.4, le processus n'adresse qu'un seul problème de performance par itération (le problème dominant). Adresser plusieurs problèmes simultanément permettrait certainement d'accélérer le processus. Cependant, il serait beaucoup plus difficile d'adapter finement les optimisations comme nous venons de le voir dans la sous-section précédente. D'autre part, dans certains cas, il arrive qu'un problème soit caché par un autre et ce n'est qu'une fois le problème dominant réduit que le second problème apparaît. Le processus doit adresser les problèmes un à un pour garantir la progression du processus.

Exemple APPLU Considérons à nouveau l'exemple du programme APPLU, dont la première itération et son adaptation ont été décrites précédemment. Nous avons donc scindé le nid de boucle (`m,1`) en deux comme le montre la figure 3.6. A l'itération suivante, le problème de dépendance sur des opérations arithmétiques n'est plus le problème dominant. Le nouveau problème dominant est un problème de latence mémoire localisé sur les références aux tableaux `ldx`, `ldy` et `ldz`. Ce nouveau problème n'a pas été créé par la "fission de boucle", il était déjà visible lors de la précédente itération car les anomalies de dépendance sur des opérations mémoires et la latence mémoire étaient déjà élevées (voir le détail de l'optimisation du programme APPLU annexe A.3). Cet exemple illustre le fait qu'il existe des cas pour lesquels il serait possible de traiter simultanément plusieurs problèmes de performance.

Exemple EQUAKE Cet autre exemple illustre les cas où certains problèmes sont latents et masqués par des problèmes plus importants. La figure 3.8 montre une partie de la procédure `smvp` du programme EQUAKE et la partie du programme principal déclarant les variables du programme. Nous avons identifié un problème de latence mémoire sur les accès aux tableaux `M`, `M23`, `C23` et `V23`. Ce problème est dû à l'utilisation de tableaux de pointeurs sur des flottants en double précision. Nous avons donc optimisé le programme en transformant ces tableaux en tableaux flottants doubles précisions (voir figure 3.9). Après cette optimisation, nous avons observé une augmentation de prés de 50% des anomalies spécifiques à l'architecture (`Traps`). Cette augmentation est plus précisément située sur des problèmes de conflits des accès mémoires. Nous

```
/* Mass matrix */
M = (double **) malloc(ARCHnodes * sizeof(double *))
if (M == (double **) NULL) {
  fprintf(stderr, "malloc failed for M")
  fflush(stderr)
  exit(0)
for (i = 0; i < ARCHnodes; i++) {
  M[i] = (double *) malloc(3 * sizeof(double))
  if (M[i] == (double *) NULL) {
    fprintf(stderr, "malloc failed for M[%d]",i)
    fflush(stderr)
    exit(0)
```

```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i*3+j] += 2.0 * M[i][j] * disp[dispt][i*3+j] -
         (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i*3+j] -
          Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
                             C23[i][j] * phi1(time) / 2.0 +
                             V23[i][j] * phi0(time) / 2.0);
```

Fig. 3.8 – *Déclaration et procédure* ***smvp***

avons localisé ces problèmes de conflits sur les accès aux fonctions `phi0`, `phi1` et `phi2` dans le même nid de boucle représenté sur la figure 3.9. La variable `time` étant constante dans le nid de boucle, nous avons réalisé une optimisation scalaire en plaçant les appels de fonctions à l'extérieur du nid de boucle. Le problème de conflits était bien présent avant l'optimisation pour la latence mémoire mais les anomalies de latence mémoire avaient pour effet de masquer ce problème. Cet exemple montre que certaines anomalies peuvent être dissimulées par d'autres. Il convient donc de traiter les anomalies une à une afin de démasquer toutes les anomalies et de garantir ainsi la progression du processus d'optimisation.

```
M = (double *) malloc(ARCHnodes * 3 * sizeof(double));
if (M == (double *) NULL) {
  fprintf(stderr, "malloc failed for M");
  fflush(stderr);
  exit(0);
```

```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
    (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
           Exc.dt * Exc.dt * (M23[i*3+j] * phi2(time) / 2.0 +
           C23[i*3+j] * phi1(time) / 2.0 +
           V23[i*3+j] * phi0(time) / 2.0);
```

Fig. 3.9 – *Optimisation globale des structures de données pour la procédure* ***smvp***

3.4.3 Annulation des itérations

La réduction d'un problème de performance ne garantit pas toujours un gain de performance. En effet, nous avons vu dans la section 3.4.2 que la réduction d'un problème de performance peut démasquer un nouveau problème de performance. Or, le nouveau problème de performance étant plus pénalisant que le précédent, le gain de performance peut être inférieur à 1. Nous autorisons le processus à procéder au plus à 2 itérations consécutives avec des gains inférieurs à 1. En contrepartie, nous autorisons le processus à annuler les itérations ayant généré des gains inférieurs à 1. Par exemple, lors de l'optimisation du programme GALGEL, nous avons effectué une fusion de boucle qui nous a permis de réduire un problème de TLB de données. Suite à cette optimisation, nous avons observé un gain de 0.97 et une augmentation du problème de dépendances sur des opérations arithmétiques entières. Nous avons conservé l'optimisation dégradant le gain et nous avons composé la fusion de boucle avec un décalage d'itération (*shifting*) pour réduire le problème de dépendances. Le gain a atteint la valeur de 1 et nous avons poursuivi le processus d'optimisation.

Nous avons également autorisé le processus à annuler des optimisations ayant des gains supérieurs à 1. Dans certains cas, après une itération avec un gain positif, le problème de performance de l'itération suivante nécessite d'effectuer une optimisation qui ne peut être réalisée que si la première optimisation est annulée. Par exemple, la seconde itération de l'optimisation du programme APPLU nécessite l'annulation de la première itération : la fusion de boucle de la seconde itération ne peut avoir lieu que si le décalage d'itérations est supprimé. En effet, le décalage d'itérations crée des dépendances inter-itérations qui inhibent la fusion des boucles.

3.4.4 Comparaison avec un compilateur statique

Le principal avantage d'un tel processus itératif par rapport à un compilateur statique est la flexibilité. Il permet de construire de longues séquences de transformations dont l'ordre est défini par la domination des problèmes de performance, contrairement aux compilateurs statiques où l'ordre des transformations est prédéfini et constant. Par exemple, la figure 3.10 montre une séquence de 7 transformations obtenue pour le programme GALGEL. L'autre avantage provient de l'application manuelle des transformations qui permet de dépasser les limites imposées aux compilateurs statiques par la représentation intermédiaire généralement basée sur les structures d'arbres syntaxiques.

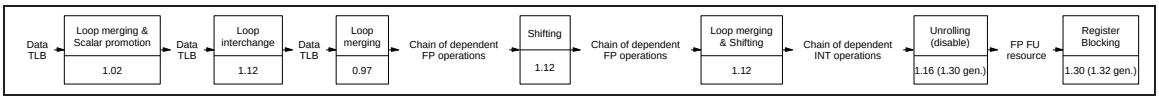


Fig. 3.10 – *Séquence de transformations réalisée sur le programme* GALGEL

Premièrement, un compilateur a généralement une stratégie d'optimisation prédéfinie limitant les variations des compositions de transformations. Par exemple, dans le compilateur ORC [ORC], les optimisations de fusion et de fission de boucles peuvent être répétées et interchangées mais ce n'est pas le cas des autres transformations. Les compilateurs manquent de flexibilité pour appliquer des séquences de transformations variables. Deuxièmement, chaque transformation de programmes est généralement appliquée syntaxiquement, c'est-à-dire que pour chaque transformation une nouvelle version de l'arbre syntaxique est générée. De telles représentations sont

incompatibles avec de longues séquences de transformations. En effet, certaines optimisations peuvent modifier l'arbre syntaxique de telle sorte que plusieurs optimisations ne sont plus applicables à l'arbre modifié. Des travaux sont en cours [BaCoGi03] sur le développement d'un outil d'application de transformations de programmes basés sur une représentation intermédiaire compatible avec de longues compositions de transformations.

3.5 Résultats expérimentaux

Cette section a deux objectifs : le premier est d'illustrer en détail le processus d'optimisation au travers de l'optimisation complète d'un programme de la suite SpecFP2000. Le second objectif est de donner les résultats expérimentaux obtenus avec l'ensemble des autres programmes. Les détails des analyses et des optimisations de trois programmes sont présentés en annexe A.

Les expériences ont été réalisées sur une machine HP AlphaServer ES45 dotée de 4 processeurs Alpha 21264C (EV68) cadencés à 1GHz avec 8 Méga-octets de mémoire cache de second niveau et 8 Giga-octets de mémoire principale. Nous avons comparé les performances des versions optimisées des programmes avec les performances *Base* des SpecFP2000 obtenues avec une compilation identique pour tous les programmes : `-arch ev6 -fast -O5 ONESTEP`, avec les compilateurs HP Fortran (V5.4) et C (V6.4). Nous avons également comparé les performances des versions optimisées des programmes avec les performances *Peak* des SpecFP2000 obtenue avec des options de compilation particulières pour chaque programme utilisant parfois le préprocesseur KAP Fortran (V4.3) et utilisant parfois les optimisations dirigées par des informations dynamiques. La table 3.4 montre les gains obtenus par compilation en *Peak* et par le processus itératif par rapport à la compilation en *Base*. Elle montre également le gain relatif. Du point de

	Compilé en *Peak*	Processus Itératif	*Gain Relatif*
WUPWISE	1.20	2.90	*2.42*
EQUAKE	2.01	2.50	*1.24*
APPLU	1.47	2.18	*1.48*
SWIM	1.00	1.51	*1.51*
MGRID	1.59	1.45	*0.91*
FACEREC	1.04	1.42	*1.36*
GALGEL	1.04	1.39	*1.34*
APSI	1.07	1.23	*1.15*
MESA	1.12	1.17	*1.04*
FMAD	1.32	1.09	*0.82*
ART	1.22	1.07	*0.87*
AMMP	1.18	1.06	*0.88*

Tab. 3.4 – *Gain obtenu par rapport aux performances* Base *des SpecFP2000*

vue des transformations de programmes, le processus génère des séquences structurées de transformations qui sont appliquées à différentes sections de programmes. Pour chaque programme, nous avons modifié 1 à 3 sections de code sur lesquelles plusieurs transformations sont successivement appliquées. En moyenne, nous avons modifié 30 lignes de source par programme. La principale limite aux nombres de modifications des programmes est l'application manuelle des

optimisations.

3.5.1 Optimisation du programme WUPWISE

Le temps d'exécution du programme WUPWISE compilé en *Base* est de 232 secondes.

1ière itération Les anomalies permettent d'identifier un problème de dépendance sur des opérations arithmétiques entières, localisé sur 4 procédures : `lsame`, `zaxpy`, `zgemm` et `zcopy`. Ces 4 procédures appartenant à la librairie mathématique optimisée CXML, nous avons choisi de l'utiliser. Nous avons compilé le programme avec l'option permettant au programme d'utiliser les librairies optimisées : `f90 -O5 -ldxml`. Le temps d'exécution a été réduit à 210 secondes. Les anomalies de dépendances sur des opérations entières ont bien été réduites. En revanche, toutes les autres anomalies ont augmenté. L'optimisation étant globale et pouvant avoir un impact important sur l'ensemble du programme, nous ne pouvons pas effectuer d'itérations d'adaptation. Nous avons donc choisi de commencer une nouvelle itération.

2nde itération Le problème dominant est maintenant un problème d'utilisation des unités fonctionnelles flottantes. Ce problème est principalement localisé dans les procédures `zcopy`, `zgemm`, `zgemm_small_nn` et `zaxpy`. La procédure `zcopy` ne faisant qu'une copie de vecteur sans faire de calcul, il est normal que les unités fonctionnelles flottantes ne soient pas utilisées. Cette procédure est appelée uniquement par la procédure `gammul`. La procédure `gammul` appelle plusieurs fois les procédures `zcopy` et `zaxpy` dans une séquence de structures de contrôles (8 instructions `if... then ...`) dont les exécutions sont exclusives. Un extrait de cette séquence est représenté dans la figure 3.11. Les paramètres d'appel aux procédures étant relativement petits, nous avons

```
IF (MU.EQ.1) THEN
  CALL ZCOPY(12, X, 1, RESULT, 1)
  CALL ZAXPY( 3,  I, X(10), 1, RESULT( 1), 1)
  CALL ZAXPY( 3,  I, X( 7), 1, RESULT( 4), 1)
  CALL ZAXPY( 3, -I, X( 4), 1, RESULT( 7), 1)
  CALL ZAXPY( 3, -I, X( 1), 1, RESULT(10), 1)
```

Fig. 3.11 – *Appels aux procédures* `zcopy` *et* `zaxpy` *dans la procédure* `gammul`

choisi de copier le code des procédures `zcopy` et `zaxpy` dans la procédure `gammul` (nous avons *"inliner"* les procédures). Par ailleurs, nous avons complètement déroulé les boucles générées par la copie, ce qui a pour effet de supprimer totalement la copie réalisée par `zcopy`. La partie de programme résultante est représentée sur la figure 3.12. Après cette optimisation, le temps d'exécution a été réduit à 191 secondes. L'expansion de codes, à elle seule, ne suffit pas à supprimer la copie réalisée par la procédure `zcopy`, elle ne suffit donc pas à réduire l'anomalie d'utilisation des unités fonctionnelles flottantes. En revanche, cette optimisation est indispensable pour réaliser le déroulage complet des boucles qui réduit l'anomalie. Il faut, tout de même, attribuer une partie du gain à l'expansion de code qui réduit considérablement le nombre d'instructions exécutées.

2nde itération, 1ière adaptation Nous avons constaté une augmentation du nombre d'anomalies de dépendance sur les opérations arithmétiques entières (`DEPoO_int`). Nous avons localisé

```
IF (MU.EQ.1) THEN
  RESULT( 1) = X( 1) + (I * X(10))
  RESULT( 2) = X( 2) + (I * X(11))
  RESULT( 3) = X( 3) + (I * X(12))
  RESULT( 4) = X( 4) + (I * X( 7))
  RESULT( 5) = X( 5) + (I * X( 8))
  RESULT( 6) = X( 6) + (I * X( 9))
  RESULT( 7) = X( 7) + ((-I) * X( 4))
  RESULT( 8) = X( 8) + ((-I) * X( 5))
  RESULT( 9) = X( 9) + ((-I) * X( 6))
  RESULT(10) = X(10) + ((-I) * X( 1))
  RESULT(11) = X(11) + ((-I) * X( 2))
  RESULT(12) = X(12) + ((-I) * X( 3))
```

Fig. 3.12 – *Procédure* `gammul` *après expansion de codes et déroulage de boucles*

ces problèmes de dépendance principalement sur les procédures `zgemm_small_cn`, `gammul`, `zgemm` et `zgemm_small_nn`. Les procédures `zgemm_small_cn` et `zgemm_small_nn` sont toutes appelées à partir de l'appel à `zgemm` en fonction des paramètres qui lui sont passés. Les appels à `zgemm` sont faits uniquement dans la procédure `su3mul` qui ne fait rien d'autre que l'appel de procédure à `zgemm` avec des bons paramètres. La procédure `su3mul` est appelée par les procédures `muldeo` et `muldoe`. La procédure `muldeo` est représentée sur la figure 3.13. La procédure `zgemm` étant déjà

```
DO 100 L=1,N4
  LP=MOD(L,N4)+1
    DO 100 K=1,N3
      KP=MOD(K,N3)+1
      DO 100 J=1,N2
        JP=MOD(J,N2)+1
        DO 100 I=(MOD(J+K+L+1,2)+1),N1,2
          IP=MOD(I,N1)+1
          CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
          CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)

          CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
          CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
          CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

          CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
          CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
          CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

          CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
          CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
          CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

          CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
```

Fig. 3.13 – *Procédure* `su3mul`

optimisée, les problèmes de dépendances sont dus au *"sucre syntaxique"* des appels de procédures dont font partie de nombreux calculs d'adresses. Nous avons donc choisi d'effectuer une expansion de code sur les procédures `zgemm`, `su3mul` et `gammul` ainsi que sur les procédures `zaxpy` et `zcopy` dans la procédure `muldeo`. La figure 3.14 montre un résumé de la procédure `muldeo`

après l'expansion de code sur laquelle sont représentés les codes expansés des premiers appels à `gammul` et `su3mul` ainsi que celui de la procédure `zcopy`. Le temps d'exécution a été réduit à 146 secondes.

```
C     CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUXP1)
AUXP1( 1) = X( 1,(IP+1)/2,J,K,L) + (IMA * X(10,(IP+1)/2,J,K,L))
AUXP1( 2) = X( 2,(IP+1)/2,J,K,L) + (IMA * X(11,(IP+1)/2,J,K,L))
AUXP1( 3) = X( 3,(IP+1)/2,J,K,L) + (IMA * X(12,(IP+1)/2,J,K,L))
AUXP1( 4) = X( 4,(IP+1)/2,J,K,L) + (IMA * X( 7,(IP+1)/2,J,K,L))
...
C     CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUXP1,AUXP3)
AUXP3( 1) = U(1,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(1,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(1,3,1,I,J,K,L) * AUXP1( 3)
AUXP3( 2) = U(2,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(2,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(2,3,1,I,J,K,L) * AUXP1( 3)
AUXP3( 3) = U(3,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(3,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(3,3,1,I,J,K,L) * AUXP1( 3)
AUXP3( 4) = U(1,1,1,I,J,K,L) * AUXP1( 4)
     %     + U(1,2,1,I,J,K,L) * AUXP1( 5)
     %     + U(1,3,1,I,J,K,L) * AUXP1( 6)
...
C     ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
RESULT( 1,(I+1)/2,J,K,L) = AUXP3(1)
RESULT( 2,(I+1)/2,J,K,L) = AUXP3(2)
RESULT( 3,(I+1)/2,J,K,L) = AUXP3(3)
RESULT( 4,(I+1)/2,J,K,L) = AUXP3(4)
...
```

Fig. 3.14 – *Procédure* `su3mul` *après expansion de code*

2^{nde} itération, 2^{nde} adaptation Après cette nouvelle expansion de code, nous avons constaté une forte augmentation de l'anomalie latence mémoire `MEM_LAT` et des anomalies spécifiques à l'architecture (`Traps`). Nous nous sommes focalisés sur les anomalies spécifiques à l'architecture qui ont plus que triplé. Ces anomalies sont plus précisément réparties entre les anomalies de conflit d'accès au cache et des anomalies de capacité (débordement de la queue d'instructions de lecture). Nous avons localisé ces anomalies, révélant deux types de problèmes, sur des instructions assembleurs de débordements (*spill code*) dans la procédure `muldeo`. L'expansion de code agressive, que nous avons réalisé, génère un corps de boucle relativement important pour lequel le compilateur n'arrive plus à optimiser l'allocation des registres et l'ordonnancement des instructions. Par conséquent, nous avons optimisé le corps de la boucle en remplaçant les tableaux temporaires `AUXP1, AUXP3, AUXM1` et `AUXM3` par des variables temporaires scalaires. Nous avons également réordonnancé les instructions afin de réduire la durée de vie des registres (nous avons remonté les instructions consommatrices des variables temporaires). Le nouveau corps de boucle est résumé dans la figure 3.15. L'optimisation scalaire et le réordonnancement ont réduit le temps d'exécution à 135 secondes. Ces optimisations ont également réduit l'anomalie de latence mémoire qui avait été augmentée par la précédente adaptation. Comme nous nous y attendions, le réordonnancement, qui a pour objectif de réduire la durée de vie des registres, a augmenté l'anomalie de dépendance flottante (le gain montre que ce compromis est nécessaire). Les autres

```
C     CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUXP1)
AUXP1_1 = X( 1,(IP+1)/2,J,K,L) + (IMA * X(10,(IP+1)/2,J,K,L))
AUXP1_2 = X( 2,(IP+1)/2,J,K,L) + (IMA * X(11,(IP+1)/2,J,K,L))
...
AUXP1_12 = X(12,(IP+1)/2,J,K,L) + ((-IMA) * X( 3,(IP+1)/2,J,K,L))
C     CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUXP1,AUXP3)
AUXP3_1 = U(1,1,1,I,J,K,L) * AUXP1_1
     %     + U(1,2,1,I,J,K,L) * AUXP1_2
     %     + U(1,3,1,I,J,K,L) * AUXP1_3
RESULT( 1,(I+1)/2,J,K,L) = AUXP3_1
AUXP3_4 = U(1,1,1,I,J,K,L) * AUXP1_4
     %     + U(1,2,1,I,J,K,L) * AUXP1_5
     %     + U(1,3,1,I,J,K,L) * AUXP1_6
RESULT( 4,(I+1)/2,J,K,L) = AUXP3_4
AUXP3_7 = U(1,1,1,I,J,K,L) * AUXP1_7
     %     + U(1,2,1,I,J,K,L) * AUXP1_8
     %     + U(1,3,1,I,J,K,L) * AUXP1_9
RESULT( 7,(I+1)/2,J,K,L) = AUXP3_7
AUXP3_10 = U(1,1,1,I,J,K,L) * AUXP1_10
     %     + U(1,2,1,I,J,K,L) * AUXP1_11
     %     + U(1,3,1,I,J,K,L) * AUXP1_12
RESULT(10,(I+1)/2,J,K,L) = AUXP3_10
AUXP3_2 = U(2,1,1,I,J,K,L) * AUXP1_1
     %     + U(2,2,1,I,J,K,L) * AUXP1_2
     %     + U(2,3,1,I,J,K,L) * AUXP1_3
RESULT( 2,(I+1)/2,J,K,L) = AUXP3_2
AUXP3_5 = U(2,1,1,I,J,K,L) * AUXP1_4
     %     + U(2,2,1,I,J,K,L) * AUXP1_5
     %     + U(2,3,1,I,J,K,L) * AUXP1_6
RESULT( 5,(I+1)/2,J,K,L) = AUXP3_5
...
AUXP3_12 = U(3,1,1,I,J,K,L) * AUXP1_10
     %     + U(3,2,1,I,J,K,L) * AUXP1_11
     %     + U(3,3,1,I,J,K,L) * AUXP1_12
RESULT(12,(I+1)/2,J,K,L) = AUXP3_12
...
```

Fig. 3.15 – *Procédure* `su3mul` *après expansion de code*

anomalies n'ayant pas été augmentées, nous avons arrêté les itérations d'adaptations pour cette itération.

Généralisation Pour la première itération, il n'y avait pas de généralisation possible puisque l'utilisation des librairies est déjà une optimisation générale. Il n'y a pas d'opportunité de généralisation de la première optimisation de la seconde itération car la procédure `zcopy` n'est appelée que dans la procédure `gammul`. En revanche, lors de la première adaptation de la seconde itération, nous avons vu que les appels aux procédures `su3mul` et `gammul` étaient répartis équitablement entre les procédures `muldeo` et `muldoe`. Or, ces deux procédures sont très similaires, nous avons donc généralisé la première et la seconde adaptation à la procédure `muldoe`. Le temps d'exécution a été réduit à 80 secondes.

Nous avons arrêté le processus d'optimisation avec un gain final de 2.90.

3.5.2 Association entre anomalies et optimisations

Comme nous l'avons expliqué précédemment, le processus d'optimisation a été construit de manière empirique par une succession de tests et d'erreurs, en particulier pour le premier programme optimisé. Nous avons réalisé plusieurs itérations par programme dans le but de trouver les transformations de programmes appropriées à chaque problème de performance. La table 3.5 montre l'association entre les problèmes de performance correspondant aux feuilles de l'arbre et les optimisations suggérées. La table indique les différents cas pour lesquels une optimisation a réduit un problème de performance (la réduction de l'anomalie correspondant au problème a toujours été au minimum de 5%). Toutes les transformations de programmes n'ont pas le même impact ; la table donne le gain moyen pour chaque transformation ainsi que le nombre de fois où elle a été utilisée.

Optimisations	Problème de cache FP	Problème de cache INT	Problème de bande passante	Dépendances sur opérations flottantes	Dépendances sur des opérations entières	Utilisation déséquilibrée des UF flottantes	Conflits d'accès au cache	TLB de données	Gain moyen	# d'applications
Agencement des données (*Data-layout*)	√						√	√	1.34	3
Ordonnancement d'instructions					√		√		1.24	2
Fission de boucle		√	√	√		√			1.21	4
Blocage pour les registres			√		√	√			1.14	5
Expansion de code (*Inlining*)		√			√	√			1.14	3
Déroulage de boucle					√	√			1.11	2
"Padding"(Data-Layout)	√								1.10	1
Blocage pour le cache	√								1.10	1
Librairies optimisées					√				1.10	1
Décalage d'itération (*Shifting*)	√		√	√	√				1.06	9
Promotion de variables scalaires					√		√	√	1.06	6
Fusion de boucle			√	√		√	√	√	1.05	8
Permutation de boucle		√					√	√	1.02	5
Nombre d'applications	3	2	8	6	7	5	8	5		

Tab. 3.5 – *Associations entre problèmes de performance et transformations de programmes*

Pour plusieurs itérations et pour plusieurs programmes, nous n'avons pas appliqué une optimisation mais deux ou trois transformations de programmes simultanément. Nous avons donc réparti le gain équitablement entre les deux ou trois optimisations réalisées.

Les optimisations sont polyvalentes : hormis le décalage d'itérations qui a été utilisé uniquement pour réduire des problèmes de dépendance (entière ou flottante), les autres optimisations ont été utilisées pour réduire des problèmes de performance variés. Les objectifs d'une même optimisation peuvent être très différents contrairement aux heuristiques des compilateurs statiques guidant les optimisations, qui sont généralement très orientés vers un objectif souvent en relation avec le flot de données et la réutilisation. Par exemple, l'algorithme de Kennedy [Ke00]

pour la fusion de boucles est orienté uniquement vers la réutilisation. Les optimisations réalisées ici montrent que la fusion de boucles peut être utilisée pour réduire des problèmes de conflits ou des problèmes d'utilisation d'unités fonctionnelles.

3.5.3 Séquences d'optimisations

La table 3.6 présente un résumé des séquences d'optimisations réalisées lors de l'optimisation des 12 programmes de la suite SpecFP2000. Pour chaque programme, une ligne de la table correspond à une itération. Chaque optimisation est représentée par une lettre indiquant si elle a été réalisée dans une phase d'optimisation ($\mathcal{O}$), dans une phase d'adaptation ($\mathcal{A}$) ou bien dans une phase de généralisation ($\mathcal{G}$). Ces lettres sont suivies d'informations se trouvant entre des crochets : on trouve le nom de la transformation en haut et le problème de performance que celle-ci a réduit en bas. Les nombres indiquent la progression du gain.

Comme nous l'avons précédemment montré dans la section 3.4.4, les gains que nous avons obtenus avec le processus itératif sont dus à la longueur et à la diversité des séquences d'optimisations. La plupart des optimisations, que nous avons utilisées, existent dans les compilateurs optimisants récents. Cependant, les représentations intermédiaires sous forme d'arbres syntaxiques, les contraintes liées aux analyses de dépendances et les ordres imposés aux phases d'optimisation limitent leur champ d'application. Certains programmes présentent des difficultés supplémentaires pour les compilateurs : les optimisations suivent des directions opposées, par exemple pour le programme APSI, nous avons été amenés à appliquer une fission de boucles puis nous avons fusionné chaque nid de boucle avec deux autres nids de boucles. Pour le programme GALGEL, nous avons également rencontré une opposition avec une optimisation de privatisation pour permettre la permutation de boucles et une optimisation de promotion de scalaires.

3.6 Conclusions

Nous proposons un processus itératif d'optimisation basé sur une analyse dynamique très détaillée. Bien que ce processus soit manuel, il n'en est pas moins systématique. Nous avons montré l'application et l'efficacité de ce processus sur un ensemble de programmes de la suite SpecFP2000.

L'efficacité de ce processus doit être attribuée d'une part à l'analyse dynamique très détaillée qui permet d'identifier précisément les problèmes de performance et d'autre part à la nature itérative du processus qui permet de construire, pour chaque programme, des séquences de transformations.

L'arbre de décision systématise l'analyse dynamique et formalise l'expérience empirique de l'optimisation manuelle d'une manière intelligible. L'expérience empirique peut donc être aisément interprétée, diffusée et réemployée.

Nous avons identifié quatre principales catégories d'anomalies pour l'architecture du processeur Alpha-21264 :

- les anomalies de dépendances (`DEP`),
- les anomalies d'utilisation des unités fonctionnelles (`FU`),
- la latence mémoire (`MEM`),

Codes	Séquences d'optimisations	Gains
WUPWISE	$\mathcal{O}[^{\text{Librairies 1.10}}_{\text{DEPoO int}}]$	1.10
	$\mathcal{O}[^{\text{Expansion/Déroulage 1.21}}_{\text{FU\_fp}}]\blacktriangleright\mathcal{A}[^{\text{Expansion/Déroulage 1.59}}_{\text{FU\_fp}}]\blacktriangleright\mathcal{A}[^{\text{Promo.Scal./Réord. 1.72}}_{\text{Bande passante}}]\blacktriangleright\mathcal{G}[2.90]$	**2.90**
EQUAKE	$\mathcal{O}[^{\text{Réarrange. 1.70}}_{\text{MEM LAT fp}}]\blacktriangleright\mathcal{G}[1.92]\blacktriangleright\mathcal{A}[^{\text{Fusion 1.95}}_{\text{Cache conflict}}]\blacktriangleright\mathcal{A}[^{\text{Promo.scal. 2.50}}_{\text{Cache conflict}}]$	**2.50**
APPLU	$\mathcal{O}[^{\text{Décalage 1.02}}_{\text{DEPoO FP}}]\blacktriangleright\mathcal{A}[^{\text{Promo.scal. 1.04}}_{\text{DEPoO fp}}]\blacktriangleright\mathcal{G}[1.06]$	1.06
	$\mathcal{O}[^{\text{Fusion 1.08}}_{\text{MEM LAT fp}}]\blacktriangleright\mathcal{A}[^{\text{Réord. 1.20}}_{\text{Bande passante}}]\blacktriangleright\mathcal{G}[1.48]$	1.48
	$\mathcal{O}[^{\text{Fission 1.51}}_{\text{DEPoO fp}}]\blacktriangleright\mathcal{A}[^{\text{Déroulage (comp.) 1.65}}_{\text{FU fp}}]\blacktriangleright\mathcal{G}[2.18]$	**2.18**
SWIM	$[^{\text{Padding/Tiling. 1.20}}_{\text{MEM LAT fp}}]\blacktriangleright\mathcal{A}[^{\text{Loop MERGING 1.51}}_{\text{Bandwidth}}]\blacktriangleright\mathcal{A}[1.51]$	**1.51**
MGRID	$\mathcal{O}[^{\text{Fission 0.75}}_{\text{FU\_fp}}]\blacktriangleright\mathcal{A}[^{\text{Fission 1.27}}_{\text{Bande passante}}]\blacktriangleright\mathcal{G}[1.32]$	1.32
	$\mathcal{O}[^{\text{Blocage Reg. 1.38}}_{\text{Bande passante}}]\blacktriangleright\mathcal{G}[1.41]$	**1.41**
FACEREC	$\mathcal{O}[^{\text{n/a. 1.00}}_{\text{FU\_int}}]$	1.00
	$\mathcal{O}[^{\text{Blocage Reg. 1.30}}_{\text{DEPoO fp}}]$	**1.30**
GALGEL	$\mathcal{O}[^{\text{Fusion/Promo.scal. 1.02}}_{\text{Data TLB}}]\blacktriangleright\mathcal{G}[^{\text{Permutation 1.12}}_{\text{Data TLB}}]\blacktriangleright\mathcal{G}[^{\text{Fusion/Promo.scal. 0.97}}_{\text{Data TLB}}]\blacktriangleright\mathcal{A}[^{\text{Décalage 1.12}}_{\text{DEPoO\_int}}]$ $\blacktriangleright\mathcal{A}[^{\text{Fusion/Décalage 1.12}}_{\text{DEPoO fp}}]\blacktriangleright\mathcal{A}[^{\text{Déroulage (Disactivé) 1.16}}_{\text{DEPoO int}}]\blacktriangleright\mathcal{G}[1.24]$	1.24
	$\mathcal{O}[^{\text{Blocage Reg. 1.30}}_{\text{FU fp}}]\blacktriangleright\mathcal{G}[1.32]$	**1.32**
APSI	$\mathcal{O}[^{\text{Réarrange. 1.07}}_{\text{Data TLB}}]\blacktriangleright\mathcal{A}[^{\text{Fission 1.08}}_{\text{Cache conflict}}]\blacktriangleright\mathcal{A}[^{\text{Permutation/Promo.scal. 1.11}}_{\text{Cache conflict}}]\blacktriangleright\mathcal{G}[^{\text{Permutation/Promo.scal. 1.22}}_{\text{Data TLB}}]$	1.22
	$\mathcal{O}[^{\text{Fission 1.22}}_{\text{Bande passante}}]$	1.22
	$\mathcal{O}[^{\text{Fusion/Expansion 1.23}}_{\text{FU fp}}]\blacktriangleright\mathcal{A}[^{\text{Décalage 1.23}}_{\text{DEPoO fp}}]\blacktriangleright\mathcal{A}[^{\text{Décalage 1.23}}_{\text{DEPoO fp}}]\blacktriangleright\mathcal{G}[1.23]$	**1.23**
MESA	$\mathcal{O}[^{\text{Expansion 1.06}}_{\text{DEPoO int}}]$	1.06
	$\mathcal{O}[^{\text{Décalage 1.13}}_{\text{DEPoO\_int}}]\blacktriangleright\mathcal{A}[^{\text{Réarrange. 1.13}}_{\text{Cache conflict}}]$	1.13
	$\mathcal{O}[^{\text{Décalage 1.19}}_{\text{DEPoO int}}]$	**1.19**
FMAD	$\mathcal{O}[^{\text{Décalage 1.00}}_{\text{DEPoO int}}]\blacktriangleright\mathcal{G}[^{\text{Décalage 1.00}}_{\text{DEPoO int}}]\blacktriangleright\mathcal{A}[^{\text{Fission 1.00}}_{\text{Cache conflict}}]\blacktriangleright\mathcal{A}[^{\text{Fusion 1.03}}_{\text{DEPoO fp}}]$	1.03
	$\mathcal{O}[^{\text{Expansion 1.09}}_{\text{Bande passante.}}]\blacktriangleright[1.09]$	**1.09**
ART	$\mathcal{O}[^{\text{Permutation/Promo.scal. 1.02}}_{\text{MEM LAT int}}]$	1.02
	$\mathcal{O}[^{\text{Fission 1.07}}_{\text{MEM LAT int}}]\blacktriangleright[1.07]$	**1.07**
AMMP	$\mathcal{O}[^{\text{Décalage 1.01}}_{\text{DEPoO\_int}}]$	1.01
	$\mathcal{O}[^{\text{Décalage 1.02}}_{\text{DEPoO int}}]$	1.02
	$\mathcal{O}[^{\text{Fusion 1.06}}_{\text{FU\_fp}}]$	**1.06**

TAB. 3.6 – *Résumé des séquences d'optimisations*

– les purges de pipeline dues à des comportements spécifiques à l'architecture des composants (`TRAPS`).

Pour chaque programme, nous avons accumulé ces quatre catégories d'anomalies sur l'ensemble des parties de code traitées. Nous avons ensuite normalisé ces anomalies afin de pouvoir les confronter et d'identifier l'anomalie dominante. La normalisation ne fait perdre aucune information. En contrepartie, l'accumulation des anomalies perd l'information de distribution des

anomalies sur le programme. Bien que nous ayons identifié et résolu de nombreux problèmes de performance, certains problèmes locaux ont pu être masqués par l'accumulation.

L'identification des problèmes de performance peut être affinée par un examen de la distribution des anomalies sur le programme. Cette modification demande la réorganisation du processus d'optimisation. En effet, une recherche sur la distribution allie l'identification d'un problème de performance et sa localisation.

Conclusions et perspectives

Conclusions

La complexité des processeurs augmentant, il est de plus en plus difficile d'intégrer des modèles précis d'architectures dans les compilateurs. En conséquence, l'efficacité des compilateurs décroît. La tendance actuelle vise à enrichir les compilateurs statiques d'informations dynamiques sur le comportement de l'architecture à la manière des techniques d'optimisation basées sur les profiles d'exécution ou des techniques de re-compilation dynamique. Pour le moment, seules quelques informations élémentaires sur le comportement de l'architecture sont utilisées.

Dans le second chapitre, nous avons montré de quelle manière les interactions entre les différents composants d'une architecture rendent complexe le comportement d'un programme. Cependant, nous avons également montré qu'il est possible de capturer cette complexité et d'en déduire les transformations à apporter au programme pour améliorer son comportement.

Dans le troisième chapitre, nous avons étudié une méthode plus systématique pour adresser le problème de la complexité ; afin de la capturer, nous avons proposé un processus itératif d'optimisation manuelle basé sur une analyse dynamique détaillée. Nous avons expérimentalement montré qu'une telle approche améliore la performance des programmes avec un gain moyen de 1,27 par rapport aux optimisations réalisées avec les combinaisons optimales des options du compilateur associées à un pré-processeur optimisant et à des optimisations dirigées par des analyses dynamiques classiques.

Cette approche est potentiellement une stratégie pour guider de futurs environnements d'optimisation itératifs, mais elle présente également des bénéfices immédiats. Premièrement, elle propose un processus systématique d'optimisation manuelle qui peut être utilisé par des ingénieurs. Deuxièmement, l'arbre de décision formalise l'expertise empirique de l'optimisation manuelle qui peut être aisément passée à des ingénieurs ou des chercheurs. Finalement, l'arbre de décision catégorise automatiquement les transformations de programmes suivant les problèmes de performance.

Perspectives

Nous investiguons actuellement plusieurs extensions de ce travail. Premièrement, nous essayons d'extraire des compteurs de performance les informations permettant de guider le processus d'optimisation. Deuxièmement, nous travaillons sur l'implémentation d'un environnement de transformations de programmes basé sur une représentation intermédiaire qui facilite les compositions de longues séquences de transformations. Troisièmement, nous travaillons sur l'automa-

tisation du processus, plus particulièrement, nous cherchons à remplacer l'inspection manuelle de programmes par une analyse statique pour trouver toutes les opportunités d'application d'un ensemble de transformations. Nous laisserons le processus itératif (son analyse dynamique) sélectionner les transformations les plus appropriées. Finalement, nous souhaitons développer les arbres de décision et les associations entre les problèmes de performance et les transformations pour de nouveaux programmes ; nous envisageons notamment de créer des arbres de décision par domaine d'application.

Table des figures

Liste des tableaux

Annexe A

Journaux d'optimisations itératives

Cette annexe a pour vocation de détailler l'optimisation itérative réalisée pour 3 des 11 programmes optimisés. Pour chacun des 11 programmes, nous avons créé un journal d'optimisation qui a la structure suivante : chaque section correspond à une itération du processus, chaque itération possède une phase d'optimisation éventuellement suivie de phases d'adaptation et de généralisation. Pour chaque itération et pour chaque phase, nous donnons le temps d'exécution du programme avant et après l'optimisation ou la séquence d'optimisations. Chaque phase est décomposée en trois étapes : l'analyse qui identifie le problème de performance à partir des anomalies, la localisation et la transformation.

Les anomalies nécessaires à l'identification d'un problème de performance sont présentées par des ensembles de deux ou trois tableaux dont le premier correspond aux anomalies globales. Les tableaux donnent le nombre d'anomalies par cycle (/cycle) à titre indicatif car l'analyse est basée soit sur les valeurs normalisées des anomalies par cycle (norme) soit sur les nombres d'anomalies en valeur absolue (absolue). Les phases d'adaptations présentent des tableaux avec deux lignes, chacune montrant les anomalies en valeur absolue avant et après la transformation précédente.

A.1 Journal d'optimisation du programme WUPWISE

A.1.1 Itération 1 : 232 ↦ 210 (sec.)

Optimisation **: 232 ↦ 210 (sec.)**

[Analyse] Problème de dépendances sur des opérations arithmétiques entières (DEPoO_int).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	2.62	3.03	1.89	6.67	1.36
norme	-0.44	**-0.23**	-0.38	-0.76	-0.93

TAB. A.1 – *Anomalies globales de* WUPWISE

Anomalies	DEP_int	DEPoM_int	DEPoO_int
absolue	657 M	196 M	**461 M**

TAB. A.2 – *Anomalies de dépendances de* WUPWISE

[Localisation] Problème localisé dans les procédures `lsame`, `zcopy`, `zaxpy` et `zgemm`.

Procedures	DEPoO_int
lsame	157 M
zaxpy	137 M
zgemm	100 M
zcopy	35 M

Tab. A.3 – *Localité des anomalies DEPoO_int*

```
f90 -O5 -ldxml
```

Fig. A.1 – *Options de compilation utilisant la librairie CXML*

[Transformation] Utilisation des librairies optimisées.

Par définition les programmes de tests de la suite SpecFP2000 ne doivent pas utiliser les librairies mathématiques optimisées afin d'évaluer uniquement la performance d'une machine et de son compilateur. Ici, le problème de dépendances est localisé sur quatre procédure appartenant aux librairies mathématiques ; nous avons donc choisi d'utiliser celles-ci.

Lors de la validation, nous avons observé une réduction des anomalies de dépendances mais nous avons également observé une augmentation de toutes les autres anomalies bien que le temps d'exécution ait diminué.

A.1.2 Itération 2 : 210 ↦ 80 (sec.)

Optimisation **: 210 ↦ 191 (sec.)**

[Analyse] Problème d'utilisation des unités fonctionnelles flottantes (FU_fp).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.50	2.26	2.37	7.67	3.39
norme	-0.12	-0.55	**0.51**	-0.64	-0.41

Tab. A.4 – *Anomalies globales de* wupwise

Anomalies	FU	FU_fadd	FU_fmul	FU_L0	FU_U0	FU_L1	FU_U1
/cycle	2.37	0.27	0.38	0.25	0.62	0.29	0.55
norme	0.51	0.52	0.20	0.63	-0.08	0.68	-0.11

Tab. A.5 – *Anomalies d'utilisation des unités fonctionnelles de* wupwise

[Localisation] Problème localisé dans les procédures `zcopy`, `zgemm`, `zgemm_small_nn` et `zaxpy`.

```
do i = 1,n
| zy(i) = zx(i)
```

Fig. A.2 – *Procédure zcopy*

La table A.6 montre qu'il faut analyser en priorité les anomalies localisées sur la procédure `zcopy`. La figure A.2 montre que cette procédure ne fait aucun calcul. Cette procédure est prin-

Procedures	Absolut FU_add
zcopy	28.8 M
zgemm	13.0 M
zgemm_small_nn	11.8 M
zaxpy	7.0 M
zgemm_beta	3.4 M
zgemm_smll_cn	2.9 M
gammul	1.3 M

Tab. A.6 – *Localisation des anomalies FU_add.*

```
-----------------------------------------------
                                called/total       parents
index  %time    self descendents  called+self    name            index
                                called/total       children
-----------------------------------------------
                0.00        0.00       2214016156     matmul_ [19]
                0.00        0.00     154214016156     wupwise_ [2]
                0.64        0.00 9728000214016156     muldeo_ [8]
                0.64        0.00 9728000214016156     muldoe_ [7]
               12.78        0.00 194560000214016156   gammul_ [6]
[10]     6.7   14.06        0.00 214016156         zcopy_ [10]
-----------------------------------------------
```

Fig. A.3 – *Appels à la procédure* `zcopy`

```
IF (MODE.EQ.0) THEN
  IF (MU.EQ.1) THEN
    CALL ZCOPY(12, X, 1, RESULT, 1)
    CALL ZAXPY( 3,  I, X(10), 1, RESULT( 1), 1)
    CALL ZAXPY( 3,  I, X( 7), 1, RESULT( 4), 1)
    CALL ZAXPY( 3, -I, X( 4), 1, RESULT( 7), 1)
    CALL ZAXPY( 3, -I, X( 1), 1, RESULT(10), 1)
  ELSE IF (MU.EQ.2) THEN
    CALL ZCOPY(12, X, 1, RESULT, 1)
    CALL ZAXPY( 3,  ONE, X(10), 1, RESULT( 1), 1)
    CALL ZAXPY( 3, -ONE, X( 7), 1, RESULT( 4), 1)
    CALL ZAXPY( 3, -ONE, X( 4), 1, RESULT( 7), 1)
    CALL ZAXPY( 3,  ONE, X( 1), 1, RESULT(10), 1)
  ELSE IF (MU.EQ.3) THEN
    CALL ZCOPY(12, X, 1, RESULT, 1)
    CALL ZAXPY( 3,  I, X( 7), 1, RESULT( 1), 1)
    CALL ZAXPY( 3, -I, X(10), 1, RESULT( 4), 1)
    CALL ZAXPY( 3, -I, X( 1), 1, RESULT( 7), 1)
    CALL ZAXPY( 3,  I, X( 4), 1, RESULT(10), 1)
  ELSE IF (MU.EQ.4) THEN
    CALL ZCOPY( 6, X,    1, RESULT,    1)
    CALL ZCOPY( 6, ZERO, 0, RESULT(7), 1)
    CALL ZSCAL( 6, TWO, RESULT, 1)
  END IF
ELSE IF (MODE.EQ.1) THEN
...
```

Fig. A.4 – *Procédure* `gammul`

cipalement appelée par la procédure `gammul` (table A.3). La procédure `gammul` est représentée sur la figure A.4.

[Transformation] Expansion de code (*inlining*) et déroulage complet (*Full unrolling*).

La procédure `gammul` appelle les procédures `zcopy` et `zaxpy`. Or, nous avions également relevé des problèmes d'utilisation des unités fonctionnelles dans cette procédure. Nous avons donc choisi d'optimiser avec de l'expansion de code et du déroulage complet appliqué aux procédures `zcopy` et `zaxpy`. Le code optimisé de la procédure `gammul` est représenté sur la figure A.5.

```
IF (MODE.EQ.0) THEN
  IF (MU.EQ.1) THEN
    RESULT( 1) = X( 1) + (I * X(10))
    RESULT( 2) = X( 2) + (I * X(11))
    RESULT( 3) = X( 3) + (I * X(12))
    RESULT( 4) = X( 4) + (I * X( 7))
    RESULT( 5) = X( 5) + (I * X( 8))
    RESULT( 6) = X( 6) + (I * X( 9))
    RESULT( 7) = X( 7) + ((-I) * X( 4))
    RESULT( 8) = X( 8) + ((-I) * X( 5))
    RESULT( 9) = X( 9) + ((-I) * X( 6))
    RESULT(10) = X(10) + ((-I) * X( 1))
    RESULT(11) = X(11) + ((-I) * X( 2))
    RESULT(12) = X(12) + ((-I) * X( 3))
...
```

FIG. A.5 – *Procédure `gammul` optimisée*

Adaptation **1 : 191 ↦ 146 (sec.)**

[Analyse] Augmentation des anomalies de dépendances sur des opérations arithmétiques entières (DEPoO_int).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
absolue avant	927 M	598 M	626 M	918 M	898 K
absolue après	737 M	**666 M**	574 M	942 M	874 K

TAB. A.7 – *Anomalies globales de* WUPWISE

Anomalies	DEP_int	DEPoM_int	DEPoO_int
absolue avant	598 M	172 M	425 M
absolue après	666 M	122 M	**543 M**

TAB. A.8 – *Anomalies de dépendances de* WUPWISE

[Localisation] Problème localisé dans les procédures `zgemm_small_cn`, `gammul`, `zgemm` et `zgemm_small_nn`.

Etant donné que les procédures `zgemm_small_cn`, `zgemm` et `zgemm_small_nn` sont des procédures de la librairies optimisée, nous ne pouvons pas modifier le code. Ces procédures sont

Procedures	Absolut DEPoO_int
zgemm_small_cn	193 M
gammul	102 M
zgemm	99 M
zgemm_small_nn	84 M
zaxpy	32 M
muldoe	25 M
zgemm_beta	2 M

Tab. A.9 – *Localisation des anomalies FU_fadd.*

```
SUBROUTINE SU3MUL(U,TRANSU,X,RESULT)
 ...
 CALL ZGEMM(TRANSU, 'NO TRANSPOSE',3,4,3,
.           ONE,U,3,X,3,ZERO,RESULT,3)
 RETURN
END
```

Fig. A.6 – *Procédure su3mul*

appelées par la procédure `su3mul` (figure A.6). Cette dernière appelle uniquement la procédure `zgemm` avec les paramètres qui lui sont passés.

Nous avons décidé d'observer les appels à la procédure `su3mul` (figure A.7), nous observons la procédure `muldeo` (figure A.8).

```
-----------------------------------------------
                1.52        0.00 58368000/116736000     muldeo_ [8]
                1.52        0.00 58368000/116736000     muldoe_ [7]
[15]     1.5    3.04        0.00 116736000          su3mul_ [15]
-----------------------------------------------
```

Fig. A.7 – *Appels à la procédure su3mul*

```
DO 100 L=1,N4
  LP=MOD(L,N4)+1
  DO 100 K=1,N3
    KP=MOD(K,N3)+1
    DO 100 J=1,N2
      JP=MOD(J,N2)+1
      DO 100 I=(MOD(J+K+L+1,2)+1),N1,2
        IP=MOD(I,N1)+1
        CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
        CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)

        CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
        CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
        CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

        CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
        CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
        CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

        CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
        CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
        CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)

        CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
 100  CONTINUE
```

Fig. A.8 – *Procédure* `muldeo`

[Transformation] Expansion de code (*inlining*) et déroulage complet (*Full unrolling*).

Nous pouvons remarquer que, comme la procédure `zgemm`, la procédure `gammul` a des problèmes de dépendances sur des opérations arithmétiques entières (voir table A.9).

```
...
DO 100 I=(MOD(J+K+L+1,2)+1),N1,2
  IP=MOD(I,N1)+1
  IM=I-1
  IF(IM.EQ.0) IM=N1
  C     CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUXP1)
  AUXP1( 1) = X( 1,(IP+1)/2,J,K,L) + (IMA * X(10,(IP+1)/2,J,K,L))
  AUXP1( 2) = X( 2,(IP+1)/2,J,K,L) + (IMA * X(11,(IP+1)/2,J,K,L))
  AUXP1( 3) = X( 3,(IP+1)/2,J,K,L) + (IMA * X(12,(IP+1)/2,J,K,L))
  AUXP1( 4) = X( 4,(IP+1)/2,J,K,L) + (IMA * X( 7,(IP+1)/2,J,K,L))
  ...
  AUXP1(12) = X(12,(IP+1)/2,J,K,L) + ((-IMA) * X( 3,(IP+1)/2,J,K,L))
  C     CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUXP1,AUXP3)
  AUXP3( 1) = U(1,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(1,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(1,3,1,I,J,K,L) * AUXP1( 3)
  AUXP3( 2) = U(2,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(2,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(2,3,1,I,J,K,L) * AUXP1( 3)
  AUXP3( 3) = U(3,1,1,I,J,K,L) * AUXP1( 1)
     %     + U(3,2,1,I,J,K,L) * AUXP1( 2)
     %     + U(3,3,1,I,J,K,L) * AUXP1( 3)
  AUXP3( 4) = U(1,1,1,I,J,K,L) * AUXP1( 4)
     %     + U(1,2,1,I,J,K,L) * AUXP1( 5)
     %     + U(1,3,1,I,J,K,L) * AUXP1( 6)
  ...
  AUXP3(12) = U(3,1,1,I,J,K,L) * AUXP1(10)
     %     + U(3,2,1,I,J,K,L) * AUXP1(11)
     %     + U(3,3,1,I,J,K,L) * AUXP1(12)
  C     CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
  ...
```

Fig. A.9 – *Procédure `muldeo` optimisée*

Nous avons choisi d'utiliser l'expansion de code et le déroulage de boucle. Les procédures `zgemm` et `su3mul` sont successivement expansées ainsi que la procédure `gammul` précédemment optimisée. Ensuite, le code expansé de la procédure `zgemm` est complètement déroulé.

Adaptation **2 : 146 ↦ 135 (sec.)**

[Analyse] Augmentation des anomalies de conflit d'accès au cache (Cache conflict) et de débordement de la queue d'instructions de lecture mémoire (LSQ/MAF full).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
absolue avant	737 M	666 M	574 M	942 M	874 K
absolue-après	899 M	869 M	747 M	2079 M	**2781 K**

Tab. A.10 – *Anomalies globales de* wupwise

Anomalies	TRAPS	MEM_TRAPS	DTBMISS1	DTBMISS2	ITBMISS	MISSPRED
Absolue	2781 K	**2733 K**	47 K	0 K	0 K	0 K

Tab. A.11 – *Anomalies spécifiques à l'architecture de* wupwise

Anomalies	MEM_TRAPS	CFT	ORDER	MAF_CFT	MAF_FULL	SQ_FULL	LQ_FULL
Absolue	2733 K	**1058 K**	199 K	195 K	0 K	323 K	**942 K**

Tab. A.12 – *Anomalies mémoires spécifiques de* wupwise

[Localisation] Problème localisé dans le corps de boucle de la procédure `muldeo`.

Procedures	Absolut CFT	Absolut LQ_FULL
zcopy	487 K	12 K
muldeo	226 K	312 K
wupwise	220 K	22 K
gammul	118 K	0 K
zgemm_small_nn	6 K	0 K

Tab. A.13 – *Localisation des anomalies de Conflit de cache et de LQ_FULL.*

[Transformation] Promotion de scalaires (*Scalar promotion*) et réordonnancement d'instructions (*Instruction scheduling*).

```
...
DO 100 I=(MOD(J+K+L+1,2)+1),N1,2
  IP=MOD(I,N1)+1
  IM=I-1
  IF(IM.EQ.0) IM=N1
  C      CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUXP1)
  AUXP1_1 = X( 1,(IP+1)/2,J,K,L) + (IMA * X(10,(IP+1)/2,J,K,L))
  AUXP1_2 = X( 2,(IP+1)/2,J,K,L) + (IMA * X(11,(IP+1)/2,J,K,L))
  AUXP1_3 = X( 3,(IP+1)/2,J,K,L) + (IMA * X(12,(IP+1)/2,J,K,L))
  AUXP1_4 = X( 4,(IP+1)/2,J,K,L) + (IMA * X( 7,(IP+1)/2,J,K,L))
  ...
  AUXP1_12 = X(12,(IP+1)/2,J,K,L) + ((-IMA) * X( 3,(IP+1)/2,J,K,L))
  C      CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUXP1,AUXP3)
  AUXP3_1 = U(1,1,1,I,J,K,L) * AUXP1_1
       %     + U(1,2,1,I,J,K,L) * AUXP1_2
       %     + U(1,3,1,I,J,K,L) * AUXP1_3
  RESULT( 1,(I+1)/2,J,K,L) = AUXP3_1
  AUXP3_4 = U(1,1,1,I,J,K,L) * AUXP1_4
       %     + U(1,2,1,I,J,K,L) * AUXP1_5
       %     + U(1,3,1,I,J,K,L) * AUXP1_6
  RESULT( 4,(I+1)/2,J,K,L) = AUXP3_4
  AUXP3_7 = U(1,1,1,I,J,K,L) * AUXP1_7
       %     + U(1,2,1,I,J,K,L) * AUXP1_8
       %     + U(1,3,1,I,J,K,L) * AUXP1_9
  RESULT( 7,(I+1)/2,J,K,L) = AUXP3_7
  AUXP3_10 = U(1,1,1,I,J,K,L) * AUXP1_10
       %     + U(1,2,1,I,J,K,L) * AUXP1_11
       %     + U(1,3,1,I,J,K,L) * AUXP1_12
  RESULT(10,(I+1)/2,J,K,L) = AUXP3_10
  AUXP3_2 = U(2,1,1,I,J,K,L) * AUXP1_1
       %     + U(2,2,1,I,J,K,L) * AUXP1_2
       %     + U(2,3,1,I,J,K,L) * AUXP1_3
  RESULT( 2,(I+1)/2,J,K,L) = AUXP3_2
  AUXP3_5 = U(2,1,1,I,J,K,L) * AUXP1_4
       %     + U(2,2,1,I,J,K,L) * AUXP1_5
       %     + U(2,3,1,I,J,K,L) * AUXP1_6
  RESULT( 5,(I+1)/2,J,K,L) = AUXP3_5
  ...
  AUXP3_12 = U(3,1,1,I,J,K,L) * AUXP1_10
       %     + U(3,2,1,I,J,K,L) * AUXP1_11
       %     + U(3,3,1,I,J,K,L) * AUXP1_12
  RESULT(12,(I+1)/2,J,K,L) = AUXP3_12
  ...
```

Fig. A.10 – *Procédure* `muldeo` *optimisée*

Généralisation : **135 ↦ 80 (sec.)**

[Analyse] Opportunité de généralisation des adaptations 1 et 2.

```
-----------------------------------------------
                1.52        0.00 58368000/116736000     muldeo_ [8]
                1.52        0.00 58368000/116736000     muldoe_ [7]
[15]     1.5    3.04        0.00 116736000         su3mul_ [15]
-----------------------------------------------
```

FIG. A.11 – *Appels à la procédure* `su3mul`

[Localisation] Problème et structure de programme similaires dans la procédure `muldoe`.

[Transformation] Transformation des adaptations 1 et 2 : Expansion de code (*inlining*), déroulage complet (*Full unrolling*), promotion de scalaires (*Scalar promotion*) et réordonnancement des instructions (*Instruction scheduling*).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
Absolue-avant	1063 M	619 M	722 M	1566 M	2460 K
Absolue-après	1111 M	509 M	715 M	1745 M	2327 K

TAB. A.14 – *Anomalies globales de* WUPWISE

Nous avons arrêté l'optimisation après avoir réduit le temps d'exécution de WUPWISE de 232 à 80 secondes, soit après avoir obtenu un gain de 2,90.

A.2 Journal d'optimisation du programme EQUAKE

A.2.1 Itération 1 : 290 ↦ 116 (sec.)

Optimisation : **232 ↦ 171 (sec.)**

[Analyse] Problème de latence mémoire flottante (MEM_LAT_fp).

Les anomalies mettent en évidence un problème de dépendances sur des opérations mémoires flottantes (tables A.15 et A.16). La normalisation de l'anomalie de latence mémoire élevée (table A.15) permet d'identifier un problème de latence et non un problème de bande passante (voir l'arbre de décisions figure 3.2).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	9.14	2.92	2.55	21.96	3.35
norme	**1.97**	-0.27	0.84	*1.12*	-0.42

TAB. A.15 – *Anomalies globales de* EQUAKE

Anomalies	DEP_fp	DEPoM_fp	DEPoO_fp
absolue	6378 M	**3617 M**	2761 M

TAB. A.16 – *Anomalies de dépendances de* EQUAKE

Anomalies	MEM_LAT	MEM_LAT_fp	MEM_LAT_int
absolue	4151 M	**3551 M**	571 M

TAB. A.17 – *Anomalies de latence mémoire de* EQUAKE

[Localisation] Problème localisé dans la procédure `smvp`.

Le code assembleur de la figure A.13 montre que les accès au tableau `A` utilisent des registres d'index différents (`t3`,`t6` et `t10`) car le tableau `A` est un tableau de pointeurs de tableaux de pointeurs. Le problème est similaire sur le tableau `w`.

Procedure	MEM_LAT_fp
smvp	2690 M
main	781 M
phi1	32 M
phi0	26 M
phi2	22 M

TAB. A.18 – *Localité des anomalies MEM_LAT_fp*

Lignes sources	MEM_LAT_fp
1212	680 M
1208	554 M
1207	430 M
1213	316 M
1214	302 M
1209	245 M
1199	112 M
1200	111 M
1201	109 M
1219	5 M

Tab. A.19 – *Localité des anomalies MEM_LAT_fp dans la procédure smvp*

```
void smvp(int nodes, double ***A, int *Acol,
                int *Aindex, double **v, double **w) {
  int i;
  int Anext, Alast, col;
  double sum0, sum1, sum2;

  for (i = 0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

    Anext++;
    while (Anext < Alast) {
      col = Acol[Anext];

1207: sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
1208: sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
1209: sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];


1212: w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
1213: w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
1214: w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
1215:       Anext++;
    }
    w[i][0] += sum0;
    w[i][1] += sum1;
    w[i][2] += sum2;
  }
}
```

Fig. A.12 – *Procédure smvp*

```
[quake.c: 1207] 0x120011040:        8f440000        ldt         $f26, 0(t3)
[quake.c: 1207] 0x120011044:        8d440008        ldt         $f10, 8(t3)
[quake.c: 1208] 0x120011048:        8f780000        ldt         $f27, 0(t10)
[quake.c: 1209] 0x12001104c:        8fc70000        ldt         $f30, 0(t6)
[quake.c: 1208] 0x120011050:        8e380008        ldt         $f17, 8(t10)
[quake.c: 1212] 0x120011054:        8c7b0000        ldt         $f3, 0(t12)
[quake.c: 1207] 0x120011058:        8de40010        ldt         $f15, 16(t3)
[quake.c: 1208] 0x12001105c:        8e580010        ldt         $f18, 16(t10)
[quake.c: 1207] 0x120011060:        43340644        s8addq        t11, a4, t3
[quake.c: 1209] 0x120011064:        8c870008        ldt         $f4, 8(t6)
[quake.c: 1207] 0x120011068:        a4840000        ldq         t3, 0(t3)
[quake.c: 1207] 0x12001106c:        8cc40008        ldt         $f6, 8(t3)
[quake.c: 1207] 0x120011070:        8ce40010        ldt         $f7, 16(t3)
[quake.c: 1212] 0x120011074:        5b40145c        mult         $f26,$f0,$f28
[quake.c: 1212] 0x120011078:        5b6b145d        mult         $f27,$f11,$f29
[quake.c: 1212] 0x12001107c:        2ffe0000        ldq_u         zero, 0(sp)
[quake.c: 1212] 0x120011080:        5bd0144d        mult         $f30,$f16,$f13
[quake.c: 1213] 0x120011084:        59401453        mult        $f10,$f0,$f19
[quake.c: 1214] 0x120011088:        5a4b1442        mult        $f18,$f11,$f2
[quake.c: 1214] 0x12001108c:        2ffe0000        ldq_u         zero, 0(sp)
[quake.c: 1212] 0x120011090:        5b9d141c        addt        $f28,$f29,$f28
[quake.c: 1213] 0x120011094:        5a2b145d        mult        $f17,$f11,$f29
[quake.c: 1207] 0x120011098:        5946144a        mult        $f10,$f6,$f10
[quake.c: 1207] 0x12001109c:        2ffe0000        ldq_u         zero, 0(sp)
```

Fig. A.13 – *Code assembleur de la procédure* **smvp**

```
/* Stiffness matrix K[ARCHmatrixlen][3][3] */

K = (double ***) malloc(ARCHmatrixlen * sizeof(double **));
if (K == (double ***) NULL) {
  fprintf(stderr, "malloc failed for K");
  fflush(stderr);
  exit(0);
}
for (i = 0; i < ARCHmatrixlen; i++) {
  K[i] = (double **) malloc(3 * sizeof(double *));
  if (K[i] == (double **) NULL) {
    fprintf(stderr, "malloc failed for K[%d]",i);
    fflush(stderr);
    exit(0);
  }
  for (j = 0; j < 3; j++) {
    K[i][j] = (double *) malloc(3 * sizeof(double));
    if (K[i][j] == (double *) NULL) {
      fprintf(stderr, "malloc failed for K[%d][%d]",i,j);
      fflush(stderr);
      exit(0);
    }
  }
}
```

Fig. A.14 – *Allocation mémoire du tableau K*

```
/* Displacement array disp[3][ARCHnodes][3] */

disp = (double ***) malloc(3 * sizeof(double **));
if (disp == (double ***) NULL) {
  fprintf(stderr, "malloc failed for disp");
  fflush(stderr);
  exit(0);
}
for (i = 0; i < 3; i++) {
  disp[i] = (double **) malloc(ARCHnodes * sizeof(double *));
  if (disp[i] == (double **) NULL) {
    fprintf(stderr, "malloc failed for disp[%d]",i);
    fflush(stderr);
    exit(0);
  }
  for (j = 0; j < ARCHnodes; j++) {
    disp[i][j] = (double *) malloc(3 * sizeof(double));
    if (disp[i][j] == (double *) NULL) {
      fprintf(stderr, "malloc failed for disp[%d][%d]",i,j);
      fflush(stderr);
      exit(0);
    }
  }
}
```

Fig. A.15 – *Allocation mémoire du tableau* `disp`

[Transformation] Réarrangement des structures de données (*Data-layout optimization*).

Nous avons modifié les allocations mémoires des tableaux `K` et `disp` passés en paramètre à la procédure `smvp`. Ainsi, les lignes de cache contiennent des données utiles (et non des pointeurs).

```
K = (double *) malloc(ARCHmatrixlen * sizeof(double)*9 );
if ( K == (double *) NULL ) {
  fprintf(stderr, "malloc failed for K");
  fflush(stderr);
  exit(0);
}
```

Fig. A.16 – *Nouvelle allocation mémoire du tableau `K`*

```
disp = (double **) malloc(3 * sizeof(double *));
if (disp == (double **) NULL) {
  fprintf(stderr, "malloc failed for disp");
  fflush(stderr);
  exit(0);
}
for (i = 0; i < 3; i++) {
  disp[i] = (double *) malloc(ARCHnodes * 3 * sizeof(double));
  if (disp[i] == (double *) NULL) {
    fprintf(stderr, "malloc failed for disp[%d]",i);
    fflush(stderr);
    exit(0);
  }
}
```

Fig. A.17 – *Nouvelle allocation mémoire du tableau `disp`*

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
absolue	6378 M	2038 M	1781 M	4174 M	2334 K
absolue	2465 M	1376 M	942 M	1985 M	1086 K

Tab. A.20 – *Anomalies globales de* equake

```
void smvp(int nodes, double *A, int *Acol,
                int *Aindex, double *v, double *w) {
  int i;
  int Anext, Alast, col;
  double sum0, sum1, sum2;
  int ANC;

  for (i = 0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];
    ANC = Anext*9;
    sum0 = A[ANC+0*3+0]*v[i*3+0] + A[ANC+0*3+1]*v[i*3+1] + A[ANC+0*3+2]*v[i*3+2];
    sum1 = A[ANC+1*3+0]*v[i*3+0] + A[ANC+1*3+1]*v[i*3+1] + A[ANC+1*3+2]*v[i*3+2];
    sum2 = A[ANC+2*3+0]*v[i*3+0] + A[ANC+2*3+1]*v[i*3+1] + A[ANC+2*3+2]*v[i*3+2];

    Anext++;
    while (Anext < Alast) {
      ANC = Anext*9;
      col = Acol[Anext];

      sum0 += A[ANC+0*3+0]*v[col*3+0] + A[ANC+0*3+1]*v[col*3+1] + A[ANC+0*3+2]*v[col*3+2];
      sum1 += A[ANC+1*3+0]*v[col*3+0] + A[ANC+1*3+1]*v[col*3+1] + A[ANC+1*3+2]*v[col*3+2];
      sum2 += A[ANC+2*3+0]*v[col*3+0] + A[ANC+2*3+1]*v[col*3+1] + A[ANC+2*3+2]*v[col*3+2];


      w[col*3+0] += A[ANC+0*3+0]*v[i*3+0] + A[ANC+1*3+0]*v[i*3+1] + A[ANC+2*3+0]*v[i*3+2];
      w[col*3+1] += A[ANC+0*3+1]*v[i*3+0] + A[ANC+1*3+1]*v[i*3+1] + A[ANC+2*3+1]*v[i*3+2];
      w[col*3+2] += A[ANC+0*3+2]*v[i*3+0] + A[ANC+1*3+2]*v[i*3+1] + A[ANC+2*3+2]*v[i*3+2];
      Anext++;
    }
    w[i*3+0] += sum0;
    w[i*3+1] += sum1;
    w[i*3+2] += sum2;
  }
}
```

Fig. A.18 – *Procédure* `smvp` *après réarrangemant des données*

Généralisation : **171 ↦ 151 (sec.)**

[Analyse] Opportunité de généralisation de l'itération 1.

[Localisation] Problème similaire dans la procédure `main`.

Procedure	MEM_LAT_fp
smvp	866 M
main	654 M
phi1	34 M
phi0	30 M
phi2	25 M

Tab. A.21 – *Localité des anomalies MEM_LAT_fp*

Lignes sources	MEM_LAT_fp
477	246 M
499	142 M
506	54 M
479	45 M
498	42 M

Tab. A.22 – *Localité des anomalies MEM_LAT_fp dans la procédure smvp*

```
    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        disp[disptplus][i*3+j] += 2.0 * M[i][j] * disp[dispt][i*3+j] -
477:          (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i*3+j] -
            Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
                               C23[i][j] * phi1(time) / 2.0 +
                               V23[i][j] * phi0(time) / 2.0);
```

Fig. A.19 – *Procédure main*

[Transformation] Transformation de l'itération 1 : Réarrangement des structures de données (*Data-layout optimization*).

Nous avons modifié l'allocation mémoire des tableaux `C, M, M23, C23` et `V23`. Les figures A.21 montre le code modifié de l'allocation du tableau `M`.

```
/* Mass matrix */

  M = (double **) malloc(ARCHnodes * sizeof(double *));
  if (M == (double **) NULL) {
    fprintf(stderr, "malloc failed for M");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    M[i] = (double *) malloc(3 * sizeof(double));
    if (M[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for M[%d]",i);
      fflush(stderr);
      exit(0);
    }
  }
```

Fig. A.20 – *Allocation mémoire du tableau* `M`

```
  M = (double *) malloc(ARCHnodes * 3 * sizeof(double));
  if (M == (double *) NULL) {
    fprintf(stderr, "malloc failed for M");
    fflush(stderr);
    exit(0);
  }
```

Fig. A.21 – *Nouvelle allocation mémoire du tableau* `M`

```
    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
          (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
            Exc.dt * Exc.dt * (M23[i*3+j] * phi2(time) / 2.0 +
                               C23[i*3+j] * phi1(time) / 2.0 +
                               V23[i*3+j] * phi0(time) / 2.0);
```

Fig. A.22 – *Procédure* `main` *après réarrangement des données*

Adaptation **1 : 151 ↦ 149 (sec.)**

[Analyse] Augmentation de l'anomalie de conflit d'accés au cache (*Cache conflict*).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
absolue	2465 M	1376 M	942 M	1985 M	1086 K
absolue	2222 M	1235 M	932 M	1769 M	**1536 K**

TAB. A.23 – *Anomalies globales de* EQUAKE

Anomalies	TRAPS	MEM_TRAPS	DTBMISS1	DTBMISS2	ITBMISS	MISSPRED
absolue	1086 K	1013 K	72 K	0 K	0 K	0 K
absolue	1536 K	**1472 K**	64 K	0 K	0K	0 K

TAB. A.24 – *Anomalies spécifiques à l'architecture de* EQUAKE

Anomalies	MEM_TRAPS	CFT	ORDER	MAF_CFT	MAF_FULL	SQ_FULL	LQ_FULL
absolue	1013 K	119 K	594 K	12 K	0 K	65 K	212 K
absolue	1472 K	88 K	**1089 K**	7 K	0 K	54 K	228 K

TAB. A.25 – *Anomalies mémoires spécifiques de* EQUAKE

[Localisation] Problème localisé sur la procédure `main` dans le nid de boucle précédement optimisé.

Les opérations de lecture et d'écriture d'un élément du tableau `disp` doivent être exécutées dans l'ordre par le processus. Le corps du nid de boucle `i,j` étant petit (ligne 519, figure A.23), ces opérations se retrouvent simultanément en cours d'exécution dans le processeur qui génére donc des purges partielles de pipeline (ORDER Traps).

Procedure	MEM_TRAPS	ORDER
main	1340 K	1020 K
phi2	35 K	34 K
phi1	22 K	20 K
phi0	14 K	14 K
smvp	59 K	0 K

TAB. A.26 – *Localité des anomalies ORDER (traps)*

Lignes sources	ORDER	
source line	MEM_TRAPS	ORDER_TRAPS
519	415 K	415 K
512	218 K	209 K
511	129 K	128 K
506	143 K	110 K
514	59 K	59 K

TAB. A.27 – *Localité des anomalies ORDER dans la procédure `main`*

```
    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        disp[disptplus][i*3+j] *= - Exc.dt * Exc.dt;

    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
          (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
            Exc.dt * Exc.dt * (M23[i*3+j] * phi2(time) / 2.0 +
                               C23[i*3+j] * phi1(time) / 2.0 +
                               V23[i*3+j] * phi0(time) / 2.0);

    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
519:    disp[disptplus][i*3+j] = disp[disptplus][i*3+j] /
                                 (M[i*3+j] + Exc.dt / 2.0 * C[i*3+j]);

    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        vel[i*3+j] = 0.5 / Exc.dt * (disp[disptplus][i*3+j] -
                                     disp[disptminus][i*3+j]);
```

Fig. A.23 – *Procédure* `main`

[Transformation] Fusion de boucles pour augmenter la distance entre les lectures et écritures sur une même adresse mémoire.

```
    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        {
        disp[disptplus][i*3+j] *= - Exc.dt * Exc.dt;

        disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
          (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
            Exc.dt * Exc.dt * (M23[i*3+j] * phi2(time) / 2.0 +
                               C23[i*3+j] * phi1(time) / 2.0 +
                               V23[i*3+j] * phi0(time) / 2.0);

        disp[disptplus][i*3+j] = disp[disptplus][i*3+j] /
                                 (M[i*3+j] + Exc.dt / 2.0 * C[i*3+j]);

        vel[i*3+j] = 0.5 / Exc.dt * (disp[disptplus][i*3+j] -
                                     disp[disptminus][i*3+j]);
        }
```

Fig. A.24 – *Procédure* `main` *après fusion*

Adaptation **2 : 149 ↦ 116 (sec.)**

[Analyse] Augmentation de l'anomalie de conflit d'accés au cache (*Cache conflict*).

[Localisation] Problème localisé sur la procédure `main`, `phi0`, `phi1` et `phi2` dans le nid de boucle précédement optimisé.

Anomalies	MEM_TRAPS	CFT	ORDER	MAF_CFT	MAF_FULL	SQ_FULL	LQ_FULL
absolue	1472 K	88 K	1089 K	7 K	0 K	54 K	228 K
absolue	1382 K	107 K	1159 K	12 K	0 K	87 K	12 K

Tab. A.28 – *Anomalies mémoires spécifiques de* equake

Des opérations de lecture sur une même adresse s'exécutent simultanément et génèrent des (ORDER Traps). Ces opérations sont dues aux appels de procédures `phi0`, `phi1` et `phi2`.

Procedure	ORDER
main	997 K
phi2	127 K
phi1	18 K
phi0	16 K
smvp	0 K

Tab. A.29 – *Localité des anomalies ORDER*

source line	ORDER_TRAPS
505	953 K
479	20 K
502	16 K
475	3 K

Tab. A.30 – *Localité des anomalies ORDER dans la procédure main*

```
    for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
        {
        disp[disptplus][i*3+j] *= - Exc.dt * Exc.dt;

        disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
          (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
            Exc.dt * Exc.dt * (M23[i*3+j] * phi2(time) / 2.0 +
                               C23[i*3+j] * phi1(time) / 2.0 +
505:                               V23[i*3+j] * phi0(time) / 2.0);

        disp[disptplus][i*3+j] = disp[disptplus][i*3+j] /
                                  (M[i*3+j] + Exc.dt / 2.0 * C[i*3+j]);

        vel[i*3+j] = 0.5 / Exc.dt * (disp[disptplus][i*3+j] -
                                     disp[disptminus][i*3+j]);
        }
```

Fig. A.25 – *Procédure main*

[Transformation] Factorisation d'invariants de boucles (*Loop invariant hosting*).

La variable `time` étant constante sur l'espace d'itération du nid de boucles, nous avons factorisé les appels de procédures et placé leurs résultats dans des variables temporaires.

```
[quake.c: 491] 0x12000d910:        a71d83d0        ldq         t10, -31792(gp)
[quake.c: 491] 0x12000d914:        8c290000        ldt         $f1, 0(s0)
[quake.c: 503] 0x12000d918:        5ca50410        cpys         $f5,$f5,$f16
...
[quake.c: 504] 0x12000d9c0:        58601443        mult         $f3,$f0,$f3
[quake.c: 504] 0x12000d9c4:        9c7e0460        stt         $f3, 1120(sp)
[quake.c: 505] 0x12000d9c8:        8c6dfff8        ldt         $f3, -8(s4)
[quake.c: 505] 0x12000d9cc:        d34003a0        bsr         ra, 0x12000e850(zero)
[quake.c: 503] 0x12000d9d0:        8c3e0460        ldt         $f1, 1120(sp)
[quake.c: 505] 0x12000d9d4:        58601440        mult         $f3,$f0,$f0
[quake.c: 501] 0x12000d9d8:        8d5e0458        ldt         $f10, 1112(sp)
[quake.c: 513] 0x12000d9dc:        a43d83d0        ldq         t0, -31792(gp)
[quake.c: 520] 0x12000d9e0:        a47d83d0        ldq         t2, -31792(gp)
[quake.c: 487] 0x12000d9e4:        a0be0440        ldl         t4, 1088(sp)
[quake.c: 487] 0x12000d9e8:        a4fe0448        ldq         t6, 1096(sp)
[quake.c: 520] 0x12000d9ec:        a61e0450        ldq         a0, 1104(sp)
[quake.c: 503] 0x12000d9f0:        59211401        addt         $f9,$f1,$f1
...
[quake.c: 520] 0x12000da58:        8d830000        ldt         $f12, 0(t2)
[quake.c: 521] 0x12000da5c:        8daafff8        ldt         $f13, -8(s1)
[quake.c: 520] 0x12000da60:        584c146c        divt         $f2,$f12,$f12
[quake.c: 520] 0x12000da64:        580d1420        subt         $f0,$f13,$f0
[quake.c: 520] 0x12000da68:        59801440        mult         $f12,$f0,$f0
[quake.c: 520] 0x12000da6c:        9c10fff8        stt         $f0, -8(a0)
[quake.c: 487] 0x12000da70:        f4bfffa7        bne         t4, 0x12000d910
```

Fig. A.26 – *Code assembleur de la procédure* `main`

```
TMP_phi2 = phi2(time) / 2.0;
TMP_phi1 = phi1(time) / 2.0;
TMP_phi0 = phi0(time) / 2.0;


for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    {
    disp[disptplus][i*3+j] *= - Exc.dt * Exc.dt;

    disp[disptplus][i*3+j] += 2.0 * M[i*3+j] * disp[dispt][i*3+j] -
      (M[i*3+j] - Exc.dt / 2.0 * C[i*3+j]) * disp[disptminus][i*3+j] -
        Exc.dt * Exc.dt * (M23[i*3+j] * TMP_phi2 +
                           C23[i*3+j] * TMP_phi1 +
                           V23[i*3+j] * TMP_phi0);

    disp[disptplus][i*3+j] = disp[disptplus][i*3+j] /
                             (M[i*3+j] + Exc.dt / 2.0 * C[i*3+j]);

    vel[i*3+j] = 0.5 / Exc.dt * (disp[disptplus][i*3+j] -
                                 disp[disptminus][i*3+j]);
    }
```

Fig. A.27 – *Procédure* `main` *après factorisation d'invariants de boucles*

L'anomalie de latence mémoire a augmenté dans la procédure `smvp` mais elle a diminué dans la procédure `main`. L'augmentation est due à un effet de bord. Nous avons arrêté l'optimisation

Anomalies	MEM_TRAPS	CFT	ORDER	MAF_CFT	MAF_FULL	SQ_FULL	LQ_FULL
absolue	1382 K	107 K	1159 K	12 K	0 K	87 K	12 K
absolue	490 K	87 K	58 K	9 K	0 K	138 K	191 K

Tab. A.31 – *Anomalies mémoires spécifiques de* equake

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
absolue	2302 M	1270 M	901 M	1752 M	1437 K
absolue	2240 M	1007 M	871 M	1966 M	576 K

Tab. A.32 – *Anomalies globales de* equake

après avoir réduit le temps d'exécution de 290 à 116 secondes, soit après un gain de 2,50.

A.3 Journal d'optimisation du programme APPLU

A.3.1 Itération 1 : 312 ↦ 294 (sec.)

Optimisation : **312 ↦ 305 (sec.)**

[Analyse] Problème de dépendances sur des opérations arithmétiques flottantes (DEPoO_fp).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	5.50	2.49	2.31	14.82	4.22
norme	**0.69**	-0.48	0.40	0.38	-0.58

TAB. A.33 – *Anomalies globales de* APPLU

Anomalies	DEP_fp	DEPoM_fp	DEPoO_fp
/cycle	5.50	2.23	3.27

TAB. A.34 – *Anomalies de dépendances de* APPLU

[Localisation] Problème localisé dans les procédures `blts`.

Procedures	DEPoO_fp
blts	0.47
buts	0.23
rhs	0.12
jacld	0.06

TAB. A.35 – *Localité des anomalies DEPoO_fp*

```
      do k = 2, nz-1
      | do j = 2, ny-1
      | | do i = 2, nx-1
      | | | do m = 1, 5
      | | | | do l = 1, 5
      | | | | | v( m, i, j, k ) =  v( m, i, j, k )
563:  | | | | |    - omega * (  ldz( m, l, i, j, k ) * v( l, i, j, k-1 )
564:  | | | | |                + ldy( m, l, i, j, k ) * v( l, i, j-1, k )
565:  | | | | |                + ldx( m, l, i, j, k ) * v( l, i-1, j, k ) )
```

FIG. A.28 – *Procédure* `blts`

Le problème est localisé sur les lignes 563, 564 et 565. En effet, l'opération de multiplication par `omega` doit attendre les exécutions de 2 additions qui, elles-mêmes, doivent attendre les exécutions de 3 multiplications.

[Transformation] Fission de boucle (*Loop fission*) et décalage d'itérations (*Shifting*) (*Enable Transfo. : Privatization*).

Nous avons privatisé les calculs intermédiaires pour pouvoir scinder le nid de boucle (`m,l`).

```
do k = 2, nz-1
  do j = 2, ny-1
    do m = 1, 5
      do l = 1, 5
        tmpldx(m,l) = ldx( m, l, 2, j, k )
             * v( l, 2-1, j, k )
        tmpldy(m,l) = ldy( m, l, 2, j, k )
             * v( l, 2, j-1, k )
        tmpldz(m,l) = ldz( m, l, 2, j, k )
             * v( l, 2, j, k-1 )
    do i = 2, nx-1
      do m = 1, 5
        do l = 1, 5
          v( m, i, j, k ) =  v( m, i, j, k )
               - omega * (  tmpldz(m,l)
               + tmpldy(m,l)
               + tmpldx(m,l) )
...
      do m = 1, 5
        do l = 1,5
          tmpldx(m,l) = ldx( m, l, i+1, j, k )
               * v( l, i-1+1, j, k )
          tmpldy(m,l) = ldy( m, l, i+1, j, k )
               * v( l, i+1, j-1, k )
          tmpldz(m,l) = ldz( m, l, i+1, j, k )
               * v( l, i+1, j, k-1 )
```

Fig. A.29 – *Procédure* `blts` *après fusion*

Adaptation **1 : 305 ↦ 300 (sec.)**

[Analyse] Augmentation des anomalies de dépendances sur des opérations arithmétiques entières (DEPoO_int).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cylce	5.50	2.49	2.31	14.82	4.22
/cycle	3.78	**3.65**	2.17	13.27	4.00

Tab. A.36 – *Anomalies globales de* applu

Anomalies	DEP_int	DEPoM_int	DEPoO_int
/cycle	2.49	0.10	2.39
/cycle	3.65	**1.02**	**2.62**

Tab. A.37 – *Anomalies de dépendances de* applu

[Localisation] Problème localisé dans les nids de boucle de la procédure optimisée `blts`.

Le problème est localisé sur les lignes 646, 648 et 650. Les opérations d'écritures flottantes sont dans la queue de lancement des instructions entières. Avec l'optimisation précédente, nous avons remplacé les dépendances entre des opérations arithmétiques par des dépendances entre des opérations arithmétiques et des opérations d'écritures.

```
      do k = 2, nz-1
      | do j = 2, ny-1
      | | do i = 2, nx-1
      | | | do m = 1, 5
      | | | | do l = 1, 5
      | | | | | v( m, i, j, k ) =  v( m, i, j, k )
      | | | | | | - omega * (  tmpldz(m,l)
      | | | | | |             + tmpldy(m,l)
      | | | | | |             + tmpldx(m,l) )
      | | | ...
      | | | | do m = 1, 5
      | | | | | do l = 1,5
646:  | | | | | | tmpldx(m,l) = ldx( m, l, i+1, j, k )
      | | | | | | |     * v( l, i-1+1, j, k )
648:  | | | | | | tmpldy(m,l) = ldy( m, l, i+1, j, k )
      | | | | | | |     * v( l, i+1, j-1, k )
650:  | | | | | | tmpldz(m,l) = ldz( m, l, i+1, j, k )
      | | | | | | |     * v( l, i+1, j, k-1 )
```

Fig. A.30 – *Procédure* `blts`

[Transformation] Modification de la fission de boucle (*Loop fission Modification*).

Nous avons modifié la fission de boucle pour rechercher un compromis entre des deux problèmes (DEPoO_fp et DEPoO_int).

```
      do k = 2, nz-1
      | do j = 2, ny-1
      | | do i = 2, nx-1
      | | | do m = 1, 5
      | | | | do l = 1, 5
      | | | | | v( m, i, j, k ) =  v( m, i, j, k )
      | | | | | | - omega *  tmpldxyz(m,l)
      ...
      | | | do m = 1, 5
      | | | | do l = 1,5
      | | | | | tmpldx = ldx( m, l, i+1, j, k )
      | | | | | |     * v( l, i-1+1, j, k )
      | | | | | tmpldy = ldy( m, l, i+1, j, k )
      | | | | | |     * v( l, i+1, j-1, k )
      | | | | | tmpldz = ldz( m, l, i+1, j, k )
      | | | | | |     * v( l, i+1, j, k-1 )
      | | | | | tmpldxyz(m,l) = tmpldx + tmpldy + tmpldz
```

Fig. A.31 – *Procédure* `blts` *après modification de la fission de boucles*

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.78	3.65	2.17	13.27	4.00
/cycle	4.03	3.31	2.15	12.83	3.64

Tab. A.38 – *Anomalies globales de* applu

Généralisation : **300 ↦ 294 (sec.)**

[Analyse] Opportunité de généralisation de l'optimisation et l'adaptation.

[Localisation] Problème et structure de programme similaire dans la procédure `buts`.

Procedures	DEPoO_fp
blts	0.47
buts	0.23
rhs	0.12
jacld	0.06

Tab. A.39 – *Localité des anomalies DEPoO_fp*

```
do k = nz-1, 2, -1
  do j = ny-1, 2, -1
    do i = nx-1, 2, -1
      do m = 1, 5
        tv( m ) = 0.0d+00
        do l = 1, 5
          tv( m ) = tv( m ) +
               omega * (  udz( m, l, i, j, k ) * v( l, i, j, k+1 )
                        + udy( m, l, i, j, k ) * v( l, i, j+1, k )
                        + udx( m, l, i, j, k ) * v( l, i+1, j, k ) )
c***diagonal block inversion
 ...
c***back substitution
 ...
```

Fig. A.32 – *Procédure buts*

[Transformation] Transformations identiques : Fission de boucle modifiée (*Loop fission Modified*) et décalage d'itérations (*Shifting*).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	4.03	3.31	2.15	12.83	3.64
/cycle	4.03	3.33	2.14	12.32	3.44

Tab. A.40 – *Anomalies globales de* applu

```
do k = nz-1, 2, -1
 do j = ny-1, 2, -1
  do m = 1, 5
   do l = 1, 5
    tmpldx = udx( m, l, nx-1, j, k )
        * v( l, nx-1+1, j, k )
    tmpldy = udy( m, l, nx-1, j, k )
        * v( l, nx-1, j+1, k )
    tmpldz = udz( m, l, nx-1, j, k )
        * v( l, nx-1, j, k+1 )
    tldxyz(m,l) = tmpldx + tmpldy + tmpldz
  do i = nx-1, 2, -1
   do m = 1, 5
    tv( m ) = 0.0d+00
    do l = 1, 5
     tv( m ) = tv( m ) +
          omega * ( tldxyz(m,l) )
c***diagonal block inversion
 ...
c***back substitution
 ...
   do m = 1, 5
    do l = 1,5
     tmpldx = udx( m, l, i-1, j, k )
         * v( l, i+1-1, j, k )
     tmpldy = udy( m, l, i-1, j, k )
         * v( l, i-1, j+1, k )
     tmpldz = udz( m, l, i-1, j, k )
         * v( l, i-1, j, k+1 )
      tldxyz(m,l) = tmpldx
          + tmpldy
          + tmpldz
```

FIG. A.33 – *Procédure* `buts` *après généralisation de la fission de boucles*

A.3.2 Itération 2 : 294 ↦ 211 (sec.)

Optimisation : **294 ↦ 289 (sec.)**

[Analyse] Problème de latence mémoire flottante (MEM_LAT_fp).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	4.03	3.33	2.11	12.33	3.44
norme	**0.08**	-0.12	0.07	*0.02*	-0.63

Tab. A.41 – *Anomalies globales de* applu

Anomalies	DEP_fp	DEPoM_fp	DEPoO_fp
/cycle	4.03	**2.10**	1.93

Tab. A.42 – *Anomalies de dépendances de* applu

[Localisation] Problème localisé dans les procédures `buts` et `blts`.

Le problème de latence mémoire est localisé sur les opérations de lectures des tableaux `udx`, `udy` et `udz`.

Procedure	DEP_fp	DEPoM_fp
buts	0.38	**0.41**
blts	0.33	0.35
rhs	0.10	0.09
jacu	0.08	0.06
Procedures	MEM_LAT	MEM_LAT_fp
buts	0.40	**0.41**
blts	0.32	0.31
rhs	0.10	0.10
jacld	0.08	0.08

Tab. A.43 – *Localité des anomalies DEPoO_fp*

[Annulation] Annulation des optimisations de la première itération afin de pouvoir appliquer une nouvelle optimisation.

```
do k = nz-1, 2, -1
 do j = ny-1, 2, -1
  do m = 1, 5
   do l = 1, 5
    tmpldx = udx( m, l, nx-1, j, k )
        * v( l, nx-1+1, j, k )
    tmpldy = udy( m, l, nx-1, j, k )
        * v( l, nx-1, j+1, k )
    tmpldz = udz( m, l, nx-1, j, k )
        * v( l, nx-1, j, k+1 )
    tldxyz(m,l) = tmpldx + tmpldy + tmpldz
  do i = nx-1, 2, -1
   do m = 1, 5
    tv( m ) = 0.0d+00
    do l = 1, 5
     tv( m ) = tv( m ) +
          omega * ( tldxyz(m,l) )
c***diagonal block inversion
 ...
c***back substitution
 ...
   do m = 1, 5
    do l = 1,5
(801)  tmpldx = udx( m, l, i-1, j, k )
         * v( l, i+1-1, j, k )
(803)  tmpldy = udy( m, l, i-1, j, k )
         * v( l, i-1, j+1, k )
(805)  tmpldz = udz( m, l, i-1, j, k )
         * v( l, i-1, j, k+1 )
       tldxyz(m,l) = tmpldx
           + tmpldy
           + tmpldz
```

FIG. A.34 – *Procédure* `buts`

[Transformation] Fusion de procédure et de boucles (*Procedure and Loop fusion*).

Les tableaux n'ont aucun potentiel de réutilisation dans le nid de boucles. Nous avons cherché la partie de programme qui produit ces tableaux pour essayer de rapprocher les producteurs des consommateurs. Nous avons donc fusionné les procédures `jacu` et `buts` ainsi que les nids de boucles (`k,j,i`).

```
C ### Début de la procédure Jacu ###
do k = nz-1, 2, -1
  do j = ny-1, 2, -1
    do i = nx-1, 2, -1
      ...
      a(1,1,i,j,k) = ...
      ...
      c(5,5,i,j,k) = ...
C ### Début de la procédure Buts ###
      do m = 1, 5
        tv( m ) = 0.0d+00
        do l = 1, 5
          tv( m ) = tv( m ) +
            omega * (  c( m, l, i, j, k ) * rsd( l, i, j, k+1 )
                     + b( m, l, i, j, k ) * rsd( l, i, j+1, k )
                     + a( m, l, i, j, k ) * rsd( l, i+1, j, k ) )
        end do
      end do
c***diagonal block inversion
      do m = 1, 5
        do l = 1, 5
        tmat( m, l ) = d( m, l, i, j, k )
      do ip = 1, 4
        ...
c***back substitution
...
      do m = 1, 5
        rsd( m, i, j, k ) = rsd( m, i, j, k ) - tv( m )
      end do
C ### END of Buts Core procedure ###
```

FIG. A.35 – *Procédure* `jacu_buts` *après la fusion des procédures* `jacu` *et* `buts`

Adaptation 1 : 289 ↦ 260 (sec.)

[Analyse] Augmentation des anomalies de dépendances sur des opérations mémoires entières (DEPoM_int) (code de débordement (*spill code*).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	4.03	3.31	2.15	12.83	3.64
/cycle	3.66	**3.55**	2.04	11.82	4.99

TAB. A.44 – *Anomalies globales de* APPLU

[Localisation] Problème localisé dans les nids de boucle de la procédure optimisée `jacu-buts`.

Problème localisé sur la ligne 2425 : les opérations d'écritures dépendent des opérations de

Anomalies	DEP_int	DEPoM_int	DEPoO_int
jacu	149 M	14 M	96 M
buts	842 M	305 M	537 M
jacu_buts	1077 M	435 M	641 M
augmentation	8%	**36%**	1%

Tab. A.45 – *Augmentation des anomalies de dépendances après la fusion*

lectures. Au niveau du code assembleur, on aperçoit un problème de code de débordement (*spill code*).

```
C ### Début de la procédure Jacu ###
do k = nz-1, 2, -1
  do j = ny-1, 2, -1
    do i = nx-1, 2, -1
      ...
      a(1,1,i,j,k) = ...
      ...
      c(5,5,i,j,k) = ...
C ### Début de la procédure Buts ###
      do m = 1, 5
        tv( m ) = 0.0d+00
        do l = 1, 5
          tv( m ) = tv( m ) +
            omega * (  c( m, l, i, j, k ) * rsd( l, i, j, k+1 )
                     + b( m, l, i, j, k ) * rsd( l, i, j+1, k )
                     + a( m, l, i, j, k ) * rsd( l, i+1, j, k ) )
        end do
      end do
c***diagonal block inversion
      do m = 1, 5
        do l = 1, 5
(2425)    tmat( m, l ) = d( m, l, i, j, k )
      do ip = 1, 4
        ...
c***back substitution
...
      do m = 1, 5
        rsd( m, i, j, k ) = rsd( m, i, j, k ) - tv( m )
      end do
C ### END of Buts Core procedure ###
```

Fig. A.36 – *Procédure* `jacu_buts`

[Transformation] Promotion de scalaires et réordonnancement d'instructions (*Enable transformations : full unrolling, computation reordering*).

Pour réduire le code de débordement, nous essayons de réduire la durée de vie des variables. Pour cela, nous avons complètement déroulé le nid de boucles qui calcule les éléments du tableau `tv`, ce qui a permis de réduire les durées de vie des tableaux `a,b` et `c`. Ensuite, nous avons remplacé ces trois tableaux par des variables scalaires.

L'augmentation des anomalies d'utilisation des unités fonctionnelles est liée à une diminution de l'utilisation des unités fonctionnelles entières.

```
c***form the first block sub-diagonal
c
                a11 = - dt * tx1 * dx1
                a12 =   dt * tx2
                a13 =   0.0d+00
                a14 =   0.0d+00
                a15 =   0.0d+00
                tv( 1 ) = tv( 1 ) + 1.0d+00 * (
                    a11 * rsd(1, i+1, j, k ) +
                    a12 * rsd(2, i+1, j, k ) +
                    a13 * rsd(3, i+1, j, k ) +
                    a14 * rsd(4, i+1, j, k ) +
                    a15 * rsd(5, i+1, j, k ) )
c
                a21 =  dt * tx2 * ...
                a22 =  dt * tx2 * ...
                a23 =  dt * tx2 * ...
                a24 =  dt * tx2 * ...
                a25 =  dt * tx2 * c2
                tv( 2 ) = tv( 2 ) + 1.0d+00 * (
                    a21 * rsd(1, i+1, j, k ) +
                    a22 * rsd(2, i+1, j, k ) +
                    a23 * rsd(3, i+1, j, k ) +
                    a24 * rsd(4, i+1, j, k ) +
                    a25 * rsd(5, i+1, j, k ) )
...
                b11 = - dt * ty1 * dy1
                b12 =   0.0d+00
                b13 =  dt * ty2
                b14 =   0.0d+00
                b15 =   0.0d+00
                tv( 1 ) = tv( 1 ) + 1.0d+00 * (
                    b11 * rsd(1, i, j+1, k ) +
                    b12 * rsd(2, i, j+1, k ) +
                    b13 * rsd(3, i, j+1, k ) +
                    b14 * rsd(4, i, j+1, k ) +
                    b15 * rsd(5, i, j+1, k ) )
...
```

Fig. A.37 – *Procédure* `jacu-buts` *après la promotion de scalaire et le réordonnancement*

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.66	3.55	2.04	11.82	4.99
/cycle	3.75	3.46	2.12	10.96	3.03

Tab. A.46 – *Anomalies globales de* applu

Généralisation : **260 ↦ 211 (sec.)**

[Analyse] Opportunité de généralisation de l'optimisation et de l'adaptation.

[Localisation] Problème et structure de programmes similaire dans la procédure `blts`.

Procedures	MEM	MEM_int
jacu-buts	0.35	0.36
blts	0.36	**0.62**
rhs	0.15	0.02
jacld	0.09	0.00

Tab. A.47 – *Localité des anomalies MEM_LAT_int*

```
    do m = 1, 5
    |  do l = 1,5
565 |  |  tmpldx(m,l) = ldx( m, l, 2, j, k )
    |  |      * v( l, 2-1, j, k )
567 |  |  tmpldy(m,l) = ldy( m, l, 2, j, k )
    |  |      * v( l, 2, j-1, k )
569 |  |  tmpldz(m,l) = ldz( m, l, 2, j, k )
    |  |      * v( l, 2, j, k-1 )
```

Fig. A.38 – *Procédure* `blts`

[Transformation] Transformations identiques : Fusion de procédure et de boucles (*Procedure and Loop fusion*). Plus, promotion de scalaires et réordonnancement d'instructions (*Enable transformations : full unrolling, computation reordering*).

```
c***form the block daigonal
               a11 = - dt * tz1 * dz1
               a12 =   0.0d+00
               a13 =   0.0d+00
               a14 = - dt * tz2
               a15 =   0.0d+00
               tv(1) = tv(1) + (
                   a11 * rsd(1, i, j, k-1 ) +
                   a12 * rsd(2, i, j, k-1 ) +
                   a13 * rsd(3, i, j, k-1 ) +
                   a14 * rsd(4, i, j, k-1 ) +
                   a15 * rsd(5, i, j, k-1 ) )

               a21 = - dt * tz2 * ...
               a22 = - dt * tz2 * ( u(4,i,j,k-1) * tmp1 )
               a23 = 0.0d+00
               a24 = - dt * tz2 * ( u(2,i,j,k-1) * tmp1 )
               a25 = 0.0d+00
               tv(2) = tv(2) + (
                   a21 * rsd(1, i, j, k-1 ) +
                   a22 * rsd(2, i, j, k-1 ) +
                   a23 * rsd(3, i, j, k-1 ) +
                   a24 * rsd(4, i, j, k-1 ) +
                   a25 * rsd(5, i, j, k-1 ) )
               ...
C BEGIN BLTS PROCEDURE CORE
               rsd(1,i,j,k) = rsd(1,i,j,k) - ( omega * tv(1) )
               rsd(2,i,j,k) = rsd(2,i,j,k) - ( omega * tv(2) )
               rsd(3,i,j,k) = rsd(3,i,j,k) - ( omega * tv(3) )
               rsd(4,i,j,k) = rsd(4,i,j,k) - ( omega * tv(4) )
               rsd(5,i,j,k) = rsd(5,i,j,k) - ( omega * tv(5) )

c***forward elimination
               do m = 1, 5
                  do l = 1, 5
                     tmat( m, l ) = d( m, l, i, j, k )
                  end do
               end do
```

Fig. A.39 – *Procédure* `jacld-blts` *après la fusion des procédures* `jacld` *et* `blts`

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.75	3.46	2.12	10.96	3.03
/cycle	3.64	3.11	2.05	9.17	2.68

Tab. A.48 – *Anomalies globales de* applu

A.3.3 Itération 3 : 211 ↦ 143 (sec.)

Optimisation : **211 ↦ 207 (sec.)**

[Analyse] Problème de dépendances sur des opérations arithmétiques flottantes (DEPoO_fp).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.64	3.11	2.05	9.17	2.68
norme	-0.07	-0.21	-0.10	-0.42	-0.70

Tab. A.49 – *Anomalies globales de* applu

Anomalies	DEP_fp	DEPoM_fp	DEPoO_fp
/cycle	3.64	1.71	1.93

Tab. A.50 – *Anomalies de dépendances de* applu

[Localisation] Problème localisé dans les procédures `jacld-blts` et `jacu-buts`.

Procedures	DEPoO_fp
jacld	0.45
jacu	0.40
rhs	0.11
l2norm	0.02

Tab. A.51 – *Localité des anomalies DEPoO_fp*

[Transformation] Fission de boucle avec privatisation.

Adaptation **1 : 207 ↦ 189 (sec.)**

[Analyse] Augmentation des anomalies d'utilisation des unités fonctionnelles flottantes (FU_fp).

Anomalies	DEP_fp	DEP_int	FU	MEM	TRAPS
/cycle	3.64	3.11	2.05	9.17	2.68
/cycle	3.06	2.63	2.13	8.72	2.36

Tab. A.52 – *Anomalies globales de* applu

Anomalies	FU	FU_fadd	FU_fmul	FU_L0	FU_U0	FU_L1	FU_U1
/cycle	2.04	0.14	0.20	0.08	0.79	0.13	0.70
/cycle	2.12	**0.17**	**0.26**	0.09	0.78	0.14	0.69

Tab. A.53 – *Anomalies d'utilisation des unités fonctionnelles de* applu

[Localisation] Problème localisé dans les nids de boucle de la procédure optimisée `jacld-blts`.

Ce problème a été généré par l'optimisation précédente qui a créé des nids de boucles ayant un unique type d'opération.

```
c***forward elimination
               do m = 1, 5
                  do l = 1, 5
                     tmat( m, l ) = d( m, l, i, j, k )
                  end do
               end do
               do ip = 1, 4
                  tmp1 = 1.0d+00 / tmat( ip, ip )
                  do m = ip+1, 5
                     tmp = tmp1 * tmat( m, ip )
                     do l = ip+1, 5
                        tmat( m, l ) =  tmat( m, l )
2233:                                 - tmp * tmat( ip, l )
                     end do
                     rsd( m, i, j, k ) = rsd( m, i, j, k )
2238:                              - rsd( ip, i, j, k ) * tmp
                  end do
               end do
c***back substitution
               do m = 5, 1, -1
                  do l = m+1, 5
                     rsd( m, i, j, k ) = rsd( m, i, j, k )
2251:                               - tmat( m, l ) * rsd( l, i, j, k )
                  end do
                  rsd( m, i, j, k ) = rsd( m, i, j, k )
                                    / tmat( m, m )
            end do
```

Fig. A.40 – *Procédure* `jacld-blts`

604R-Anomalies	FU_fadd	FU_fmul	605-Anomalies	FU_fadd	FU_fmul
ssor	7.8M	1.0M	ssor	6.4M	1.2M
rhs	3.9M	25.9M	rhs	3.8M	26.9M
l2norm	0.1M	0.9M	l2norm	0.1M	0.9M
jacu	37.5M	50.8M	jacu	37.4M	50.7M
jacld	**28.5M**	**34.3M**	jacld	**40.6M**	**55.4M**

Tab. A.54 – *Augmentation des anomalies d'utilisation des unités fonctionnelles flottantes*

[Transformation] Déroulage complet.

Pour permettre aux heuristiques du compilateur de réordonnancer les instructions au mieux, nous avons complètement déroulé les différents nids de boucles.

Anomalies	FU	FU_fadd	FU_fmul	FU_L0	FU_U0	FU_L1	FU_U1
/cycle	2.13	0.17	0.26	0.09	0.78	0.14	0.69
/cycle	2.29	**0.12**	**0.18**	0.11	0.88	0.18	0.83

Tab. A.55 – *Anomalies d'utilisation des unités fonctionnelles de* applu

Généralisation : **189 ↦ 143 (sec.)**

[Analyse] Opportunité de généralisation de l'optimisation et l'adaptation.

```
c***forward elimination
               do m = 1, 5
                  do l = 1, 5
                     tmat( m, l ) = d( m, l, i, j, k )
                  end do
               end do
               do ip = 1, 4
                  tmp1 = 1.0d+00 / tmat( ip, ip )
                  do m = ip+1, 5
                     tmp = tmp1 * tmat( m, ip )
                     do l = ip+1, 5
                        tmat( m, l ) =  tmat( m, l )
2233:                                 - tmp * tmat( ip, l )
                     end do
                     rsd( m, i, j, k ) = rsd( m, i, j, k )
2238:                                  - rsd( ip, i, j, k ) * tmp
                  end do
               end do
c***back substitution
               do m = 5, 1, -1
                  do l = m+1, 5
2249:                      ttrsd(m,l) = tmat( m, l ) * rsd( l, i, j, k )
                  end do
               end do
               do m = 5, 1, -1
                  do l = m+1, 5
                     rsd( m, i, j, k ) = rsd( m, i, j, k )
2258:                                  - ttrsd(m,l)
                  end do
                  rsd( m, i, j, k ) = rsd( m, i, j, k )
2264:                                 / tmat( m, m )
             end do
           end do
```

FIG. A.41 – *Procédure* `jacld-blts` *après la privatisation et la fission de boucle*

[Localisation] Problème et structure de programmes similaire dans la procédure `jacu-buts`.

Procédures	FU_fadd	FU_fmul
jacu-buts	**0.56**	**0.51**
rhs	0.06	0.27
jacld-blts	0.28	0.19
ssor	0.10	0.01
l2norm	0.00	0.01

TAB. A.56 – *Localité des anomalies FU_fp*

[Transformation] Transformations identiques : Déroulage complet.

Le programme de la procédure `jacu-buts` est similaire à celui de la procédure `jacld-blts` représenté sur la figure A.42.

Nous avons arrêté l'optimisation d'APPLU après avoir réduit le temps d'exécution de 312 à 143 secondes, soit un gain de 2,18.

```
C  FULL UNROLLING
C ### ip = 1 ###
                  tmp1 = 1.0d+00 / tmat( 1, 1 )
C ### m = 2 ###
                     tmp = tmp1 * tmat( 2, 1 )
C ### l = 2 ###
                        tmat( 2, 2 ) =  tmat( 2, 2 )
                            - tmp * tmat( 1, 2 )
C ### l = 3 ###
                        tmat( 2, 3 ) =  tmat( 2, 3 )
                            - tmp * tmat( 1, 3 )
C ### l = 4 ###
                        tmat( 2, 4 ) =  tmat( 2, 4 )
                            - tmp * tmat( 1, 4 )
C ### l = 5 ###
                        tmat( 2, 5 ) =  tmat( 2, 5 )
                            - tmp * tmat( 1, 5 )
                     rsd( 2, i, j, k ) = rsd( 2, i, j, k )
                      - rsd( 1, i, j, k ) * tmp
C ### m = 3 ###
                     tmp = tmp1 * tmat( 3, 1 )
                     ...
C ### m = 5 ###
                     ...
C ### ip = 2 ###
                     ...
c***back substitution
C FULL UNROLLING
C ### m = 5 ###
                  rsd( 5, i, j, k ) = rsd( 5, i, j, k )
                                   / tmat( 5, 5 )
C ### m = 4 ###
...
```

FIG. A.42 – *Procédure* `jacld-blts` *après déroulage complet*

Bibliographie

[AbToAn00] Jaume Abella and Sid Ahmed Ali Touati and Alan Anderson and Carlos Ciuraneta and Josep M. Codina and Min Dai and Christine Eisenbeis and Grigori Fursin and Anotio González and Josep Llosa and Michael O'Boyle and Andry Randrianatovina and Jesus Sánchez and Olivier Temam and Xavier Vera and Gragory Watts. **The MHAOTEU Toolset**. In *Agent-based simulation, planning and control of the 16thIMACS World Congress*, 2000.

[AhSeUl86] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman. **Compilers — Principles, Techniques, and Tools**. Addison-Wesley, Reading, Massachusetts, USA, 1986.

[AlCoGr04] L. Almagor and Keith D. Cooper and Alexander Grosul and Timothy J. Harvey and Steven W. Reeves and Devika Subramanian and Linda Torczon and Todd Waterman. **Finding effective compilation sequences**. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 231–239. ACM Press, 2004.

[AlKe01] Randy Allen and Ken Kennedy. **Optimizing Compilers for Modern Architectures**. Morgan Kaufmann Publishers, 2001.

[BaCoGi03] C. Bastoul and A. Cohen and S. Girbal and S. Sharma and O. Temam. **Putting Polyhedral Loop Transformation to Work**. In *10th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2003.

[BaDuBa00] Vasanth Bala and Evelyn Duesterwald and Sanjeev Banerjia. **Dynamo : a transparent dynamic optimization system**. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12. ACM Press, 2000.

[BaGrSh94] David F. Bacon and Susan L. Graham and Oliver J. Sharp. **Compiler transformations for high-performance computing**. *ACM Computing Surveys*, 26(4) :345–420, 1994.

[BrCoKe89] P. Briggs and K. D. Cooper and K. Kennedy and L. Torczon. **Coloring heuristics for register allocation**. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 275–284. ACM Press, 1989.

[Ca96] Steve Carr. **Combining Optimization for Cache and Instruction-Level Parallelism**. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 238. IEEE Computer Society, 1996.

[CaFeHu95] Larry Carter and Jeanne Ferrante and Susan Flynn Hummel. **Hierarchical tiling for improved superscalar performance**. In *Proceedings of the 9th International Symposium on Parallel Processing*, pages 239–245. IEEE Computer Society, 1995. See also : Hierarchical tiling : a methodology for high performance. Technical report UCSD, 1996.

[CaGu97] Steve Carr and Yiping Guan. **Unroll-and-jam using uniformly generated sets**. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357. IEEE Computer Society, 1997.

[CaKe94] Steve Carr and Ken Kennedy. **Scalar replacement in the presence of conditional control flow**. *Softw. Pract. Exper.*, 24(1) :51–77, 1994.

[ChPaHaLe01] Siddhartha Chatterjee and Erin Parker and Philip J. Hanlon and Alvin R. Lebeck. **Exact analysis of the cache behavior of nested loops**. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 286–297. ACM Press, 2001.

[CoKi95] Stephanie Coleman and Kathryn S. McKinley. **Tile size selection using cache organization and data layout**. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290. ACM Press, 1995.

[CoLo99] Robert Cohn and P. Geoffrey Lowney. **Feedback Directed Optimization in Compaq's Compilation Tools for Alpha**. In *Proceedings of the Second Workshop on Feedback-Directed Optimization, held in conjunction with MICRO-33*, pages 3–12, 1999.

[Cxml] **Compaq eXtended Mathematic Library Reference Guide**.

[DeHiWa97] J. Dean and J.E. Hicks and C.A. Waldspurger and W.E. Weihl and G.Z. Chrysos. **ProfileMe : Hardware support for instruction level profiling on out-of-order processors**. In *In Proceedings of the 30^{th} International Symposium on Microarchitecture*, NC, December 1997.

[Epic] Harsh Sharangpani. **Itanium Microarchitecture Design**. *Microprocessor Forum*, 1999.

[GhMaMa99] Somnath Ghosh and Margaret Martonosi and Sharad Malik. **Cache miss equations : a compiler framework for analyzing and tuning memory behavior**. *ACM Trans. Program. Lang. Syst.*, 21(4) :703–746, 1999.

[GiMu86] Philip B. Gibbons and Steven S. Muchnick. **Efficient instruction scheduling for a pipelined architecture**. In *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 11–16. ACM Press, 1986.

[IPF2] **Intel Itanium2 Processor Reference Manual for Software Development and Optimization**. http ://developer.intel.com/design/itanium2/manuals.

[Kap] **KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX**. http ://www.hp.com/techsevers/software/kap.html.

[Ke00] Ken Kennedy. **Fast greedy weighted fusion**. In *Proceedings of the 14th international conference on Supercomputing*, pages 131–140. ACM Press, 2000.

[KiFr03] Thomas Kistler and Michael Franz. **Continuous program optimization : A case study**. *ACM Trans. Program. Lang. Syst.*, 25(4) :500–548, 2003.

[KiKnOB00] T. Kisuki and P. Knijnenburg and M. O'Boyle and H. Wijshoff. **Iterative compilation in program optimization**. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[KiKnOB00+] Toru Kisuki and Peter M. W. Knijnenburg and Michael F. P. O'Boyle. **Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation**. In *IEEE PACT*, pages 237–248, 2000.

[KKGO00] T. Kisuki and P. Knijnenburg and K. Gallivan and M. O'Boyle. **The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling**. In *Parallel Architectures and Compilation Techniques (PACT'00)*. IEEE Computer Society Press, October 2001.

[KuHiHi04] Prasad Kulkarni and Stephen Hines and Jason Hiser and David Whalley and Jack Davidson and Douglas Jones. **Fast searches for effective optimization phase sequences**. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182. ACM Press, 2004.

[KuZhMo03] Prasad Kulkarni and Wankang Zhao and Hwashin Moon and Kyunghwan Cho and David Whalley and Jack Davidson and Mark Bailey and Yunheung Paek and Kyle Gallivan. **Finding effective optimization phase sequences**. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23. ACM Press, 2003.

[LaRoWo91] Monica D. Lam and Edward E. Rothberg and Michael E. Wolf. **The cache performance and optimizations of blocked algorithms**. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74. ACM Press, 1991.

[McTe96] Kathryn S. McKinley and Olivier Temam. **A quantitative analysis of loop nest locality**. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 94–104. ACM Press, 1996.

[MiCaFe97] Nicholas Mitchell and Larry Carter and Jeanne Ferrante and Karin Hogstedt. **Quantifying the Multi-level Nature of Tiling Interactions**. In *Languages and Compilers for Parallel Computing*, pages 1–15, 1997.

[MoBo01] Antoine Monsifrot and François Bodin. **Computer aided hand tuning (CAHT) : "applying case-based reasoning to performance tuning"**. In *Proceedings of the 15th International Conference on Supercomputing*, pages 196–203. ACM Press, 2001.

[MoBoQu02] Antoine Monsifrot and François Bodin and Rene Quiniou. **A Machine Learning Approach to Automatic Production of Compiler Heuristics**. In *Proceedings of the 10th International Conference on Artificial Intelligence : Methodology, Systems, and Applications*, pages 41–50. Springer-Verlag, 2002.

[Mu97] Steven S. Muchnick. **Advanced compiler design and implementation**. Morgan Kaufmann Publishers Inc., 1997.

[NaJuLa94] Juan J. Navarro and Toni Juan and Tomás Lang. **MOB forms : a class of multilevel block algorithms for dense linear algebra operations**. In *Proceedings of the 8th international conference on Supercomputing*, pages 354–363. ACM Press, 1994.

[OProfile] **OProfile project**. http ://oprofile.sourceforge.net.

[ORC] **Open Research Compiler**. http ://ipf-orc.sourceforge.net.

[PerfMon3] **Perfmon project**. http ://www.hpl.hp.com/research/linux/perfmon.

[Pi93] Shlomit S. Pinter. **Register allocation with instruction scheduling**. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 248–257. ACM Press, 1993.

[RiTs98] Gabriel Rivera and Chau-Wen Tseng. **Data transformations for eliminating conflict misses**. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 38–49. ACM Press, 1998.

[RiTs98+] Gabriel Rivera and Chau-Wen Tseng. **Eliminating conflict misses for high performance architectures**. In *Proceedings of the 12th international conference on Supercomputing*, pages 353–360. ACM Press, 1998.

[RiTs99] Gabriel Rivera and Chau-Wen Tseng. **A Comparison of Compiler Tiling Algorithms**. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 168–182. Springer-Verlag, 1999.

[SmSo95] James E. Smith and Gurindar S. Sohi. **The microarchiture of superscalar processors**. *Proceedings of IEEE*, 1995. http ://members.jcom.home.ne.jp/kgoto.

[SrLe98] Srikanth T. Srinivasan and Alvin R. Lebeck. **Load latency tolerance in dynamically scheduled processors**. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 148–159. IEEE Computer Society Press, 1998.

[StAmMaOR03] Mark Stephenson and Saman Amarasinghe and Martin Martin and Una-May O'Reilly. **Meta optimization : improving compiler heuristics with machine learning**. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90. ACM Press, 2003.

[TeGrJa93] O. Temam and E. D. Granston and W. Jalby. **To copy or not to copy : a compile-time technique for assessing when data copying should be used to eliminate cache conflicts**. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 410–419. ACM Press, 1993.

[TrVaVaAu03] Spyridon Triantafyllis and Manish Vachharajani and Neil Vachharajani and David I. August. **Compiler optimization-space exploration**. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215. IEEE Computer Society, 2003.

[VoEi00] Michael J. Voss and Rudolf Eigenmann. **ADAPT : Automated De-Coupled Adaptive Program Transformation**. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 163. IEEE Computer Society, 2000.

[VoEi01] Michael J. Voss and Rudolf Eigemann. **High-level adaptive program optimization with ADAPT**. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102. ACM Press, 2001.

[VTune] **Intel VTune Performance Analysers**. http ://www.intel.com/software/products/vtune.

[Wa02] Gregory Watts. **Vers un environnement et une méthodologie d'optimisation de programmes sur la hiérarchie mémoire**. *PH.D. Thesis*, 2002.

[We97] Volker Weispfenning. **Complexity and uniformity of elimination in Persburger arithmetic**. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 48–53. ACM Press, 1997.

[WhSo90] D. Whitfield and M. L. Soffa. **An approach to ordering optimizing transformations**. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 137–146. ACM Press, 1990.

[WoLa91] Michael E. Wolf and Monica S. Lam. **A data locality optimizing algorithm**. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44. ACM Press, 1991.

[ZhCaWh02] Wankang Zhao and Baosheng Cai and David Whalley and Mark W. Bailey and Robert van Engelen and Xin Yuan and Jason D. Hiser and Jack W. Davidson and Kyle Gallivan and Douglas L. Jones. **VISTA : a system for interactive code improvement**. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 155–164. ACM Press, 2002.