Université de Montréal

Compilation optimisante à l'aide de métaheuristiques

par

Fernanda Kri

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiae Doctor (Ph.D.)
en informatique

Décembre 2002

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée :

Compilation optimisante à l'aide de métaheuristiques

présenté par :

Fernanda Kri

a été évaluée par un jury composé des personnes suivantes :

El Mostapha Aboulhamid

(président-rapporteur)

Marc Feeley

(directeur de recherche)

Jacques Ferland

(membre du jury)

Etienne Gagnon

(examinateur externe)

Ettore Merlo

(représentant du doyen de la FES)

Abstract

The construction of efficient compilers is very complex, since it has to face several optimization problems and it depends on the characteristics of the architecture of the machines for which they generate code. In this thesis we propose the use of the metaheuristic methods in compilation. Our thesis is that this makes it possible to obtain efficient code while simplifying the construction of the compilers. First, we show the viability to use the metaheuristic methods in compilation by replacing GNU C's register allocator by an allocator based on a genetic algorithm and by an allocator based on simulated annealing. Our allocators show a performance similar to GNU C. Afterwards, we define a unified approach to carry out the compilation using metaheuristic methods. This approach eliminates the weaknesses found in the traditional approaches. It allows to take account of the interdependencies between various problems present in the process of compilation ; it does not make assumptions on the characteristics of the source programs so it permits to be more homogeneous on all of them. Finally, the approach allows the compiler to adapt easily to new changes to the architecture, since it is not very dependent on the characteristics of the machine. We test our approach on the problems of register allocation and instruction scheduling. We carry out an analysis of performance by comparing it with the more traditional approaches for these problems and we obtain profits on the speed of the generated code varying between -2% and 26%.

Keywords : optimizing compiler, metaheuristic methods.

Sommaire

La construction de compilateurs performants est très complexe puisqu'elle comporte plusieurs problèmes d'optimisation et qu'elle est dépendante des caractéristiques de l'architecture des machines pour lesquelles ils génèrent du code. Dans cette thèse nous proposons l'utilisation des méthodes métaheuristiques en compilation. Notre thèse est que ceci permet d'obtenir du code performant tout en simplifiant la construction des compilateurs. Au début, nous démontrons la viabilité d'utiliser les méthodes métaheuristiques en compilation en remplaçant l'allocateur de registres de GNU C par un allocateur basé sur un algorithme génétique et par un allocateur basé sur un recuit simulé. Nos allocateurs montrent une performance semblable à celle de GNU C. Après, nous définissons une approche unifiée pour réaliser la compilation à l'aide de méthodes métaheuristiques. Cette approche élimine les faiblesses des approches traditionnelles. Elle permet de tenir compte des interdépendances entre différents problèmes impliqués dans le processus de compilation ; elle ne fait pas de suppositions sur les caractéristiques des programmes sources pouvant, ainsi, être plus homogène sur l'ensemble des programmes. Finalement l'approche permet d'adapter facilement le compilateur aux changements apportés à l'architecture puisqu'elle est peu dépendante des caractéristiques de la machine. Nous testons notre approche sur les problèmes d'allocation de registres et d'ordonnancement d'instructions. Nous réalisons une analyse de performance en le comparant avec les approches plus classiques pour ces problèmes et nous obtenons de gains de vitesse du code généré qui varient entre -2% et 26%.

Mots clés : compilation optimisante, méthodes métaheuristiques.

Table des matières

1 Introduction **1**

1.1 Mise en contexte 2

1.1.1 La compilation 2

1.1.2 Les méthodes métaheuristiques 5

1.2 Pourquoi utiliser des métaheuristiques ? 6

1.3 Quels problèmes en compilation sont abordables en utilisant des métaheuristiques ? 8

1.4 Est-ce viable d'utiliser des métaheuristiques dans la compilation ? 8

1.5 Est-ce profitable d'utiliser les métaheuristiques en compilation ? 9

1.6 Est-ce que c'est coûteux ? 10

1.7 Quels sont les avantages ? 10

1.8 Contributions et limitations 11

1.9 Structure du document 13

2 État de l'art **14**

2.1 La compilation 14

2.2 Le problème d'allocation de registres 15

2.2.1 Les allocateurs par coloriage de graphe 17

2.2.2 Les allocateurs à la volée 23

2.2.3 L'allocation intraprocédurale 24

2.2.4 L'allocation interprocédurale 24

2.2.5 L'allocation globale 25

2.3 Le problème d'ordonnancement d'instructions 25

2.3.1 Ordonnancement statique et ordonnancement dynamique 27

2.3.2 Ordonnancement de boucles et ordonnancement des blocs de base 28

2.3.3 L'allocation et l'ordonnancement 29

2.4 Les méthodes métaheuristiques 31

2.4.1 Algorithmes génétiques 32

2.4.2 Recuit simulé 36

2.4.3 Le coloriage de graphe avec des métaheuristiques 38

2.4.4 L'ordonnancement avec les métaheuristiques 39

2.4.5 Les métaheuristiques et la compilation 40

2.5 Résumé 40

3 Une première expérience avec le compilateur GNU C 42

3.1 La conception 44

3.1.1 La représentation du problème 44

3.1.2 La fonction de coût 45

3.2 Les graphes de test 46

3.3 L'ajustement de paramètres 47

3.3.1 L'algorithme génétique 48

3.3.2 Le recuit simulé 53

3.4 Résultats 53

3.4.1 Le compilateur GNU C 55

3.4.2 Analyse de performance 56

3.5 Résumé 59

4 Compilateur génétique : une approche unifiée 60

4.1 Les approches traditionnelles 61

4.2 L'idée de base 64

4.2.1 La représentation 66

4.2.2 Les opérateurs génétiques 67

4.2.3 La fonction de coût 68

4.3 La conception 69

4.3.1 La représentation pour l'allocation de registres 71

4.3.2 La représentation pour l'ordonnancement d'instructions 73

4.3.3 Les opérateurs génétiques 75

4.3.4 Le meilleur choix 77

4.3.5 La fonction de coût 80

4.3.6 Le compilateur 82

4.4 L'implantation 84

4.5 Résumé 87

5 Résultats **89**

5.1 Ajustement d'AG-Op 89

5.1.1 La fonction de coût 90

5.1.2 L'individu initial 91

5.1.3 Les opérateurs de croisement 94

5.1.4 Les paramètres 97

5.2 Analyse de performance 100

5.3 Résumé 120

6 Conclusions et travaux futurs **121**

6.1 Contributions 121

6.2 Travaux futurs 124

Bibliographie **135**

A Résultat de l'ajustement de paramètres **xviii**

A.1 AG-Allocateur xviii

A.2 RS-Allocateur xx

B Code assembleur généré par GNU C et RS-Allocateur **xxi**

C L'implantation d'AG-Op **xxiv**

D Résultat de l'ajustement de paramètres pour AG-Op **xxviii**

E Code source pour le programme de test *arithm* **xxxiii**

Table des figures

1.1 Structure d'un compilateur classique 2

2.1 (a) Exemple de programme. (b) Intervalles de vie. 17

2.2 Programme avec des différentes assignations des variables aux registres . 18

2.3 Allocateur de type Chaitin 19

2.4 Allocateur proposé par Briggs et al. 21

2.5 Allocateur proposé par Appel 22

2.6 (a) Exemple de programme . (b) Graphe de dépendance. (c) Programme avec ordonnancement. 26

2.7 Pseudocode de l'ordonnancement par liste 29

2.8 Opérateur de croisement classique 33

2.9 Opérateur de mutation classique. 33

2.10 Pseudocode d'un algorithme génétique 34

2.11 Opérateur OBX 36

2.12 Opérateur PBX 36

2.13 Pseudocode pour un recuit simulé 37

3.1 Graphe de convergence en utilisant l'opérateur de mutation par échange 49

3.2 Graphe de convergence en utilisant l'opérateur de mutation par descente 50

3.3 Graphe de convergence pour l'opérateur de mutation par descente 51

3.4 Graphe de convergence pour les opérateurs de mutation 51

3.5 Graphe de convergence en utilisant les opérateurs de croisement 52

3.6 Graphe de convergence en utilisant différentes tailles de population . . . 53

3.7 Graphe de convergence en utilisant différentes températures (g-96) . . . 54

3.8 Graphe de convergence en utilisant différentes températures (graph61) . 55

4.1 Exemple d'individu utilisé dans plusieurs optimisations 67

4.2 Structure de notre AG-Op 68

4.3 Exemple de programme et trois représentations possibles pour l'allocation 74

4.4 Exemple d'ordonnancement réalisable et irréalisable 75

4.5 Opérateur spécifique pour l'ordonnancement 76

4.6 Exemple d'allocation et d'ordonnancement 79

4.7 Individu décrivant l'ordonnancement d'instructions 80

4.8 Structure d'AG-Op pour les problèmes d'allocation de registres et d'ordonnancement d'instructions 82

4.9 Structure d'AG-Op avec un compilateur idéal 85

4.10 Structure d'AG-Op avec lcc 87

5.1 Convergence d'AG-Op avec les deux fonctions de coût (problème bj) . . 90

5.2 Convergence d'AG-Op avec les deux fonctions de coût (problème tfftdp) 91

5.3 Convergence d'AG-Op avec les trois individus initiaux (bj) 92

5.4 Convergence d'AG-Op avec les trois individus initiaux (bj) 92

5.5 Convergence d'AG-Op avec les trois individus initiaux (bj) 93

5.6 Convergence d'AG-Op avec les trois individus initiaux (bj) 94

5.7 Convergence d'AG-Op avec différents opérateurs de croisement pour l'allocation de registres (bj) 95

5.8 Convergence d'AG-Op avec différents opérateurs de croisement pour l'allocation de registres (bj) 96

5.9 Convergence d'AG-Op avec différents opérateurs de croisement pour l'ordonnancement d'instructions (bj) 96

5.10 Convergence d'AG-Op avec différents opérateurs de croisement pour l'ordonnancement d'instructions (bj) 97

5.11 Convergence d'AG-Op en utilisant différentes tailles de population (bj) . 98

5.12 Convergence en utilisant différents pourcentages de mutation 99

5.13 Convergence en utilisant différents pourcentages de sélection 100

5.14 Convergence d'AG-Op sur *bj8* 107

5.15 Convergence d'AG-Op sur *fftdp8* 108

5.16 Code assembleur pour *bubble* 112

5.17 Code assembleur pour *selection* 113

5.18 Pourcentage d'amélioration par rapport au pire cas 116

5.19 Pourcentage d'amélioration par rapport à lcc et l'ordonnancement par liste117

5.20 Pourcentage d'amélioration par rapport au coloriage et l'ordonnancement par liste 117

Liste des tableaux

3.1 Caractéristiques des graphes de test 47

3.2 Résultats obtenus en utilisant différentes tailles de population 48

3.3 Résultats obtenus en utilisant RS-Allocateur et GA-Allocateur 56

3.4 Programmes test utilisés 57

3.5 Temps de compilation requis par RS-Allocateur, GA-Allocateur et GNU C 57

3.6 Temps d'exécution des programmes de test 58

3.7 Nombre de fois que la procedure est exécutée 59

5.1 Description des programmes de test utilisés 102

5.2 Caractéristiques des programmes de test utilisés 103

5.3 Temps d'exécution des programmes de test compilés avec GNU C et *lcc*. 104

5.4 Temps de compilation obtenus par *lcc* et AG-Op. 105

5.5 Performance des différentes heuristiques pour l'ordonnancement d'instruction par liste 106

5.6 Temps d'exécution obtenus pour les programmes de test qui utilisent peu de variables 109

5.7 Résultats obtenus avec l'allocateur de *lcc* et avec l'allocateur par coloriage 114

5.8 Temps d'exécution en compilant avec toutes les approches 115

5.9 Amélioration obtenue par AG-Op par rapport à lcc+sch et color+sch . . 118

5.10 Amélioration obtenue par AG-Op par rapport au pire cas 119

A.1 Résultats opérateur de mutation 1 : échange xix

A.2 Résultats opérateur de mutation 2 : descente. TP=100 xix

A.3 Résultats opérateur de mutation 2 : descente. TP=50 xix

A.4 Résultats opérateur de croisement xx

A.5 Résultats pour différentes tailles de population xx

A.6 Résultats pour différentes températures xx

D.1 Résultats obtenus pour les différents individus initiaux. TP=10, Mut=1% xxviii

D.2 Résultats obtenus pour les différents individus initiaux. TP=10, Mut=5% xxix

D.3 Résultats obtenus pour les différents individus initiaux. TP=20, Mut=1% xxix

D.4 Résultats obtenus pour les différents individus initiaux. TP=20, Mut=5% xxix

D.5 Résultats obtenus pour les différents opérateurs de croisements pour le problème d'allocation de registres. TP=10 xxx

D.6 Résultats obtenus pour les différents opérateurs de croisements pour le problème d'allocation de registres. TP=20 xxx

D.7 Résultats obtenus pour les différents opérateurs de croisements pour le problème d'ordonnancement d'instructions. TP=10 xxx

D.8 Résultats obtenus pour les différents opérateurs de croisements pour le problème d'ordonnancement d'instructions. TP=20 xxxi

D.9 Résultats obtenus pour les différentes tailles de population xxxi

D.10 Résultats obtenus pour les différentes probabilités de mutations xxxi

D.11 Résultats obtenus pour les différents pourcentages de sélection xxxii

Remerciements

Je veux juste dire MERCI...

MERCI à Dieu...

MERCI à Daniel, mon mari, pour tout. Pour son appui inconditionnel, pour l'encouragement, pour le support moral et technique, pour son amour...

MERCI à mes filles, Avril et Cloé, pour la motivation et l'inspiration...

MERCI à ma famille, que même de loin a été toujours à mon côté...

MERCI à Danny, Wissam, Etienne, Jean-François, Eric et Ouiza pour leur aide, pour l'encouragement et surtout pour tous les bons moments passés au laboratoire...

MERCI encore à Danny et Etienne pour faire de cette thèse un document lisible...

MERCI à mon directeur de recherche, Marc Feeley, pour me laisser la liberté de travailler à ma façon...

MERCI à Luz Margarita du Ministère d'Éducation et Planification (MIDEPLAN) du gouvernement du Chili pour son constant appui...

MERCI au MIDEPLAN et à l'Université de Santiago pour l'appui financière...

MERCI à tous ceux et celles qui ont collaboré de quelque façon à la réussite de ce projet...

MERCI.

Chapitre 1

Introduction

La construction de compilateurs performants est très complexe puisqu'elle comporte plusieurs problèmes d'optimisation et qu'elle est dépendante des caractéristiques de l'architecture. Nous proposons l'utilisation de méthodes métaheuristiques en compilation. Notre thèse est que ceci permet d'obtenir du code performant tout en simplifiant la construction des compilateurs.

Indubitablement, la proposition de cette thèse génère plusieurs questions. Pourquoi utiliser des métaheuristiques ? Quels problèmes de compilation peuvent être abordés en utilisant des métaheuristiques ? Est-ce viable d'utiliser des métaheuristiques en compilation ? Est-ce profitable d'utiliser les métaheuristiques en compilation ? Est-ce que c'est coûteux ? Quels sont les avantages ? Notre travail vise à répondre à ces questions.

Pour mieux comprendre pourquoi nous croyons que l'utilisation de méthodes métaheuristiques pour résoudre les problèmes d'optimisation que nous retrouvons en compilation sera fructueux, nous présentons une brève mise en contexte sur ces deux domaines.

1.1 Mise en contexte

1.1.1 La compilation

De façon générale, la compilation consiste à traduire un programme écrit dans le langage source (généralement de haut niveau) en un programme équivalent écrit dans un langage cible (de bas niveau) compréhensible par la machine. Pour ce faire, les compilateurs doivent résoudre une série de problèmes tout en tenant compte de différentes contraintes, telles que les caractéristiques de l'architecture (jeu d'instructions, structure du pipeline, nombre et type de registres, etc.), le temps de compilation désiré et les caractéristiques particulières des programmes sources.

La structure traditionnelle d'un compilateur comporte trois parties [AU78] : la partie frontale (*front-end*), qui prend le code source et le traduit en code intermédiaire ; l'optimiseur, qui effectue des optimisations indépendantes de l'architecture sur le code intermédiaire ; la partie arrière (*back-end*), qui génère le code objet à partir du code intermédiaire en effectuant quelques optimisations dépendantes de l'architecture (figure 1.1).

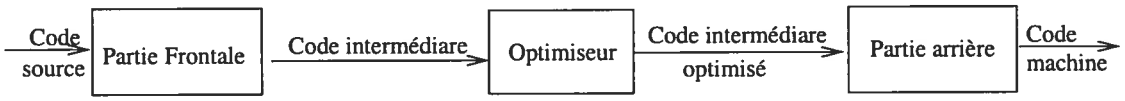


FIG. 1.1 – Structure d'un compilateur classique

Au cours de la phase d'optimisation, le compilateur doit résoudre plusieurs problèmes afin de générer du code performant. Certains de ces problèmes doivent être résolus afin de simplement générer du code objet tandis que d'autres visent à faire des optimisations. La résolution de ces dernières est optionnelle. Nous donnons une brève description de quelques-uns de ces problèmes.

- Sélection d'instructions : trouver les instructions machine les plus appropriées pour une représentation intermédiaire donnée.
- Transformation de programme (*inlining* et *loop unrolling*) : insérer la définition

des procédures à leur site d'appel et dérouler les boucles pour minimiser les coûts associés aux branchements et au mécanisme d'appel de fonction.
- Optimisation de cache : trouver les meilleurs emplacements en mémoire pour les procédures et données du programme afin de minimiser les conflits de cache.
- Ordonnancement d'instructions : trouver le meilleur ordre d'exécution des instructions.
- Allocation de registres : assigner les variables présentes dans le code intermédiaire aux registres de la machine.

Ces problèmes sont des problèmes d'optimisation et ils sont normalement résolus à l'aide d'heuristiques. Prenons à titre d'exemple l'ordonnancement d'instructions. Pour résoudre ce problème, le programme doit d'abord être représenté par un graphe de dépendances. Chaque nœud correspond à une instruction du programme et les arêtes représentent un lien causal entre les instructions. Par exemple, si une instruction i_1 calcule une valeur qui est utilisée par une instruction i_2, alors i_1 doit s'exécuter avant i_2 et, donc, il y a une arête de i_1 à i_2 dans le graphe. En utilisant cette représentation, le problème est vu comme «la recherche d'une permutation des nœuds qui permet de minimiser le temps d'exécution, tout en respectant les dépendances indiquées par le graphe de dépendances».

Les heuristiques que les compilateurs utilisent sont développées en fonction du problème et des caractéristiques de l'architecture ciblée. Cependant, étant donné que les architectures deviennent rapidement de plus en plus complexes, les heuristiques se basent sur des simplifications et/ou des généralisations des architectures pour pouvoir continuer à résoudre simplement les problèmes d'optimisation. Par exemple, pour le problème d'ordonnancement d'instructions, le compilateur obtient une permutation des nœuds du graphe de dépendances en assignant les nœuds selon une fonction de priorité (qui respecte les dépendances dans le graphe). Cette fonction de priorité est une heuristique qui essaie d'estimer le temps d'exécution de chaque instruction en tenant compte des caractéristiques du programme et du *pipeline* de la machine. Généralement, les heu-

ristiques fonctionnent assez bien. Il faut comprendre que le concepteur procède à des tests à l'aide de programmes soit-disant typiques pour valider l'heuristique. Cependant, il n'y a aucune garantie que cette heuristique est bien adaptée aux programmes déviant de la norme, comme par exemple de très gros modules ou des programmes générés par d'autres programmes. Cela s'explique par le fait que le concepteur de l'heuristique construit un modèle qui fait abstraction à l'excès des caractéristiques de l'architecture et du programme source.

Les compilateurs actuels implantent un grand nombre d'optimisations. Puisque tout dépendant des besoins de l'usager et des caractéristiques du programme, l'application d'une optimisation ne génère pas nécessairement un code plus performant que lorsque l'optimisation n'est pas effectuée, on laisse la possibilité à l'usager de choisir quelles optimisations effectuer (par exemple, avec des options de ligne de commande).

Finalement, plusieurs problèmes liés au compilateur sont interdépendants. Mais, pour des raisons de simplicité et de modularité, ils sont résolus séparément par les compilateurs actuels. Le fait d'ignorer ces interdépendances force à appliquer certaines optimisations plusieurs fois ou à choisir un "meilleur" ordre pour les appliquer. Par exemple, le compilateur GNU C effectue l'optimisation des branchements trois fois. La première application s'effectue tout de suite après la génération du code intermédiaire. La deuxième s'effectue après l'élimination des expressions communes. Et la troisième, juste avant la génération du code objet. Ceci est dû au fait que les autres optimisations effectuées par GNU C peuvent créer de nouvelles possibilités pour l'optimisation des branchements. Aussi, l'élimination des expressions communes est effectuée deux fois : une fois avant l'optimisation des boucles et l'autre, après. Pour faire face à la dépendance entre les problèmes d'allocation de registres et d'ordonnancement d'instructions, GNU C effectue l'ordonnancement d'instructions deux fois : une fois avant et une fois après l'allocation de registres. On trouve aussi des compilateurs, tel GHC (*Glasgow Haskell Compiler*), qui itèrent plusieurs phases de transformation jusqu'à ce qu'un certain critère (heuristique) soit atteint.

1.1.2 Les méthodes métaheuristiques

Les méthodes métaheuristiques sont des méthodes d'optimisation non classiques basées sur des techniques d'intelligence artificielle et qui sont communément utilisées pour résoudre des problèmes complexes (habituellement NP-durs). Les méthodes les plus connues sont les algorithmes génétiques [Hol75], les algorithmes de recuit simulé [KGV83], les algorithmes de recherche tabou [GL95] et les systèmes de fourmis [DMC91].

Les méthodes métaheuristiques n'assurent pas l'obtention de solutions optimales et il n'y a presqu'aucun résultat théorique qui garantisse de bonnes performances (sauf pour les algorithmes génétiques où le théorème du schéma garantit de façon probabiliste la convergence de l'algorithme dans quelques cas particuliers). Par contre, les métaheuristiques ont été utilisées avec beaucoup de succès au cours des deux dernières décennies pour résoudre des problèmes réels de haute complexité.

Afin de résoudre un problème en utilisant des métaheuristiques, il faut définir une fonction de coût et une représentation du problème. La fonction de coût est la fonction que les métaheuristiques essaient d'optimiser (minimiser ou maximiser). La représentation est une codification des solutions possibles au problème.

Toutes ces méthodes commencent avec une solution initiale (ou un ensemble de solutions initiales) générée le plus souvent de façon aléatoire. En utilisant ses propres paradigmes, chaque métaheuristique apporte des modifications à cette solution initiale dans le but de l'améliorer. Chaque nouvelle solution est évaluée en utilisant la fonction de coût afin de connaître sa qualité. Le processus continue jusqu'à ce qu'un critère d'arrêt soit satisfait.

Les méthodes métaheuristiques ont été largement utilisées pour résoudre des problèmes théoriques comme le coloriage de graphe, le voyageur de commerce, le problème de Prim, etc. De plus, elles ont aussi montré de très bonnes performances sur des problèmes réels comme le *cutting stock*, la planification de routes, l'entraînement de réseaux de neurones, etc.

Malheureusement, à cause du manque de résultats théoriques, nous ne pouvons pas établir *a priori* si une méthode métaheuristique peut être performante dans la résolution d'un problème donné. Nous ne pouvons pas non plus savoir quelle métaheuristique ni quelle représentation sont les plus appropriées pour le problème. Donc, la seule façon de les découvrir est par l'expérimentation. Typiquement, lorsqu'on décide d'utiliser une métaheuristique, on en choisit arbitrairement une. La représentation est définie en se basant sur des représentations pour des problèmes semblables ou en se basant sur l'intuition. Après, en fonction des résultats obtenus, la représentation ou la métaheuristique peuvent être modifiées dans le but d'obtenir de meilleurs résultats.

Les principaux avantages des métaheuristiques par rapport aux heuristiques sont les suivants.

- Sur un grand nombre de problèmes NP-durs, les heuristiques donnent des résultats d'assez bonne qualité lorsque les problèmes sont de petite taille. Cependant, lorsque la taille du problème est plus grande, les heuristiques donnent des solutions de mauvaise qualité. Parfois, elles ne sont même pas capables de produire une solution en respectant les contraintes de temps de traitement et/ou de mémoire. Par contre, les métaheuristiques peuvent résoudre des problèmes de taille plus grande sans augmenter de façon importante leurs besoins en temps et en mémoire (à condition que la représentation du problème soit appropriée).
- Si la fonction de coût est bien définie (c'est-à-dire, qu'elle donne une bonne évaluation de la qualité de la solution) une métaheuristique a du succès de façon plus consistante sur tout l'ensemble des instances.

1.2 Pourquoi utiliser des métaheuristiques ?

Dans plusieurs cas, les métaheuristiques sont plus performantes que les heuristiques. Surtout lorsque les problèmes sont de grande taille ou lorsque le problème est multi-objectif.

En compilation, plusieurs problèmes sont simplifiés afin de pouvoir être résolus à l'aide d'heuristiques dans un temps raisonnable. Par exemple, le problème d'allocation de registres est souvent résolu localement à chaque procédure ou même à chaque bloc de base. Une résolution globale à l'aide d'heuristiques traditionnelles est trop complexe soit par la mémoire requise, soit par le temps d'exécution trop élevé, soit par la mauvaise qualité de la solution lorsque les problèmes sont de grande taille. La même situation se produit avec le problème d'ordonnancement d'instructions, où le problème est résolu au niveau de blocs de base ou au niveau du corps des boucles.

Nous considérons comme fondamentale l'existence des dépendances entre les différents problèmes que le compilateur doit résoudre. Ces dépendances sont presque toujours ignorées par les approches de compilation conventionnelles même si elles ont une incidence importante sur la qualité du code objet généré. Nous croyons que la résolution de ces problèmes simultanément peut apporter une amélioration importante à la qualité du code objet. Les compilateurs classiques n'utilisent pas cette approche car la complexité des problèmes ne permet pas de trouver des heuristiques simples qui réussissent à résoudre plus d'un problème à la fois.

L'utilisation des métaheuristiques pourrait permettre de résoudre les problèmes d'optimisation liés à la compilation de façon plus globale que les heuristiques utilisées actuellement. De plus, elles permettraient de résoudre plus d'un problème à la fois et donc de tenir compte des dépendances entre les problèmes.

Finalement, comme la métaheuristique ne fait pas de suppositions sur les caractéristiques des programmes sources, sa performance est plus homogène sur l'ensemble des programmes à compiler. On peut donc espérer que les optimisations seront de bonne qualité même pour les programmes déviant de la normale.

1.3 Quels problèmes en compilation sont abordables en utilisant des métaheuristiques ?

Tous les problèmes d'optimisation liés à la compilation sont de bons candidats, mais principalement ceux actuellement résolus à l'aide d'heuristiques qui abstraient à l'excès le problème, ceux résolus au niveau local et ceux qui sont interdépendants.

La sélection d'instructions, l'allocation de registres et l'ordonnancement d'instructions sont les principaux candidats car ils ont un grand impact sur la qualité du code généré. Ils sont interdépendants et, en général, sont résolus à très petite échelle.

1.4 Est-ce viable d'utiliser des métaheuristiques dans la compilation ?

Trouver une représentation et une fonction de coût pour le problème visé, permet d'appliquer une métaheuristique en compilation. Mais, nous ne pouvons pas savoir *a priori* si nous obtiendrons des résultats compétitifs avec les méthodes traditionnelles.

Afin de répondre à cette question, nous avons effectué une première expérience en remplaçant l'heuristique utilisée par le compilateur pour un problème spécifique par une métaheuristique (chapitre 3). Cette façon est la plus directe d'utiliser les méthodes métaheuristiques en compilation. Nous abordons le problème d'allocation de registres parce qu'il s'exprime comme un problème de coloriage de graphe et ce dernier a été résolu efficacement à l'aide de méthodes métaheuristiques. Cette première expérience nous permet de confirmer que l'utilisation des métaheuristiques en compilation est viable. Cependant, comme nous verrons dans le chapitre 3, nos résultats ne démontrent pas d'amélioration par rapport aux méthodes traditionnelles. Or, nous croyons que pour obtenir des gains, il faut utiliser les méthodes métaheuristiques là où elles obtiennent de meilleures performances, c'est-à-dire, lorsque le problème est de très grande taille

ou lorsque le problème est multi-objectif. En d'autres mots, là où les heuristiques n'ont pas habituellement de bonnes performances, soit à cause du temps de traitement, soit à cause de la qualité de la solution.

1.5 Est-ce profitable d'utiliser les métaheuristiques en compilation ?

Pour que l'utilisation des métaheuristiques soit profitable, nous croyons qu'il faut utiliser une approche plus agressive, c'est-à-dire, résoudre des problème simultanément ou globalement ou même les deux à la fois.

Dans le chapitre 4, nous proposons un compilateur qui utilise les métaheuristiques mais qui maintient la structure traditionnelle des compilateurs [AU78] (figure 1.1). L'optimiseur de notre approche est basé sur un algorithme génétique qui effectue toutes les optimisations désirées (indépendantes ou dépendantes de l'architecture) au niveau du code intermédiaire tout en tenant compte de la dépendance entre les problèmes.

Nous proposons l'utilisation d'une fonction de coût non traditionnelle : le temps réel d'exécution du code généré. Ce choix est basé sur l'importance d'utiliser une fonction de coût précise pour permettre aux méthodes métaheuristiques d'atteindre de bonnes performances. Si le critère d'optimisation est le temps d'exécution du code généré, alors la fonction de coût la plus exacte est le temps réel d'exécution.

Il faut noter que, même si plusieurs exécutions d'un même programme sur la même machine ne donnent pas nécessairement les mêmes temps d'exécution, l'utilisation du temps réel comme fonction de coût est tout de même plus exacte que l'utilisation d'une estimation basée sur des données du programme comme le nombre de débordement, le nombre de *stalls* dans le pipeline et le nombre d'instructions. Étant donné que nous résolvons plus d'un problème simultanément, trouver une estimation qui permet d'établir qu'un code a un temps d'exécution plus petit ou plus grand qu'un autre est non trivial.

De plus, l'utilisation du temps réel permet de prendre en compte les effets qui ne sont pas mesurables en considérant uniquement le code statique. Par exemple, les défauts de page et de cache, les caractéristiques (propriétaires) d'une architecture (organisation des pipelines, circuit d'ordonnancement dynamique d'instructions, etc).

Nous avons testé notre approche en résolvant simultanément les problèmes d'allocation de registres et d'ordonnancement d'instructions.

1.6 Est-ce que c'est coûteux ?

L'utilisation de méthodes métaheuristiques est coûteuse par rapport au temps de compilation. Généralement, les métaheuristiques utilisent plus de temps que les heuristiques traditionnelles pour résoudre un problème mais elles sont capables de résoudre des problèmes de plus grande taille. De plus, l'utilisation du temps réel d'exécution comme fonction de coût ajoute un coût important au temps de compilation. Chaque solution (programme) générée par la métaheuristique doit être exécutée afin de mesurer son temps réel d'exécution. Donc, le temps requis par la métaheuristique est toujours plus grand que :

$$N * Te_i$$

où :

N est le nombre de solutions évaluées par la métaheuristique ;

Te_i est le temps d'exécution du programme généré i.

1.7 Quels sont les avantages ?

Les méthodes métaheuristiques permettent de résoudre plus d'un problème à la fois, de résoudre des problèmes de plus grande taille et d'utiliser le temps réel d'exécution

comme critère d'optimisation. Étant donné l'importance de ces trois facteurs nous pouvons nous attendre à obtenir des augmentations importantes de la qualité du code généré, surtout chez les programmes où les heuristiques traditionnelles ne fonctionnent pas très bien.

De nos jours, la complexité de la compilation est augmentée par les caractéristiques des architectures. Les *pipelines* qui permettent de profiter du parallélisme au niveau des instructions, la mémoire cache qui permet de diminuer les temps d'accès moyens aux données et aux instructions, les circuits permettant l'exécution hors-séquence et l'exécution spéculative font que le comportement d'un programme soit difficile à prédire. Donc, il est très difficile de déterminer la performance d'un programme en analysant son code. Par conséquent, les heuristiques, qui fonctionnaient bien autrefois sur des machines simples, n'arrivent pas à profiter des caractéristiques des machines actuelles. L'utilisation du temps réel d'exécution comme fonction de coût nous permet de mesurer l'impact réel des optimisations sur la machine et, donc, de profiter de ses caractéristiques.

D'autre part, l'utilisation du temps réel d'exécution permet de faire en sorte que le compilateur n'ait besoin que de très peu d'information sur l'architecture ciblée. Par conséquent, si l'architecture de la machine change, il suffit en principe de récompiler les programmes sur la nouvelle architecture. Le compilateur a donc la capacité de s'adapter automatiquement à l'environnement d'exécution.

1.8 Contributions et limitations

La recherche présentée dans cette thèse est une des premières qui considère l'utilisation de méthodes métaheuristiques en compilation et elle démontre que cette utilisation est viable et avantageuse. Elle se distingue des travaux de ce domaine par le traitement simultané de plusieurs problèmes d'optimisation et par l'évaluation de l'approche dans deux compilateurs bien connus pour le langage C (GCC et LCC).

L'expérience présentée dans le chapitre 3 est un cas particulier et simple de l'utilisation de métaheuristiques en compilation. Cette expérience démontre que l'approche par métaheuristique peut être un substitut satisfaisant (pas de perte de vitesse du code généré) pour un algorithme d'allocation de registre classique.

L'approche unifiée présentée dans le chapitre 4 permet de résoudre plusieurs problèmes impliqués dans la compilation de façon globale et simultanée. On pourrait, par exemple, réaliser la sélection d'instruction, l'allocation de registres et l'ordonnancement d'instruction en même temps et en considérant plusieurs procédures à la fois ou même le programme au complet.

Pour tester notre approche unifiée nous choisissons les problèmes d'allocation de registres et d'ordonnancement d'instructions. Au lieu de construire au complet un nouveau compilateur, nous utilisons un compilateur existant auquel nous incorporons notre optimiseur basé sur un algorithme génétique. Pour des raisons techniques qui seront exposées dans le chapitre 4, l'utilisation de ce compilateur impose des restrictions qui ne sont pas propres de l'approche unifiée que nous proposons. Voici les principales : 1. L'allocation de registres doit s'effectuer pour chaque procédure (allocation intraprocédurale) et non pour plusieurs procédures à la fois ou pour le programme au complet. 2. L'ordonnancement d'instructions doit se faire dans chaque bloc de base et non pour toute la procédure ou pour plusieurs procédures.

Même si ces restrictions limitent nos ambitions initiales, l'expérience demeure intéressante puisque le problème d'allocation de registres intraprocédurale ainsi que l'ordonnancement pour bloc de base sont des problèmes difficiles (NP-Dur) et leur résolution efficace a un impact important sur la qualité du code généré, comme le montrent les résultats présentés dans le chapitre 5. Dans le pire cas notre approche a un impact négatif négligeable (réduction de la vitesse du code généré de 1%). Cependant, nous avons trouvé des cas où le gain de vitesse est important (augmentation de la vitesse du code généré jusqu'à 26%).

1.9 Structure du document

Cette thèse est organisée comme suit. Le chapitre 2 présente les aspects théoriques nécessaires à la compréhension de notre recherche. Le chapitre 3 expose notre première expérience sur l'utilisation des métaheuristiques en compilation où nous remplaçons une heuristique traditionelle par une métaheuristique pour résoudre le problème d'allocation de registres. Au chapitre 4, nous présentons le cœur de notre recherche : une structure de compilateur basée sur un algorithme génétique qui permet de résoudre plusieurs problèmes de compilation simultanément. Nous y décrivons aussi la conception et l'implantation de la structure appliquée à la résolution des problèmes d'allocation de registres et d'ordonnancement d'instructions. Au chapitre 5, nous réalisons l'ajustement de paramètres propres à la métaheuristique et l'évaluation de la performance de notre approche. Finalement, le chapitre 6 expose nos conclusions et décrit quelques travaux futurs.

Chapitre 2

État de l'art

2.1 La compilation

Un compilateur optimisant [App97] (*optimizing compiler*) ne traduit pas seulement un programme d'un langage de haut niveau à un langage de bas niveau, mais il essaie aussi d'optimiser le code généré. Cette optimisation doit répondre à un ensemble de critères spécifiques qui dépendent des besoins de l'usager : la vitesse du code généré, la taille du programme objet, le temps de compilation, etc. Plusieurs optimisations peuvent être réalisées pendant le processus de compilation. Quelques-unes sont indépendantes de la machine et sont réalisées tôt au cours de la compilation. D'autres sont dépendantes de la machine et sont réalisées vers la fin de la compilation. Muchnick [Muc97] présente une liste détaillée des optimisations et du moment où elles sont réalisées.

Parmi les problèmes que le compilateur doit résoudre nous retrouvons la sélection d'instructions, l'allocation de registres et l'ordonnancement d'instructions (*instruction scheduling*). Ils ont un grand impact sur la qualité du code généré.

- La sélection d'instructions : trouver les instructions machine les plus appropriées pour chaque instruction du code intermédiaire.

- L'allocation de registres : assigner les variables présentes dans le code intermédiaire à des registres de la machine.
- L'ordonnancement d'instructions : trouver l'ordre des instructions qui est le plus rapide en respectant la sémantique du programme.

Pour tester notre approche nous utilisons les problèmes d'ordonnancement d'instructions et d'allocation de registres. Ainsi, nous les décrivons de façon plus détaillée et nous expliquons aussi les méthodes couramment utilisées pour leur résolution.

2.2 Le problème d'allocation de registres

Typiquement, la partie frontale d'un compilateur suppose une source inépuisable de variables temporaires. Alors elle génère une représentation intermédiaire comportant un grand nombre de variables. Lors du processus de génération de code, ces variables doivent être assignées à un registre ou à un emplacement mémoire.

Sur les processeurs modernes les instructions dont les opérandes sont dans des registres sont exécutées plus rapidement que celles impliquant un accès à la mémoire. C'est pourquoi une utilisation judicieuse des registres est très importante lors de la génération de code.

Il faut déterminer quelles variables sont stockées dans les registres et identifier les registres auxquels elles sont assignées à chaque point du programme. Cette assignation doit respecter les conflits entre les variables.

Une variable est *vivante* si sa valeur sera utilisée dans le futur et elle est *morte* quand sa valeur n'est plus nécessaire. L'ensemble d'instructions où une variable est vivante s'appelle l'*intervalle de vie* de cette variable. Par exemple, dans le programme de la figure 2.1 la variable a est définie à l'instruction *1* et la dernière fois qu'elle est utilisée est dans l'instruction *12*, alors la variable a est vivante de l'instruction *2* à l'instruction *12*. Un algorithme itératif est utilisé pour obtenir les intervalles de vie de toutes les variables

d'un programme. Cet algorithme est $O(n^4)$ dans le pire cas pour des programmes avec n instructions [App97]. Si les intervalles de vie de deux variables se chevauchent, alors elles sont en conflit et ne peuvent pas être assignées au même registre. Le problème de déterminer l'assignation de variables aux registres en tenant compte de ces conflits est connu comme le problème d'allocation de registres [AU78].

En général, le nombre de variables (les variables définies dans le code source, les variables temporaires générées par le compilateur et dans certains cas les constantes) est beaucoup plus élevé que le nombre de registres. Il y a deux techniques principales pour aborder le problème [AU78, Muc97] :

- Débordement (*spill*) : une variable est assignée à un emplacement en mémoire et chaque référence à cette variable doit être effectuée en utilisant des instructions de chargement et de stockage.
- Division (*split*) : l'intervalle de vie d'une variable est divisé en plusieurs intervalles de vie plus petits. Ceci ne diminue pas la compétition pour les registres mais diminue les conflits entre les variables.

À titre d'exemple, la figure 2.1(a) montre un programme simple dans lequel 13 variables sont utilisées, tandis que la figure 2.1(b) montre les intervalles de vie de chaque variable. La variable a ne peut pas être assignée au même registre que la variable b, puisque l'intersection des deux intervalles est non-vide. De même, la variable d est vivante dans le même intervalle que e, etc.

Si l'allocateur a assez de registres pour réaliser l'assignation (6 pour le programme de la figure 2.1), il ne sera pas nécessaire de réaliser un débordement vers la mémoire et le code généré sera celui de la figure 2.2(a). Par contre, s'il n'a pas assez de registres, l'allocateur doit décider quelles variables sont assignées à des registres et lesquelles sont débordées vers la mémoire. Les figures 2.2(b) et 2.2(c) montrent le résultat de l'allocation si l'allocateur décide de stocker en mémoire les variables a et b respectivement (A et B sont des emplacements en mémoire). Dans chaque cas, la variable stockée en mémoire doit être chargée avant chaque utilisation et stockée après.

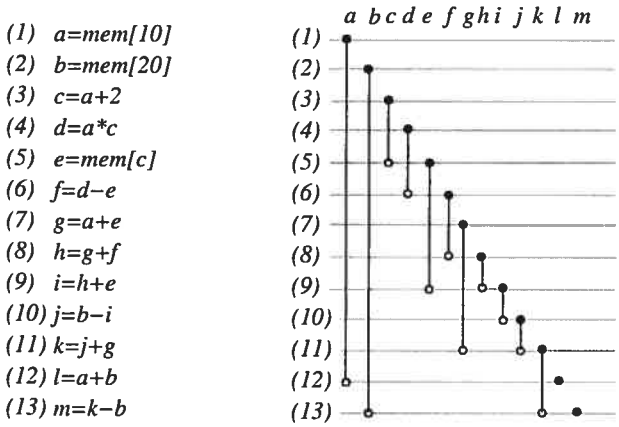


FIG. 2.1 – (a) Exemple de programme. (b) Intervalles de vie.

Dans le premier cas (figure 2.2(b)), il faut faire quatre chargements de la variable, tandis que dans le deuxième (figure 2.2(c)), il faut en faire trois.

En considérant la division d'intervalles de vie, l'allocateur peut décider, par exemple, de stocker en mémoire la variable *b* pendant la première partie du programme et ensuite de stocker la variable *a* en mémoire et assigner *b* au registre *r1*, tel qu'il est montré dans la figure 2.2(d).

Dans la littérature sur le problème d'allocation de registres, on trouve des approches différentes pour résoudre le problème, dont les plus importantes sont les allocateurs par coloriage de graphe et les allocateurs à la volée. On constate aussi que le problème est résolu à divers niveaux : l'allocation intraprocédurale, l'allocation interprocédurale et l'allocation globale.

2.2.1 Les allocateurs par coloriage de graphe

Un allocateur de registres par coloriage [CAC+82] de graphe représente les conflits entre les variables par un graphe d'interférence G(N,A). Les nœuds $n_i \in N$ représentent

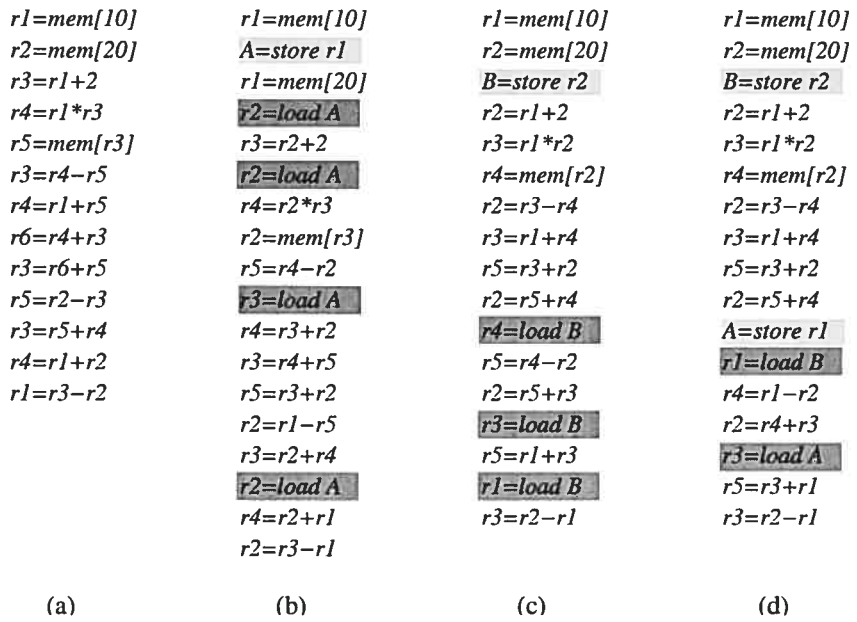


FIG. 2.2 – Programme avec des différentes assignations des variables aux registres

les variables du programme et une arête $(i, j) \in A$ existe si et seulement si le nœud n_i a un conflit avec n_j ; c'est-à-dire qu'il existe un point dans le programme où les deux variables contiennent des valeurs utiles au calcul.

L'allocateur essaie de trouver un k-coloriage pour le graphe, où k est le nombre de registres de la machine. Si l'allocateur ne peut pas trouver un coloriage, il doit choisir un ou plusieurs nœuds à stocker en mémoire (débordement).

Chaitin et al. [CAC+82] ont été les premiers à concevoir et implanter un allocateur par coloriage de graphe. Cet allocateur prend la décision de faire déborder une variable sur la base du nombre de références et du nombre de conflits (le nombre des arêtes dans le graphe d'interférence) de la variable.

L'allocateur de registres de Chaitin a servi comme base à un grand nombre d'allocateurs (allocateurs de type Chaitin). Nous décrivons ici les cinq phases principales de ce type d'allocateur. Ces phases sont illustrées à la figure 2.3.

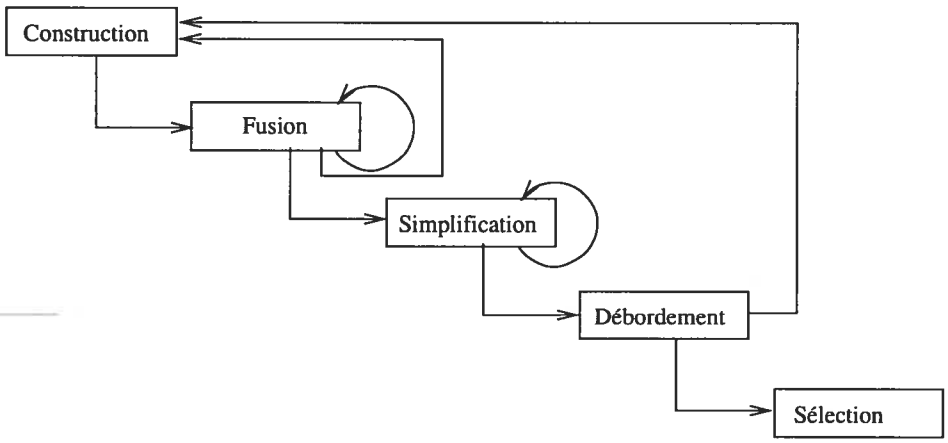


FIG. 2.3 – Allocateur de type Chaitin

1. **Construction** (*build*) : construire le graphe d'interférence à partir du programme.
2. **Fusion** (*coalesce*) : éliminer les instructions de déplacement (*move*) inutiles. S'il n'existe pas d'arête d'interférence entre la source v_1 et la destination v_2 d'une instruction de déplacement ($v_2 = v_1$), les nœuds v_1 et v_2 peuvent être combinés en un seul nœud qui contient l'union des arêtes d'interférence de v_1 et v_2. Après avoir réalisé toutes les fusions possibles, il faut reconstruire le graphe d'interférence et répéter le processus.
3. **Simplification** (*simplify*) : colorier le graphe en utilisant une heuristique simple. Les nœuds de degré inférieur à k (nombre de couleurs) sont retirés du graphe, un à un, et sont mis dans une pile. En effect, ces nœuds peuvent toujours être coloriés facilement si on a k couleurs.
4. **Débordement** (*spill*) : s'il n'y a pas un nœud de degré inférieur à k, l'heuristique de simplification échoue, alors il faut choisir un nœud à faire déborder vers la mémoire. Le nœud choisi est éliminé du graphe et le programme est récrit en incorporant les instructions de chargement (*load*) et de stockage (*store*) nécessaires. Il est à noter que le fait d'ajouter des opérations de chargement et de stockage introduit aussi des nouvelles variables temporaires. Le processus au complet est répété jusqu'à ce qu'aucun nœud ne se fasse déborder.

Une heuristique est utilisée pour choisir la variable qui sera stockée en mémoire. Cette heuristique essaie de quantifier deux aspects :

a) Le coût d'envoyer cette variable vers la mémoire. Ce coût est proportionnel au nombre de fois que des instructions de chargement et de stockage sont exécutées. Le compilateur approximise ce coût en tenant compte du nombre de références à la variable ainsi que de la profondeur de la boucle où ces références se trouvent.

b) Le bénéfice d'éliminer cette variable du graphe d'interférence.

La priorité de débordement est donnée par :

$$(u_i + \sum_b pboucle_i(b) * uboucle_i(b))/d_i$$

où :

u_i : nombre de références à la variable i en dehors des boucles ;

$uboucle_i$: nombre de références à la variable i dans une boucle de profondeur $pboucle_i$;

d_i : le degré du nœud dans le graphe d'interférence.

Si la profondeur de la boucle n'est pas connue, une estimation ou une valeur constante est utilisée.

5. **Sélection** : assigner une couleur aux nœuds du graphe. Prendre, successivement, un nœud dans la pile et lui assigner une couleur légale (différente de la couleur des nœuds avec lesquels il entre en conflit, ce qui est toujours possible).

Briggs et al. [BCT92, BCT94, BCKT89] proposent quelques modifications qui provoquent une diminution importante du nombre de variables qui doivent se faire déborder (figure 2.4) :

- **Fusion conservatrice** : la fusion (à l'etape 2) n'est réalisée que si elle ne transforme pas le graphe en un graphe non-coloriable. Soient n_i et n_j des nœuds candidats à la fusion. Si le nouveau nœud n_p est de degré plus grand que k (nombre de couleurs), la fusion est interdite.

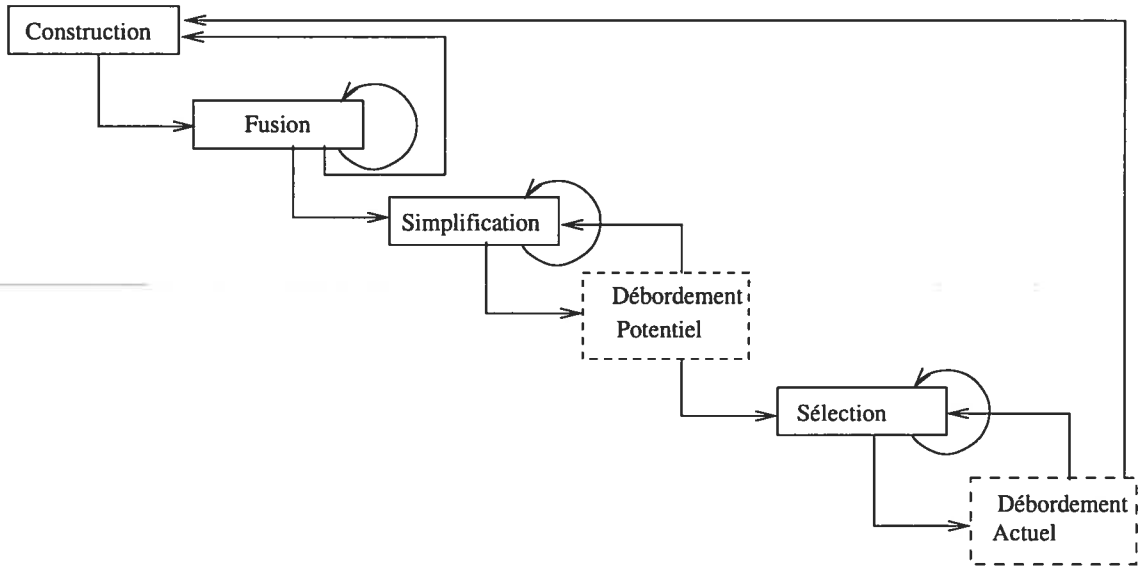


FIG. 2.4 – Allocateur proposé par Briggs et al.

- **Coloriage optimiste** : le débordement d'une variable est retardé jusqu'à la phase de sélection. Durant la phase de simplification, la variable sélectionnée est marquée comme un débordement potentiel et est mise dans la pile. Dans la phase de sélection s'il n'est pas possible de trouver une couleur pour cette variable, alors elle se fait vraiment déborder.
- **Rematérialisation** : dans les cas où la variable à déborder est une constante ou une valeur facile à calculer à partir d'une autre (par exemple $v_1 = v_2 + 1$), ce n'est pas nécessaire de la garder en mémoire. Alors, il est possible (et moins cher) de la recharger ou de la recalculer.

L'allocateur de George et Appel [GA96] utilise le coloriage optimiste et la fusion conservatrice proposée par Briggs. Par contre, il applique les étapes de fusion et de simplification répétitivement (figure 2.5). Il réussit à diminuer de façon considérable le nombre d'instructions de déplacement, en obtenant ainsi une amélioration moyenne du temps d'exécution du code généré de 5%. Les étapes de cet allocateur sont :

1. **Construction** : construire le graphe d'interférence et étiqueter les nœuds comme

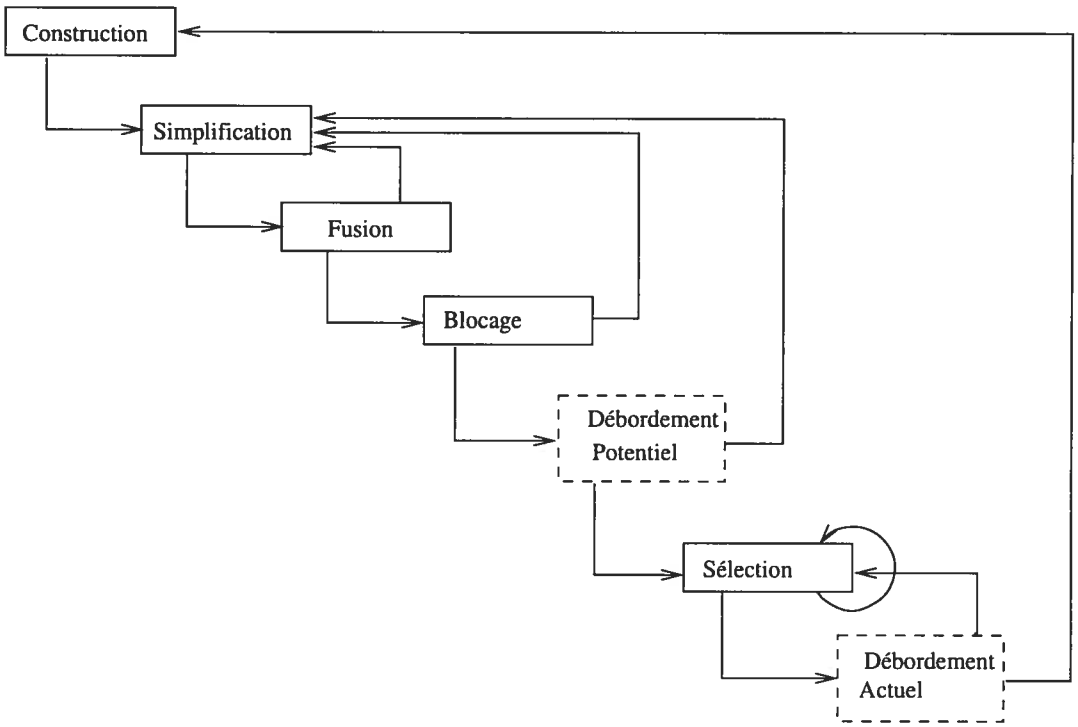


FIG. 2.5 – Allocateur proposé par Appel

étant *move-related* ou *non-move-related*, où un nœud *move-related* est un nœud qui est la source ou la destination d'une instruction de déplacement.

2. **Simplification** : sélectionner successivement les nœuds *non-move-related* de degré inférieur à k (nombre de couleurs) à partir du graphe et les mettre dans une pile.
3. **Fusion** : réaliser la fusion conservatrice.
4. **Blocage** : quand les étapes de simplification et de fusion ne peuvent pas être appliquées, un nœud *move-related* est sélectionné pour le débordement ; ceci fait en sorte que des nœuds *move-related* se transforment en *non-move-related* et alors les étapes de simplification et de fusion peuvent être exécutées à nouveau.
5. **Sélection** : assigner la couleur aux nœuds du graphe. Prendre chaque nœud de la pile successivement et lui assigner une couleur faisable.

D'autres travaux, comme ceux de Bernstein et al. [BGG+89] et Lueh [Lue96], utilisent

des heuristiques différentes pour choisir les nœuds au cours de l'étape de débordement.

Dans l'allocateur original de Chaitin, les instructions de stockage sont placées après chaque définition de la variable (une variable est *définie* dans l'instruction qui utilise la variable comme destination) et les instructions de chargement, avant chaque utilisation (une variable est *utilisée* dans les instructions où la variable est utilisée comme source). Bergner et al. [BDEO97], Cooper et Simpson [CS98] et Lueh et al. [LGAT96] ont développé des heuristiques pour mieux placer ces instructions, de façon à diminuer les débordements et à ne pas limiter le parallélisme au niveau des instructions.

Callahan et Koblenz [CK91] proposent un allocateur par coloriage de graphe qui réalise le coloriage pour des parties du programme en considérant sa structure de flux. Cet allocateur permet de placer les débordements vers la mémoire dans les portions moins exécutées du programme.

Chow et Hennessy [CH90] présentent un allocateur par coloriage de graphe qui assigne les registres en fonction d'une priorité basée sur le nombre de fois que la variable est référencée, la fréquence d'exécution du bloc de base où la variable est utilisée et la taille de l'intervalle de vie de la variable. Ils décrivent, aussi, différentes modifications qui peuvent être apportées aux allocateurs par coloriage de graphe.

2.2.2 Les allocateurs à la volée

Ces méthodes cherchent un compromis entre la qualité de l'allocation et le temps de compilation. Ils sont intéressants lorsque le temps de compilation est un facteur important. Par exemple, dans les systèmes de compilation dynamique.

Tout comme les allocateurs par coloriage de graphe, les allocateurs à la volée utilisent l'information des intervalles de vie pour guider l'allocation. L'allocateur parcourt les intervalles de vie en suivant l'ordre dans lequel ceux-ci apparaissent dans le code, et il considère le nombre d'intervalles qui sont vivants à chaque point du code. S'il y a

plus d'intervalles de vie que de registres, l'allocateur décide de façon heuristique quelles variables se font déborder.

Traub et al. [THS98] et Poletto et Sarkar [PS99] proposent et évaluent des allocateurs à la volée. Dans leurs travaux, ils ont observé des résultats semblables entre l'allocateur de ce type et ceux par coloriage de graphe en ce qui concerne le temps d'exécution du code généré ; mais l'allocateur par coloriage de graphe utilise plus de temps de compilation.

2.2.3 L'allocation intraprocédurale

Pour simplifier le problème, l'allocateur de registres analyse seulement une fonction du programme à la fois. Il ne considère pas les fonctions qui sont appelées. L'allocation intraprocédurale est en général NP-dur [FL98, LFK97]. Dans le cas spécial où l'expression est un arbre (il n'y a pas de boucles) et tous les registres sont équivalents, le problème peut être résolu de façon optimale en temps O(n) avec l'algorithme de Sethi-Ullman [AS87]. Les méthodes les plus utilisées pour aborder ce problème sont les allocateurs par coloriage de graphe et ceux à la volée.

2.2.4 L'allocation interprocédurale

Étant donné les besoins en registres de chaque procédure du programme, l'allocateur sélectionne les registres disponibles pour chaque procédure et génère les débordements nécessaires au cours des appels. L'allocateur essaie de mettre les débordements dans les appels les moins fréquents. Certains allocateurs par coloriage de graphe ont été adaptés au problème interprocédural [Muc97, CH90].

Kurlander et Fischer [KF96] considèrent que le coût de débordement est fonction de la fréquence des appels. Ils modélisent le problème comme le dual du problème de flux de coût mininal et ils proposent un algorithme linéaire pour le résoudre.

Une approche différente utilisée par Wall [Wal86] réalise l'allocation interprocédurale

à l'édition de liens. Cet allocateur utilise une heuristique basée sur le fait que deux procédures qui ne sont pas actives au même moment peuvent utiliser les mêmes registres.

2.2.5 L'allocation globale

L'allocateur analyse plusieurs procédures simultanément et détermine une allocation pour les variables. Ceci peut être fait avec un allocateur par coloriage de graphe ; c'est-à-dire, le graphe d'interférence pour plusieurs procédures est construit et ensuite un allocateur par coloriage de graphe est appliqué. Malheureusement, si le graphe d'interférence est trop grand l'allocateur de registres n'aura pas de bonnes performances, soit par la qualité de l'assignation, soit par le temps ou mémoire requis lors de la compilation.

Lueh et al. [LGAT00] proposent un allocateur par coloriage de graphe qui construit le graphe d'interférence de façon incrémentale.

Des allocateurs à la volée sont aussi utilisés pour réaliser une allocation globale.

2.3 Le problème d'ordonnancement d'instructions

Les processeurs actuels possèdent plusieurs unités fonctionnelles qui peuvent travailler en parallèle (typiquement une ou plusieurs unités de : traitement des entiers, traitement de nombres flottants, accès de mémoire, etc.). Cette caractéristique permet d'exploiter le parallélisme implicite que contiennent la plupart des programmes. Cette caractéristique s'appelle le parallélisme au niveau des instructions (*instructions level parallelism*, ou ILP) [HP96].

Pour que deux instructions d'un programme puissent être exécutées en même temps, elles doivent être indépendantes. Par exemple, dans le code de la figure 2.6(a) les instructions 1, 2 et 3 sont dépendantes car elles ont une dépendance de données. Alors elles doivent être exécutées dans cet ordre sinon le résultat obtenu ne sera pas celui désiré.

Par contre, l'instruction 3 est indépendante de l'instruction 6, alors elles peuvent être exécutées dans n'importe quel ordre et même en parallèle. Ces dépendances peuvent être représentées par un graphe, tel que montré à la figure 2.6(b). Même si les instructions 3 et 6 sont indépendantes, il faut qu'il existe assez d'unités fonctionnelles dans le *pipeline* pour qu'elles puissent être réellement exécutées en parallèle. C'est-à-dire, comme les deux instructions sont du type arithmétique entière, elles ont besoin de l'unité des entiers pour être exécutées. Alors, pour qu'elles soient exécutées en parallèle la machine doit au moins avoir deux unités de traitement des entiers.

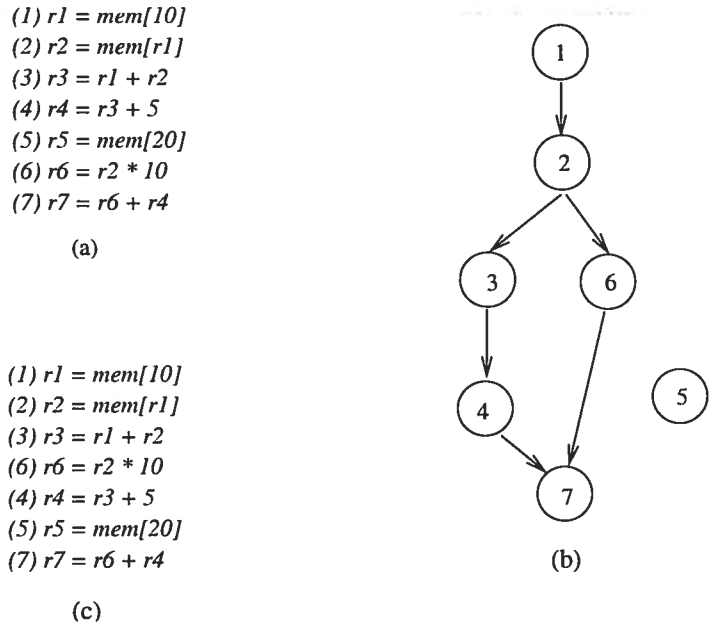


FIG. 2.6 – (a) Exemple de programme . (b) Graphe de dépendance. (c) Programme avec ordonnancement.

Pour profiter de l'ILP le code doit être réécrit de façon à ce que les instructions soient générées dans un ordre qui permet au processeur de plus facilement reconnaître que ces instructions sont indépendantes et exécutables en parallèle. En général, cela veut dire qu'il faut grouper les instructions exécutables en parallèle. Dans notre exemple, si nous voulons que les instructions 3 et 6 soient exécutées en parallèle, il faut réécrire le code comme le montre la figure 2.6(c). La problématique de trouver le meilleur ordre d'exécu-

tion des instructions s'appelle l'ordonnancement d'instructions (*instruction scheduling*).

Le problème d'ordonnancement d'instructions est NP-dur [App97]. Actuellement, ils existe différentes approches pour aborder le problème dont les principales sont : l'ordonnancement statique, l'ordonnancement dynamique, l'ordonnancement de boucles et l'ordonnancement des blocs de base.

2.3.1 Ordonnancement statique et ordonnancement dynamique

Plusieurs architectures actuelles (le Pentium, par exemple) sont capables de réaliser l'ordonnancement d'instructions au cours de l'exécution (ordonnancement dynamique) en lisant à l'avance plusieurs instructions. L'avantage de cette approche est qu'elle peut profiter des caractéristiques connues seulement à l'exécution, tels que les exceptions de cache (*cache misses*), et ainsi obtenir un meilleur ordonnancement. Par contre, le désavantage est que l'ordonnancement doit être fait en temps réel, donc peu d'instructions sont considérées à la fois, ce qui donne une optimisation locale. En plus, inclure le matériel pour faire l'ordonnancement augmente normalement la période du cycle d'horloge de la machine. Des techniques comme le Scoreboarding [HP96] et l'algorithme de Tomasulo [Tom67] sont utilisées pour implanter ce mécanisme.

L'ordonnancement statique (ordonnancement réalisé par le compilateur) n'a pas à résoudre le problème en temps réel et, ainsi, il peut faire une analyse plus globale pour produire une meilleure solution. Par contre, il ne peut pas prévoir le comportement dynamique du programme. Jusqu'à présent, ce n'est pas clair laquelle des deux approches est la plus performante. Une bonne comparaison entre les deux approches pour des machines RISC se trouve dans [LPU95]. Il reste qu'un ordonnancement statique peut aider l'ordonnancement dynamique du processeur.

2.3.2 Ordonnancement de boucles et ordonnancement des blocs de base

L'ordonnancement de boucles (*loop scheduling*) consiste à reordonner les instructions dans le corp d'une boucle. Conceptuellement, le compilateur déroule les boucles et il essaie d'exploiter le parallélisme. L'algorithme proposé par Rau [Rau94] et appelé *modulo scheduling* est le plus utilisé pour résoudre ce problème. Différentes modifications ont été apportées dans [Huf93] et dans [ZLAV01]. Dans [CLG02], une comparaison de différentes techniques de *modulo scheduling* est présentée.

Comme son nom l'indique, l'ordonnancement des blocs de base (*basic block scheduling*) consiste à réordonner les instructions dans chaque bloc de base du programme. Étant donné qu'il n'y a pas de boucles, les dépendances entre variables peuvent être représentées par un DAG (*Directed Acyclic Graph*). L'algorithme le plus utilisé pour résoudre ce problème est l'ordonnancement par liste (*list scheduling*) qui a été proposé par Gibbons et Muchnick [GM86]. Cet algorithme itère en assignant une instruction à la fois jusqu'à ce que toutes les instructions du bloc de base soient assignées. À chaque pas, l'algorithme utilise une heuristique pour choisir une instruction entre toutes les instructions qui sont prêtes à être exécutées (celles où toutes les instructions dépendantes précédentes sont déjà exécutées). Pour déterminer quelles instructions sont prêtes à être exécutées, il est nécessaire d'estimer le temps que prend l'exécution de chaque instruction. Pour faire ceci, il faut considérer la latence des instructions, les dépendances entre instructions et les caractéristiques du pipeline de la machine. Plus cette estimation est exacte, plus le résultat de l'ordonnancement par liste est bon.

D'autres modifications ont été apportées par la suite, dont la plupart sont des fonctions heuristiques. Cooper et al. [CSS98] font une évaluation empirique d'ordonnancement par liste. La figure 2.7 montre le pseudocode pour un ordonnancement par liste.

D'autre part, Wilken et al. [WLH00] présentent une approche basée sur la programmation entière pour résoudre le problème d'ordonnancement dans un bloc de base ; ils

```
ListScheduling {

        DAG = le graphe de précédence du bloc de base
        Inst   = ensemble de tous les nœuds ∈ DAG
        Prêtes  = ensemble des feuilles ∈ DAG
        Sch    = ∅

        Pour chaque nœud n ∈ DAG
                Assigner priorité à n en utilisant une heuristique

        Tant que Inst ≠ ∅ {
                n = élément de Prêtes avec la plus haute priorité
                Prêtes = Prêtes \{n}
                Sch    = Sch ∪ {n}
                Inst   = Inst \ {n}

                Pour chaque père p de n dans DAG
                        Si tous les enfants de p ∈ Sch
                                Prêtes = Prêtes ∪{p}
        }
}
```

FIG. 2.7 – Pseudocode de l'ordonnancement par liste

comparent leur approche avec GNU C et obtiennent une amélioration de 14%.

2.3.3 L'allocation et l'ordonnancement

Les problèmes d'allocation de registres et d'ordonnancement sont fortement interdépendants [GH88]. D'une part, puisque l'ordonnancement d'instructions détermine les intervalles de vie des variables, alors si le compilateur fait l'ordonnancement avant de faire l'allocation, le graphe d'interférence sera probablement modifié, ce qui pourrait mener à une allocation de registres différente. De plus, en faisant l'allocation de registres, le compilateur définit des dépendances entre des instructions qui n'existaient pas dans le programme original (dépendances de sortie et anti-dépendances) puisqu'il assigne le même registre à des variables différentes et introduit de nouvelles instructions (de chargement et de stockage). D'autre part, si le compilateur fait l'ordonnancement après l'allocation, le graphe qui représente les dépendances entre les instructions dépend de l'allocation choisie. Or, ce n'est pas clair laquelle des optimisations devrait être fait en premier. Une approche très répandue consiste à faire un premier ordonnance-

ment d'instructions (*pre-pass scheduling*) avant l'allocation de registres et un deuxième ordonnancement d'instructions après (*post-pass scheduling*).

Quelques recherches essaient de résoudre les problèmes d'allocation de registres et d'ordonnancement d'instructions en même temps. Plusieurs d'entre elles travaillent au niveau des boucles en utilisant des algorithmes comme le *modulo scheduling*, parmi lesquelles nous trouvons :

- Zalamea et al. [ZLAV01] proposent un algorithme itératif pour résoudre les problèmes d'ordonnancement d'instructions et d'allocation de registres dans les boucles. Leur algorithme est un *modulo scheduling* qui intègre une heuristique pour faire l'allocation de registres.
- Eichenberger et al. [EDA96] proposent une méthode pour déterminer les besoins optimaux des registres dans une boucle. Ils incorporent cette méthode dans un algorithme de *modulo scheduling* et ils obtiennent une diminution de jusqu'à 24% sur les besoins des registres.
- Ning et Gao [NG93] définissent une structure pour résoudre en même temps l'allocation de registres et l'ordonnancement d'instructions. Ils formulent le problème de l'ordonnancement optimal dans une boucle et d'assignation des registres symboliques sous forme d'un problème de programmation entière qu'ils résolvent en le transformant en un problème de flux de coût minimal. Après, ils utilisent un algorithme de coloriage pour assigner des registres réels aux registres symboliques.
- Finalement, Gopal et Govindrajan [VG99] présentent une évaluation de différentes techniques pour l'ordonnancement d'instructions et l'allocation de registres pour des machines effectuant l'ordonnancement dynamiquement.

Il y a aussi quelques travaux qui traitent ces problèmes au niveau des blocs de base.

- Berson et al. [BGS98] proposent différentes façons d'intégrer l'allocation et l'ordonnancement et font une analyse comparative entre elles.
- Norris et Pollock [NP95] décrivent des stratégies pour aider à la coopération entre

l'ordonnancement et l'allocation. Ils proposent une première technique pour l'ordonnancement qui tient compte de l'utilisation des registres pour décider de modifier ou non l'ordonnancement courant. Ils en proposent une deuxième qui fait l'allocation des registres en utilisant un allocateur par coloriage de graphes où ils modifient le graphe d'interférence en ajoutant des arêtes en fonction de l'ordonnancement d'instructions.

- Goodman et Hsu [GH88] présentent une discussion sur l'interdépendance des problèmes d'allocation et d'ordonnancement et proposent deux méthodes pour aborder les deux problèmes simultanément. La première intègre dans le même pas une technique qui fait l'ordonnancement en minimisant le nombre de délais dans le pipeline avec une autre qui minimise l'utilisation des registres. La deuxième méthode utilise un graphe de dépendances (contrairement à l'approche typique qui utilise un graphe d'interférence) pour faire l'allocation de registres. Moins d'interférences sont ajoutées avec cette représentation.
- Finalement, Kurlander et al. [KPF95] proposent un algorithme optimal pour faire l'ordonnancement et l'allocation quand l'expression est un arbre et présentent une extension pour le cas général.

2.4 Les méthodes métaheuristiques

Les méthodes métaheuristiques ont été largement utilisées au cours des deux dernières décennies pour la résolution de problèmes d'optimisation combinatoire. Ces méthodes n'assurent pas l'obtention d'une solution optimale, mais elles cherchent à atteindre un point d'équilibre entre la qualité de la solution et le temps de calcul. Il y a peu de résultats théoriques qui garantissent la convergence des méthodes mais les résultats empiriques nous permettent de leur faire confiance. Dans cette section nous décrivons sommairement les méthodes métaheuristiques utilisées dans cette thèse : algorithmes génétiques et recuit simulé.

2.4.1 Algorithmes génétiques

Les algorithmes génétiques (AGs) sont des méthodes de recherche qui se basent sur le processus d'évolution naturelle [Hol75, Gol89, Dav91]. Pour résoudre un problème avec un AG, il faut trouver une représentation pour les solutions (individus) de telle sorte qu'elle soit capable d'encoder toutes les solutions possibles au problème. Chaque solution a une valeur numérique associée qui représente une mesure de sa qualité. Cette valeur est donnée par une fonction de coût spécifique à chaque problème.

L'AG commence avec un ensemble de solutions aléatoires (population) et les fait évoluer (d'une génération à l'autre) avec l'espoir qu'à la fin du processus d'évolution la qualité de la population se soit améliorée. L'évolution est réalisée en utilisant des opérateurs génétiques, dont les plus importants sont :

- **Sélection** : cet opérateur sélectionne les individus qui se reproduisent en fonction de leur capacité à s'adapter au milieu ; les individus de meilleure qualité ont une plus grande probabilité de se reproduire et, ainsi, de transmettre leurs caractéristiques aux générations suivantes. L'opérateur de sélection proposé originalement [Hol75] est la roulette pondérée ou sélection proportionnelle (*roulette-wheel selection*). Cet opérateur est utilisé pour choisir aléatoirement les individus qui se reproduisent, en donnant à chaque individu une probabilité en fonction de sa qualité.
- **Croisement** : cet opérateur simule le processus de reproduction par l'intermédiaire duquel les caractéristiques des parents sont transmises aux enfants. L'opérateur de croisement doit combiner d'une certaine façon les individus parents sélectionnés pour générer les enfants. Le *croisement à un point (one point crosover)* choisit une position k au hasard, où $1< k <$L (où L est la taille de l'individu), et il génère deux enfants, en prenant les valeurs dans les positions 1 à k d'un parent et les valeur dans les positions k+1 à L de l'autre, comme le montre la figure 2.8.
- **Mutation** : cet opérateur apporte une variabilité génétique en changeant aléa-

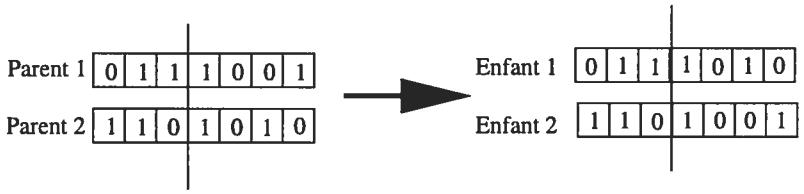


FIG. 2.8 – Opérateur de croisement classique

toirement une petite partie de l'individu. Une mutation est appliquée à chaque position de l'individu avec une très faible probabilité. Si la position doit muter, la valeur à cette position est changée (figure 2.9).

FIG. 2.9 – Opérateur de mutation classique.

Si l'encodage de la solution est une chaîne binaire et les opérateurs d'origine (décrits ci-dessus) sont utilisés, le théorème du schème garantit [Hol75] la convergence de l'AG. Malheureusement, pour résoudre plusieurs problèmes, il faut utiliser des représentations plus élaborées ; par exemple, des solutions dans lesquelles chaque élément est un nombre ou un symbole [Dav91]. Dans ces cas, il est nécessaire de trouver des opérateurs génétiques spécifiques au problème.

Les AGs utilisent un ensemble de paramètres qui doivent être ajustés. La valeur de ces paramètres dépend du problème qui sera résolu et de la taille de l'instance. Le bon ajustement de ces paramètres est crucial dans l'obtention d'une bonne performance, soit au niveau de la qualité de la solution, soit au niveau du temps de traitement. Les principaux paramètres sont :

- **La taille de la population** (TP) : le nombre d'individus de la population initiale. Généralement, ce nombre reste constant pendant tout le processus d'évolution. Une population de plus grande taille aura un temps de convergence plus lent (temps de traitement plus élevé) et une population de plus petite taille court le risque de

tomber dans un optimum local.

- **Le nombre de générations** : le nombre de générations utilisées par l'AG. Plus le nombre de générations est grand, plus l'AG aura la possibilité d'améliorer la qualité de sa population. Cependant, un grand nombre de générations peut être inutile puisque lorsque l'AG converge (tous les individus de la population sont semblables), même si l'AG se poursuit, il n'y aura pas d'amélioration de la qualité et il y aura un gaspillage important de temps.
- **La probabilité de mutation** : la probabilité qui sert à déterminer si un nouvel individu doit être muté. La mutation est le mécanisme qui évite à l'AG de tomber dans des optimums locaux. Ainsi, une probabilité de mutation trop petite n'atteint pas l'objectif. Par contre, une probabilité trop haute perturbe le processus d'évolution et entraine l'AG dans un comportement aléatoire (plus qu'évolutif).

La figure 2.10 montre le pseudocode d'un AG :

```
AlgorithmeGénétique {

    P_0 = Population initial aléatoire
    Pour chaque individu i ∈ P_0
        Évaluer individu i

    gen = 0
    Tant que critère d'arrêt == faux {
        gen = gen +1
        P_gen = Appliquer l'opérateur de croisement sur P_gen-1

        Pour chaque individu i ∈ P_gen {
            Appliquer l'opérateur de mutation à l'individu i
            Évaluer individu i
        }
    }
}
```

FIG. 2.10 – Pseudocode d'un algorithme génétique

Dans un AG, les seuls éléments qui dépendent du problème sont la représentation et la fonction de coût. Tout le processus d'évolution est indépendant du problème qui est résolu. Cependant, dans certains cas il faut incorporer à l'AG de l'information sur le problème pour obtenir de bonnes performances. Voici les deux modifications les plus

importantes et les plus utilisées [Dav91] :

- Définir des opérateurs génétiques spécifiques au problème : les opérateurs génétiques (surtout le croisement) peuvent être redéfinis en considérant le sens de l'individu. L'objectif est d'assurer que les bonnes caractéristiques des parents seront maintenues chez les enfants. La définition de bonne caractéristique dépend du problème.
- Population initiale : au lieu de commencer avec une population aléatoire, l'AG commence avec une population de meilleure qualité (par exemple, une population générée par une heuristique). Ceci peut accélérer la convergence de l'algorithme, en obtenant ainsi un temps de traitement plus court, mais il y a un risque de forcer l'algorithme à une convergence prématurée vers un optimum local.
- Stocker le meilleur individu : d'une itération (génération) à l'autre, nous conservons l'individu de meilleure qualité généré jusqu'ici. À la fin de l'algorithme, ce dernier correspond à la solution obtenue.
- Élitisme : l'utilisation d'élitisme vise à favoriser que les meilleurs individus survivront d'une génération à l'autre. Pour ce faire, le ou les individus de meilleure qualité sont tout simplement copiés à la nouvelle population, ou bien, un pourcentage d'individus (**pourcentage de sélection**) de la population sont choisis par l'opérateur de sélection pour passer directement à la nouvelle génération. L'utilisation d'élitisme permet d'améliorer la qualité de la population, mais elle peut force à une convergence prématurée vers un optimum local.

Dans la littérature, nous trouvons beaucoup d'opérateurs génétiques conçus pour des problèmes particuliers. Cependant il y a deux opérateurs de croisement qui sont très souvent utilisés pour des problèmes de permutations comme le coloriage de graphe et l'ordonnancement, et que nous décrivons ci-dessous [Dav91] :

- **Le croisement basé sur l'ordre** (OBX) : L'idée de cet opérateur est de maintenir dans l'enfant l'ordre relatif de quelques valeurs du parent. Un sous-ensemble des positions est sélectionné chez le premier parent, les éléments provenant de ces

positions sont copiés chez l'enfant aux positions où ces éléments apparaissent chez le deuxième parent. Les positions restantes sont remplies avec les éléments du deuxième parent qui ne sont pas déjà chez l'enfant en le parcourant à partir de la première position (figure 2.11). La taille du sous-ensemble des positions est un paramètre de l'AG. Les valeurs les plus souvent utilisées varient entre 25% et 30% de la taille du chromosome.

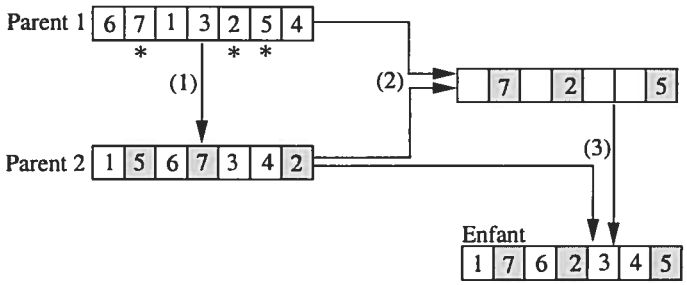


FIG. 2.11 – Opérateur OBX

- **Le croisement basé sur la position** (PBX) : L'idée de cet opérateur est de maintenir dans l'enfant la position de quelques valeurs du parent. Un sous-ensemble des positions est sélectionné chez le premier parent, les éléments provenant de ces positions sont copiés chez l'enfant aux mêmes positions. Les positions restantes sont remplies avec les éléments du deuxième parent qui ne sont pas déjà chez l'enfant en le parcourant à partir de la première position (figure 2.12).

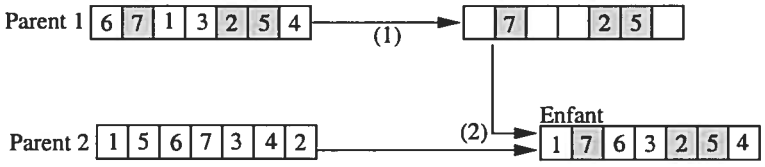


FIG. 2.12 – Opérateur PBX

2.4.2 Recuit simulé

La méthode du recuit simulé (RS) (*simulated annealing*) [KGV83] est une méthode de recherche locale basée sur le procédé de trempe des substances pures. À partir de la

solution courante, l'algorithme génère successivement une solution voisine en espérant que celle-ci soit de meilleure qualité. Si c'est le cas, l'algorithme adopte la solution voisine comme solution courante et continue le processus. Sinon, il utilise une fonction de probabilité pour déterminer s'il accepte ou non la nouvelle solution même si la qualité se détériore ; ce mécanisme permets au RS de s'échapper des optimums locaux.

La solution voisine est générée en utilisant un opérateur qui dépend du problème. Généralement, l'opérateur modifie aléatoirement la valeur de quelques-unes des variables dans la solution courante.

Lorsque le processus de recherche commence, l'algorithme accepte des détériorations importantes de la qualité par rapport à la solution courante. Au fur et à mesure que le processus avance le niveau de détérioration tolérée devient de plus en plus petit. Ainsi, la probabilité d'accepter une solution de moindre qualité que la solution courante diminue. La *température* (un des paramètres du RS) simule ce comportement tel que décrit dans l'algorithme de la figure 2.13.

```
RecuitSimulé {
    Temp = température initiale
    s = solution initiale
    Tant que Temp ≥ TempératureFinale {
        Répéter rep fois {
            s' = solution voisine de s
            d = f(s') − f(s)
            Si d ≤ 0
                s = s'
            sinon
                Si NombreAléatoire(0,1) ≤ e^(−d/Temp) {
                    s = s'
        Temp = Temp * FacteurRefroidissement
    }
}
```

FIG. 2.13 – Pseudocode pour un recuit simulé

Le RS, comme l'AG, nécessite l'ajustement de quelques paramètres :

- **Température initiale** : la température à laquelle le processus commence. Une

température plus haute permet d'accepter plus facilement des solutions voisines de moindre qualité.

- **Température finale** : la température à laquelle l'algorithme s'arrête.
- **Facteur de refroidissement** : ce nombre définit la vitesse à laquelle la température descend. Une valeur entre 0.7 et 0.99 est généralement utilisée.
- **Le nombre de répétition de la boucle interne** (*rep*) : le nombre de fois que la boucle interne s'exécute à la même température.

2.4.3 Le coloriage de graphe avec des métaheuristiques

Les méthodes métaheuristiques ont été utilisées pour résoudre le problème de coloriage de graphes.

- Eiben et van der Hauw [EvdH96] considèrent l'application de différents AGs au problème de 3-coloriage. Ils testent différentes représentations, opérateurs génétiques et paramètres pour le problème et les comparent avec les approches classiques de coloriage de graphe.
- Fleurent et Ferland [FF97] proposent différentes approches pour adapter les AGs au problème de coloriage de graphe. Ils présentent des algorithmes génétiques hybrides et un algorithme de recherche tabou pour le problème. Ils démontrent que pour des problèmes de grande taille la recherche tabou est plus rapide que les approches basées sur AGs.
- Hertz et Werra [CHdW87] et Chams et al. [CHdW87] résolvent le problème de coloriage avec une recherche tabou et un recuit simulé, respectivement, en utilisant un opérateur pour définir le voisinage très simple et une fonction de pénalité pour traiter les solutions irréalisables.
- Johnson et al. [JAMS91] utilisent un recuit simulé afin de résoudre le problème. Ils présentent une analyse comparative de trois opérateurs pour définir le voisinage.
- Galinier et Hao [GH99] présentent un algorithme hybride (AG et recherche locale) et des opérateurs de croisement spécifiques pour le problème de coloriage. Ils ob-

tiennent une performance supérieure aux algorithmes traditionnels. Ils réussissent à améliorer le meilleur résultat connu pour quelques graphes typiques.

2.4.4 L'ordonnancement avec les métaheuristiques

Les AGs et, en général les méthodes métaheuristiques, ont été utilisées avec beaucoup de succès sur différentes instances du problème d'ordonnancement. Par exemple, sur le *job-shop* [Pin95] nous trouvons des travaux intéressants dans [VAL94, FRC93], sur l'ordonnancement sur multiprocesseurs, dans [CW94, SdRS97] et sur l'ordonnancement d'opérations pour la synthèse de circuits à partir d'une spécification comportementale dans [SBS94]. Cependant, à notre connaissance, les seuls travaux sur l'ordonnancement d'instructions sont présentés par Beaty [Bea91, Bea93]. Il utilise un algorithme d'ordonnancement par liste où la priorité de chaque instruction est trouvée par une métaheuristique (AG et méthode de recherche tabou). Dans [Bea91] il propose une approche pour réaliser l'ordonnancement à l'aide d'un algorithme génétique mais il ne présente pas de résultats. Dans [Bea93], il évalue différents opérateurs génétiques et de génération de voisinage. Ces résultats montrent que les deux méthodes permettent de trouver des bonnes solutions pour le problème d'ordonnancement d'instructions mais il ne présente que des résultats pour un bloc de base en particulier.

A notre avis, l'idée est bonne et ressemble aux approches hybrides qui utilisent ensemble les méthodes métaheuristiques et les heuristiques (comme nous le ferons pour résoudre le problème d'allocation de registres dans le chapitre 4). Par contre, le manque de détails dans la définition de ses approches et le manque de résultats numériques, font qu'il n'est pas possible de tirer des conclusions sur la performance réelle de cette approche.

2.4.5 Les métaheuristiques et la compilation

Bien que les méthodes métaheuristiques aient été largement utilisées pour résoudre des problèmes NP-durs, leur utilisation en compilation est rare. Nous trouvons quelques travaux sur l'ordonnancement pour multiprocesseur dans [CW94, SdRS97], mais ces derniers abordent le problème de façon abstraite, et loin de tout compilateur réel. Les seuls travaux que nous connaissons qui soient vraiment arrivés à utiliser ces méthodes dans la compilation, sont ceux présentés par Williams [WW96] et Nisbet [Nis98]. Ils utilisent des AGs pour effectuer la parallélisation automatique des boucles dans un compilateur pour un langage parallèle.

Ces deux recherches utilisent les AGs afin de trouver le meilleur ordre d'application des différentes optimisations pour paralléliser les boucles d'un programme. Notre recherche est différente de ces derniers sur plusieurs aspects :

- nous ne sommes pas intéressés à la parallélisation des programmes,
- nous voulons résoudre les problèmes d'optimisation en tant que tels et non juste déterminer l'ordre dans lequel les optimisations doivent être appliquées,
- nous envisageons de résoudre plusieurs problèmes simultanément.

2.5 Résumé

Les méthodes métaheuristiques sont des méthodes de recherche non traditionnelles qui ont été utilisées pour résoudre des problèmes d'optimisation NP-durs. Parmi ces méthodes se trouvent les algorithmes génétiques et le recuit simulé.

Les algorithmes génétiques sont basés sur le processus d'évolution naturel. Pour résoudre un problème, ils commencent avec un ensemble de solutions au problème générées le plus souvent de façon aléatoire. Après, ils récombinent les solutions (en utilisant des opérateurs génétiques) pour générer de nouvelles solutions qui sont évaluées par une fonction de coût qui mesure sa qualité.

Le recuit simulé est une méthode de recherche locale basée sur le procédé de trempe des substances pures. Il commence avec une solution initiale sur laquelle il applique un opérateur de voisinage afin de parcourir l'espace de recherche. Au début, l'algorithme cherche partout dans l'espace de solutions mais à mesure que la solution s'améliore il cherche plus localement, puisqu'il suppose qu'il est proche d'un optimum.

L'utilisation de méthodes métaheuristiques pour résoudre des problèmes NP-durs impliqués dans le processus de compilation est rare. Les compilateurs utilisent des approches heuristiques classiques pour faire face à ces problèmes.

Parmi les problèmes importants que le compilateur doit résoudre, nous trouvons le problème d'allocation de registres qui est souvent modélisé comme un problème de coloriage de graphe, et le problème d'ordonnancement d'instructions qui est résolu localement dans des blocs de base en utilisant des algorithmes comme l'ordonnancement par liste.

À notre connaissance les seuls travaux qui utilisent les méthodes métaheuristiques en compilation sont :

- Beaty [Bea91, Bea93] qui utilise un algorithme génétique et un algorithme de recherche tabou pour résoudre le problème d'ordonnancement d'instructions. Il utilise un algorithme d'ordonnancement par liste où la priorité de chaque instruction est trouvée par la métaheuristique.
- Williams [WW96] et Nisbet [Nis98] utilisent des AGs pour effectuer la parallélisation automatique des boucles dans un compilateur pour un langage parallèle.

Ces travaux sont des approches spécifiques pour des problèmes particuliers, notre but est d'utiliser les métaheuristiques en compilation de façon globale. Nous voulons profiter des avantages des métaheuristiques pour résoudre plus d'un problème simultanément, et ainsi pouvoir tenir compte automatiquement de leur interdépendances.

Chapitre 3

Une première expérience avec le compilateur GNU C

Étant donné le manque de travaux sur l'utilisation de méthodes métaheuristiques en compilation il est difficile de savoir *a priori* si son utilisation est viable. De plus, il est difficile de déterminer quelle est la meilleure façon de les utiliser.

Nous croyons qu'une approche générale qui profite de la dépendance des problèmes de compilation peut être une alternative intéressante. Cependant, avant d'essayer une telle approche nous croyons convenable de faire une expérience plus simple, pour vérifier la viabilité de l'utilisation de métaheuristiques en compilation.

Comme première expérience, nous proposons de remplacer l'heuristique qu'un compilateur utilise pour résoudre un problème donné, par une métaheuristique pour résoudre le même problème.

Nous nous attendons à ce que la métaheuristique soit compétitive aux approches traditionnelles, c'est-à-dire, qu'elle obtienne des résultats comparables par rapport à la vitesse du code généré. Si ceci est vrai, nous avons de bonnes chances qu'une approche plus générale, qui profite des aspects que les heuristiques traditionnelles ignorent, soit

performante. Mais, si la métaheuristique n'est pas au moins comparable aux heuristiques traditionnelles, il sera prématuré de s'embarquer dans une approche plus générale.

Puisque notre but est uniquement de valider notre hypothèse (que les méthodes métaheuristiques ont un potentiel en compilation) à l'aide d'un premier prototype, nous n'avons pas cherché à étudier toutes les ramifications de nos choix d'implantation. Nous nous basons sur d'autres travaux et notre intuition pour guider nos choix.

Nous proposons de résoudre le problème d'allocation de registres, puisqu'il a un grand impact sur la qualité du code généré et parce qu'il se modélise comme un problème de coloriage de graphe. Le problème de coloriage de graphe a été résolu avec beaucoup de succès en utilisant des métaheuristiques. Nous utilisons donc les travaux sur le problème de coloriage de graphe comme base pour définir notre allocateur de registres.

Nous considérons uniquement le débordement des variables comme technique de résolution. Nous ne faisons pas de division d'intervalles de vies, ce qui veut dire que lorsque nous assignons un registre à une variable, cette variable est dans le même registre pour toute la durée sa vie. Ce choix se justifie par sa simplicité et aussi parce que la plupart des compilateurs utilisent cette stratégie. Nous pouvons donc faire des comparaisons valables.

Afin de résoudre le problème d'allocation intraprocédurale nous remplaçons l'allocateur de GNU C [Sta91] par un allocateur basé sur un algorithme génétique (AG-Allocateur) et par un allocateur basé sur un recuit simulé (RS-Allocateur). Nous faisons une analyse de performance des deux allocateurs afin de comparer la qualité du code généré par rapport au temps d'exécution.

Dans ce chapitre nous présentons la représentation du problème et la fonction de coût utilisées par les deux allocateurs et les différents ajustements apportés à chaque algorithme. Ensuite, nous donnons les résultats de l'ajustement des paramètres. Pour commencer, nous considérons le problème isolément, ce qui veut dire que nous prenons un graphe d'interférence donné et nous générons une assignation. Enfin, nous incluons

notre algorithme dans le compilateur GNU C pour faire des tests sur des programmes réels. Tous les résultats présentés dans ce chapitre ont été obtenus en utilisant une machine avec un processeurs AMD Athlon 1200 MHz avec 512 Mo de mémoire vive, et le système d'exploitation Red Hat Linux 7.1.

3.1 La conception

Dans cette section, nous décrivons la représentation du problème et la fonction de coût utilisée par AG-Allocateur et RS-Allocateur.

3.1.1 La représentation du problème

Nous représentons le problème d'allocation de registres par une chaîne a, de longueur n, de nombres entiers entre -1 et r-1, où n est le nombre de variables du graphe d'interférence et r est le nombre de registres disponibles dans la machine. Dorénavant, nous appellerons cette chaîne un individu. La valeur a_i représente le registre auquel la variable est assignée. La valeur -1 signifie que la variable est assignée à la mémoire.

Nous sélectionnons plusieurs opérateurs génétiques pour AG-Allocateur en raison de leur bonne performance lorsqu'utilisés pour résoudre des problèmes comme le voyageur de commerce [Pot96], l'ordonnancement [Kri96] et le coloriage de graphe [FF97]. Ces opérateurs sont implantés et évalués empiriquement afin d'identifier les plus performants.

- Croisement : nous implantons les opérateurs OBX et PBX décrits dans le chapitre précédent.
- Mutation : nous implantons les deux opérateurs de mutation suivants :
  1. Changement aléatoire : cet opérateur conserve l'idée de base de l'opérateur de mutation, puisqu'il fait un petit changement dans l'individu en changeant l'élément à une position choisie aléatoirement par une autre valeur choisie aléatoirement aussi (entre -1 et $r-1$).

2. Recherche par descente (*Hill-Climbing*) : cet opérateur provoque un changement considérable dans l'individu. Il effectue une recherche locale à partir de l'individu. Pour ce faire, il applique un opérateur de voisinage un nombre fixe de fois ou jusqu'à ce qu'un optimum local soit trouvé. Nous utilisons un changement aléatoire dans une position de l'individu comme opérateur de voisinage.

Pour RS-Allocateur il faut définir l'opérateur de voisinage et l'individu initial.

- L'opérateur de voisinage : cet opérateur consiste à faire un nombre n de changements aléatoires dans l'individu, ce qui équivaut à utiliser n fois l'opérateur de mutation (changement aléatoire) d'AG-Allocateur.
- L'individu initial : RS-Allocateur commence avec le pire individu réalisable. Cet individu a toutes ses variables en mémoire, i.e. tous les éléments de l'individu sont égaux à -1.

3.1.2 La fonction de coût

La métaheuristique doit aussi estimer le coût de chaque configuration. La fonction de coût est la fonction que la métaheuristique essaie de minimiser. Par conséquent la précision de cette fonction est fondamentale. Dans notre cas, nous voulons minimiser le temps d'exécution du code généré. Ainsi, la fonction de coût devrait idéalement donner le temps d'exécution du programme lorsque l'allocation de registre définie dans l'individu est utilisée. Pour cette première expérience nous utilisons une fonction de coût approximative.

Étant donné que les instructions qu'utilisent des registres sont exécutées beaucoup plus rapidement que celles qui font des accès à la mémoire, le temps d'exécution d'un programme est fortement affecté par le nombre de débordements réalisés. Donc, nous utilisons ce dernier comme fonction de coût.

Pour nous permettre d'implanter cette fonction de coût, nous supposons que les données suivantes sont connues : le nombre de fois que chaque variable i est utilisée en dehors d'une boucle (u_i), le nombre de fois que la variable i est utilisée dans une boucle b ($uboucle_i(b)$) et la profondeur de la boucle b ($pboucle_i(b)$).

Soient

$$x_i = \begin{cases} 0 & ssi\ a_i >= 0\ dans\ l'individu \\ 1 & ssi\ a_i < 0\ dans\ l'individu \end{cases}$$

c_b : le nombre de fois que la boucle b est exécutée.

a_i : assignation de la variable i

Le coût de débordement est donné par

$$\sum_i (x_i * (u_i + \sum_b uboucle_i(b) * pboucle_i(b) * c_b))$$

Il convient de souligner qu'une configuration est réalisable si et seulement si l'allocation codé par l'individu respecte les conflits entre les variables. Dans le cas contraire, nous ajoutons une pénalité à la fonction de coût de façon à ce qu'aucune configuration non-réalisable ait un meilleur coût qu'une autre qui est réalisable. La valeur de la pénalité est le coût de la pire solution faisable, c'est-à-dire, celle où toutes les variables sont en mémoire.

3.2 Les graphes de test

Pour effectuer des tests sur nos algorithmes, nous utilisons deux sortes de graphes : des graphes aléatoires de densité 0.5 (ils ont la moitié des arêtes du graphe complet) et des graphes d'interférence réels générés par un compilateur pour le langage ML [App96]. La table 3.1 montre les caractéristiques des graphes utilisés.

Nom	Nombre de noeuds	Type de graphe
g-96	96	Aléatoire
g-212	212	Aléatoire
graph17	17	Réel
graph18	18	Réel
graph28	28	Réel
graph42	42	Réel
graph58	58	Réel
graph61	61	Réel
graph62	62	Réel
graph95	95	Réel
graph99	99	Réel
graph136	136	Réel

TAB. 3.1 – Caractéristiques des graphes de test

3.3 L'ajustement de paramètres

Même s'il existe quelques résultats théoriques [PL00, GDC92] indiquant comment fixer les paramètres pour un AG, dans les cas pratiques cet ajustement est basé sur l'intuition et sur des tests numériques. Ceci s'explique par le fait que les résultats théoriques sont développés pour analyser les paramètres indépendamment les uns des autres et ils sont testés sur des problèmes simples. À notre avis, ces résultats théoriques cherchent à montrer qu'il est possible de garantir la convergence des AGs plutôt qu'à donner des directives pour l'ajustement de paramètres.

Généralement, l'ajustement de paramètres d'une métaheuristique est réalisé sur quelques instances du problème ciblé. Pour ces instances on réalise des tests avec différentes valeurs pour les paramètres et on les généralise. Au début, les valeurs des paramètres sont choisies aléatoirement ou en se basant sur des paramètres qui ont été utilisés avec succès pour des problèmes semblables. Après, on raffine les paramètres en les modifiant un à un. L'ordre dans lequel les paramètres doivent être ajustés n'est pas clair ; alors, on se base sur les résultats obtenus. Généralement, on ajuste en premier les paramètres auxquels l'algorithme est plus sensible.

Nous utilisons certains graphes (g-96, graph42, graph61 et graph99) afin d'effectuer l'ajustement de paramètres. Cependant, étant donné que les résultats sont semblables dans tous les cas, nous exposons dans cette section les résultats obtenus pour g-96. Les résultats pour les autres graphes se trouvent à l'annexe A. Dans tous les graphiques de cette section l'abscisse représente le coût de la solution, c'est-à-dire le temps d'exécution du programme à compiler, et l'ordonnée indique le nombre de générations.

3.3.1 L'algorithme génétique

Pour utiliser l'AG, nous devons premièrement choisir les opérateurs génétiques (croisement et mutation) et après déterminer les paramètres avec lesquels l'AG aura une meilleure convergence. Étant donné que le processus pour choisir les opérateurs génétiques est le même que pour déterminer les paramètres, nous les traitons de la même façon. Dans cette section nous parlerons de paramètres pour nous référer aux deux cas.

Taille de population		g-96	graph42
25	Individus évalués	8637	4201
	Coût	48	21
50	Individus évalués	9011	2363
	Coût	21	18
75	Individus évalués	11290	9634
	Coût	25	36
100	Individus évalués	7254	12622
	Coût	18	41
150	Individus évalués	13295	19293
	Coût	45	48
200	Individus évalués	17342	26891
	Coût	21	48
300	Individus évalués	28201	32605
	Coût	25	56

TAB. 3.2 – Résultats obtenus en utilisant différentes tailles de population

Dans des tests préliminaires nous avons observé que l'AG est très sensible à l'opérateur de mutation, alors nous commençons par choisir l'opérateur de mutation et la probabilité de mutation. Après nous choisirons l'opérateur de croisement.

Pour la taille de la population nous avons testé différentes valeurs sur les graphes g-96 et graph42. La table 3.2 montre la qualité de la solution obtenue et le nombre d'individus évalués dans chaque cas. Pour g-96 les meilleurs résultats sont obtenus en utilisant des tailles de population de 50, 100 et 200, tandis que pour graph42 les meilleurs tailles sont 25, 50, 75. Notons que ces valeurs sont à peu près 0.5, 1 et 2 fois le nombre de nœuds du graphe. Donc pour ajuster la taille de la population nous utiliserons ces fonctions.

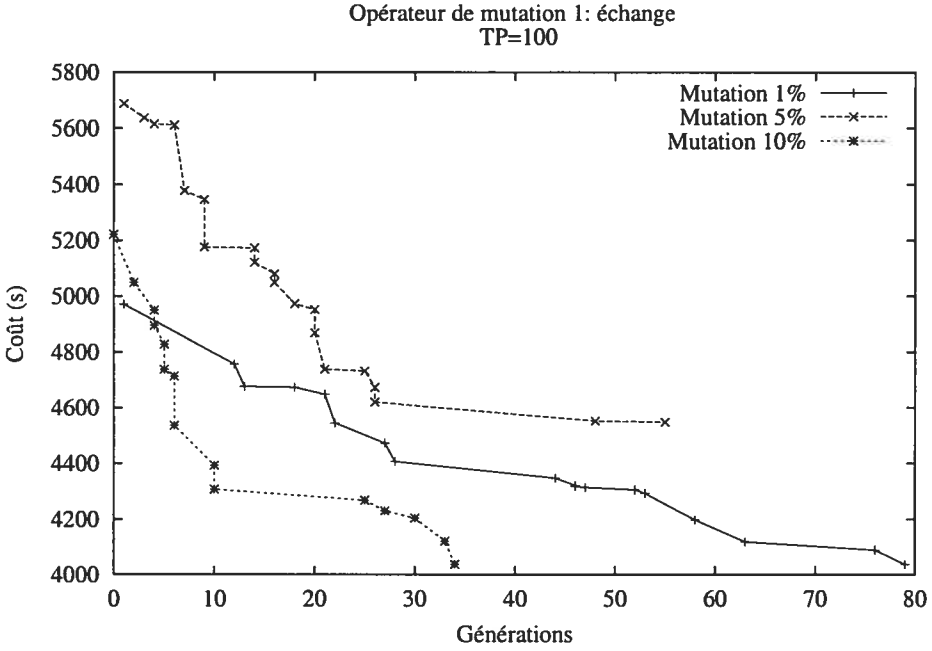


FIG. 3.1 – Graphe de convergence en utilisant l'opérateur de mutation par échange

Premièrement, nous testons les deux opérateurs de mutation en utilisant le croisement PBX. Les figures 3.1 et 3.2 montrent les résultats obtenus en utilisant l'opérateur de mutation d'échange et de descente, respectivement, avec différentes probabilités de mutation. Nous pouvons constater que le choix de l'opérateur de mutation a un grand impact : la solution de la meilleure qualité obtenue en utilisant l'opérateur d'échange a un coût de 3400 après 35 générations, tandis qu'en utilisant la descente, nous obtenons une solution de coût de 21 après moins de 20 générations. Il faut cependant remarquer que la mutation par descente demande beaucoup plus d'effort que celle par l'échange. Alors nous testons cet opérateur en utilisant différentes probabilités de mutation et une

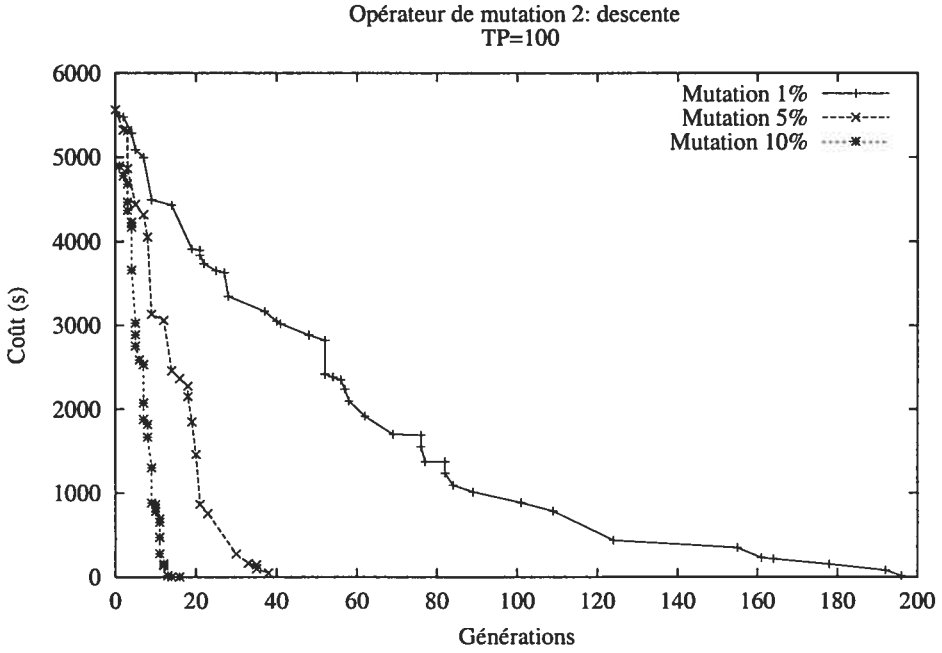


FIG. 3.2 – Graphe de convergence en utilisant l'opérateur de mutation par descente

plus petite taille de population (TP) (figure 3.3).

Afin de choisir entre les deux opérateurs de mutation, nous comparons ceux qui montrent les meilleures performances :

- la mutation par échange avec probabilité de mutation de 10% ;
- la mutation par descente avec probabilité de mutation de 10% et une taille de population de 100 et
- la mutation par descente avec probabilité de mutation de 50% et une taille de population de 50.

Dans ces tests, nous considérons le coût de la solution versus le nombre d'individus évalués par l'algorithme (figure 3.4).

Grâce à la figure 3.4 nous constatons que la mutation par descente avec une probabilité de 10% est la meilleure, car elle permet d'obtenir une solution de bonne qualité en évaluant un nombre plus petit d'individus, ce qui minimise le temps de traitement.

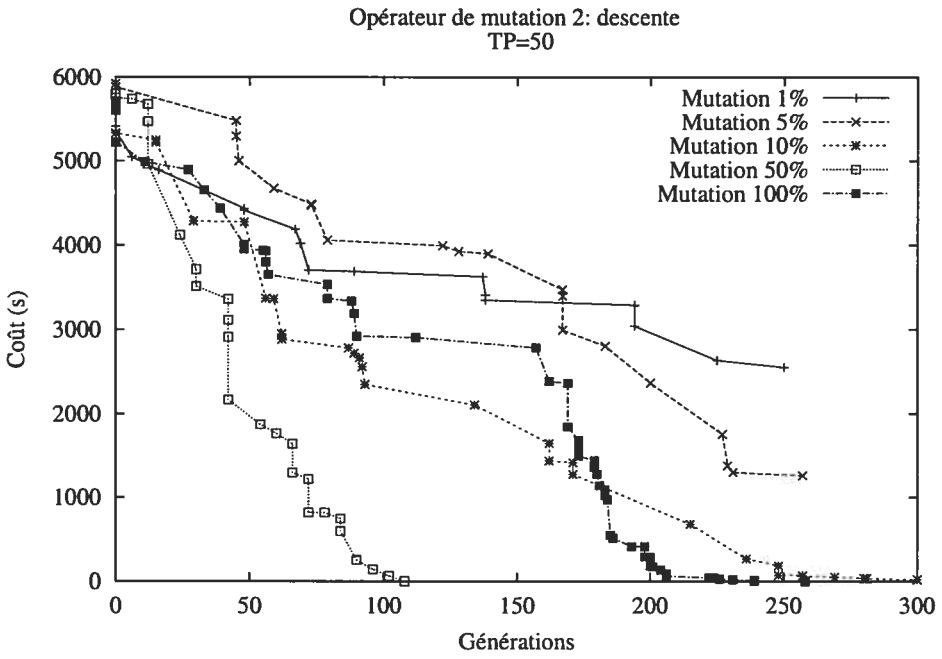


FIG. 3.3 – Graphe de convergence pour l'opérateur de mutation par descente

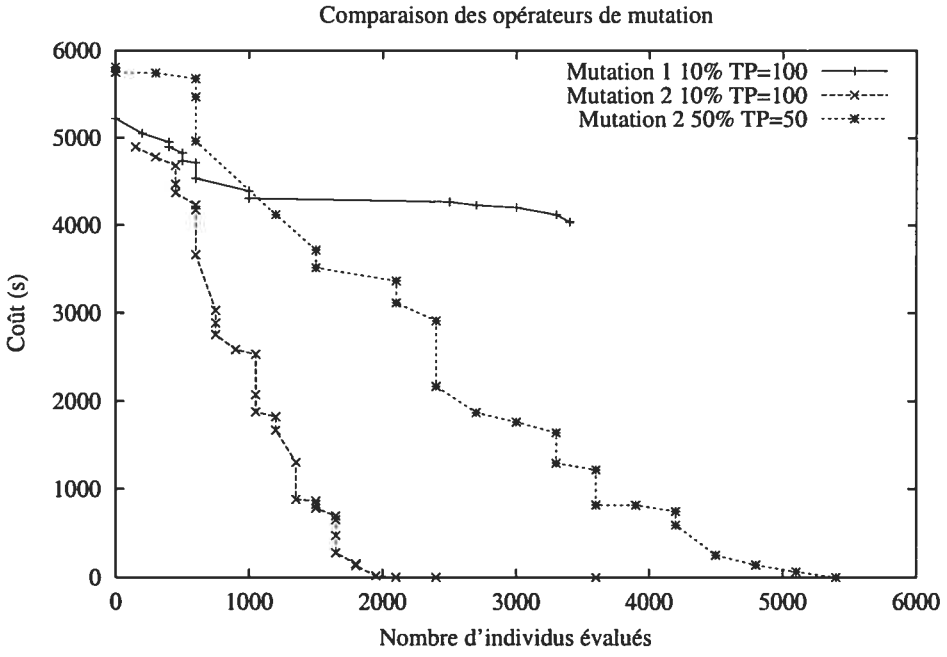


FIG. 3.4 – Graphe de convergence pour les opérateurs de mutation

Maintenant, nous faisons les tests nous permettant de choisir l'opérateur de croisement. La figure 3.5 illustre le comportement des opérateurs de croisement avec la mutation par descente. Bien que la différence entre les opérateurs ne soit pas très grande, l'opérateur OBX montre une convergence un peu plus rapide. Pour tous les graphes OBX obtient des solutions de meilleure qualité, et pour 3 des 4 graphes utilisés pour l'ajuster les paramètres requiert l'évaluation d'un nombre plus petit d'individus. Par conséquent, nous utilisons cet opérateur.

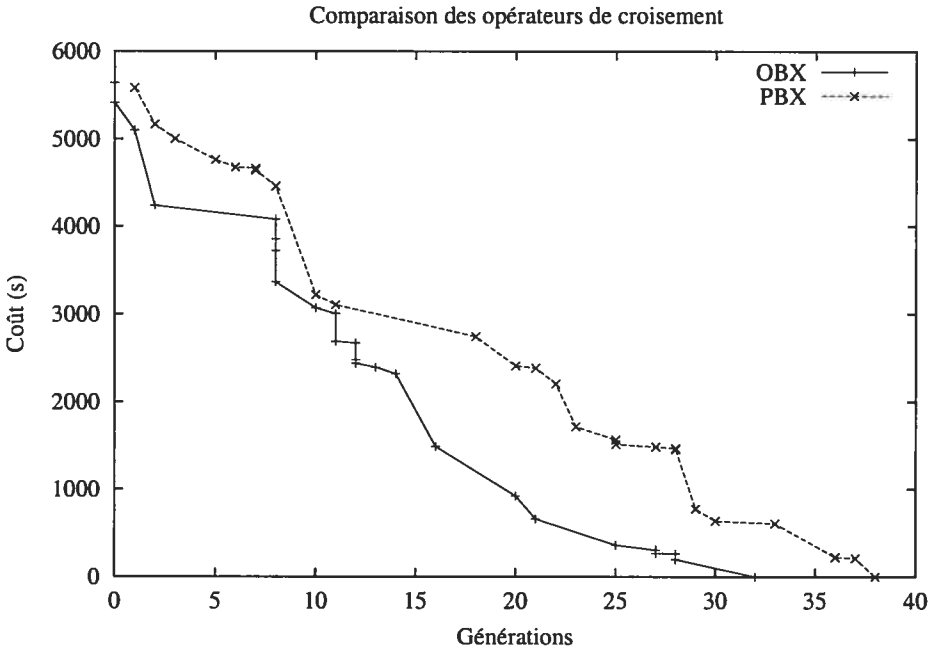


FIG. 3.5 – Graphe de convergence en utilisant les opérateurs de croisement

La figure 3.6 présente les courbes obtenues pour les différentes tailles de population. Les valeurs testées pour la taille de la population sont n, $n/2$ et $2*n$, où n est le nombre de nœuds du graphe d'interférence. Nous observons que AG-Allocateur en utilisant une taille de population de n trouve des bonnes solutions en évaluant un nombre plus petit d'individus.

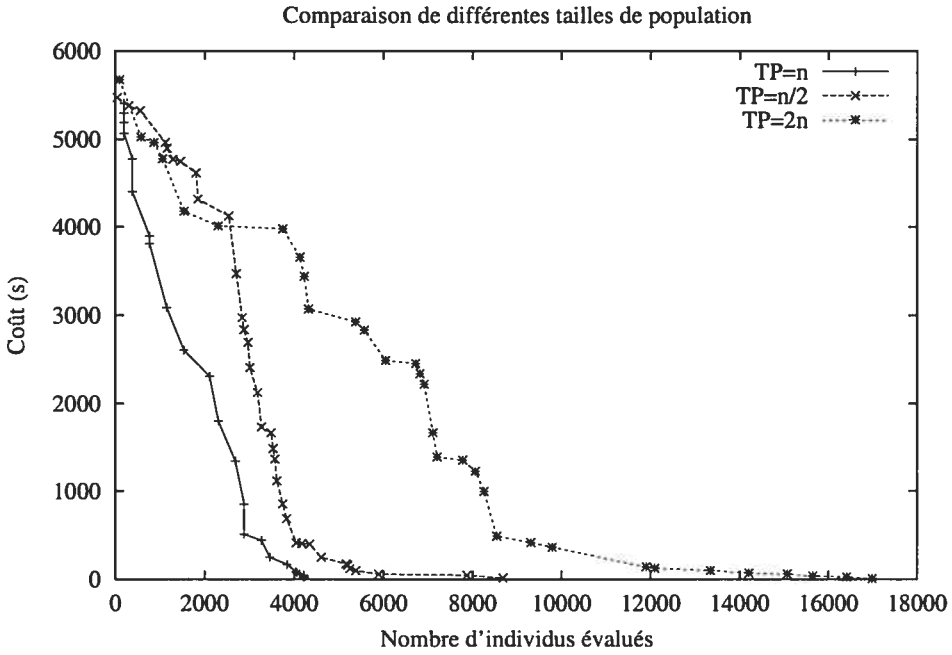


FIG. 3.6 – Graphe de convergence en utilisant différentes tailles de population

3.3.2 Le recuit simulé

Pour ajuster les paramètres de RS-Allocateur, nous considérons seulement la température, qui est le paramètre le plus important dans cette méthode. Les figures 3.7 et 3.8 présentent les courbes obtenues sur les graphes g-96 et graph61, respectivement, avec différentes températures. Pour ces graphes, aussi bien que pour les autres, les courbes de convergence sont très similaires. Alors nous choisissons les valeurs qui exigent l'effort minimal : 5*n comme température initiale et 0.1 comme température finale (où n est le nombre de nœuds du graphe).

3.4 Résultats

Nous avons observé que plusieurs exécutions d'un même programme peuvent présenter des variations allant jusqu'à un 5% dans leur temps d'exécution. Pour éviter que ce

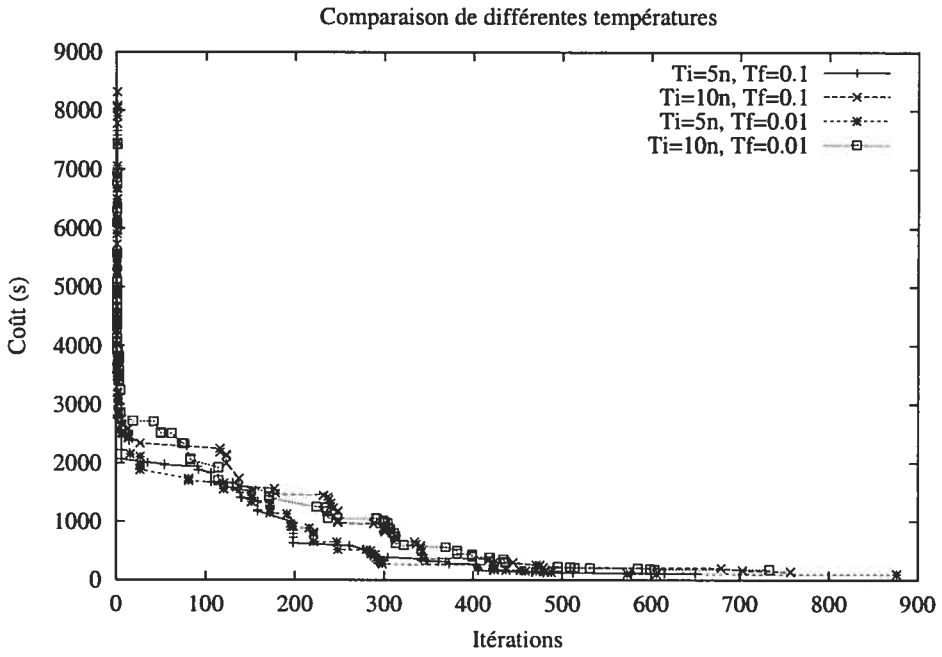


FIG. 3.7 – Graphe de convergence en utilisant différentes températures (g-96)

fait perturbe notre analyse, nous prenons la moyenne de 15 exécutions.

La table 3.3 présente les résultats obtenus sur 10 graphes de différentes tailles en considérant une machine ayant 21 registres [App96] disponibles pour l'allocation des variables. Ici, tous les graphes utilisés sont des graphes provenant de problèmes réels. D'après la table 3.3, nous pouvons constater que dans presque tous les cas, le coût de la solution est plus petit que 21. Ceci veut dire que les allocateurs réussissent à trouver un coloriage pour le graphe. Par conséquent, la différence dans le coût est due au nombre de registres utilisés. Pour graph95, graph99 et graph136 AG-Allocateur n'arrive pas à trouver un coloriage pour le graphe complet et il doit faire déborder des variables en mémoire, ce qui augmente le coût de la solution.

Même si RS-Allocateur montre une meilleure performance qu'AG-Allocateur, nous croyons convenable d'intégrer les deux allocateurs dans GNU C et d'effectuer des tests sur des programmes réels. AG-Allocateur a une bonne performance pour les graphes de petite et moyenne tailles (il trouve toujours une coloration) et, étant donné que nous ne

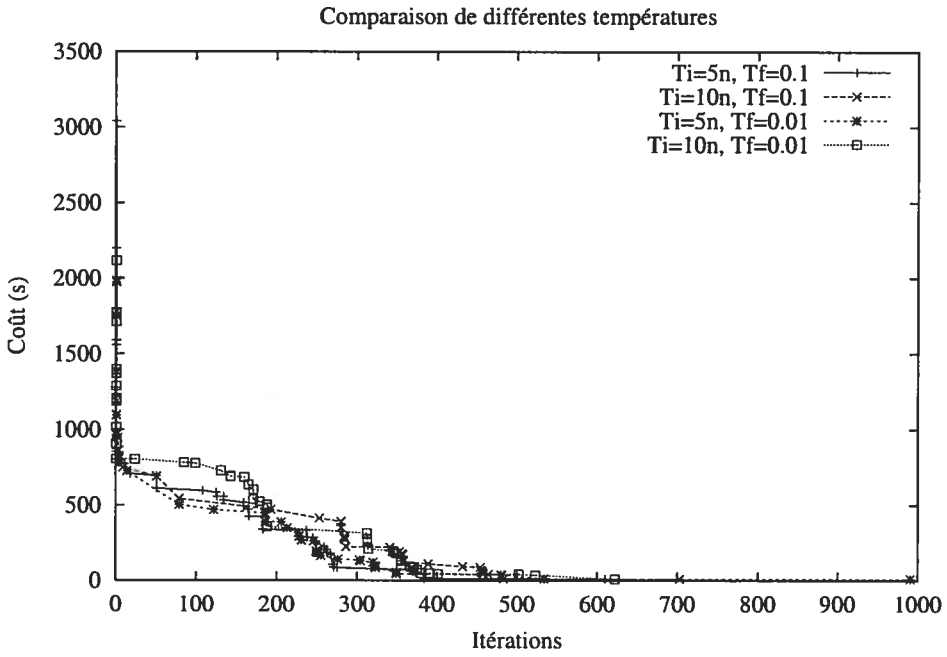


FIG. 3.8 – Graphe de convergence en utilisant différentes températures (graph61)

connaissons pas la taille des graphes des programmes réels, nous ne pouvons pas établir *a priori* lequel des deux allocateurs sera plus performant.

3.4.1 Le compilateur GNU C

Afin de mesurer l'impact de notre allocateur de registres sur la performance du code généré, nous avons intégré l'allocateur à un compilateur. Nous avons choisi GNU C (version 2.95.2) [Sta91] puisqu'il possède des caractéristiques qui le rendent fort attirant :

- GNU C est un compilateur très complet et très utilisé, ce qui nous donne un point de comparaison réaliste.
- L'allocateur de registres de GNU C est assez isolé pour qu'il puisse être modifié sans perturber les autres parties du compilateur. Ainsi, nous pouvons mesurer l'impact sur la performance qui est provoquée par l'allocateur seulement.
- Les sources de GNU C sont disponibles.

Graphe	RS-Allocateur Coût Solution	RS-Allocateur Temps de calcul (s)	AG-Allocateur Coût Solution	AG-Allocateur Temps de calcul (s)
graph17	2	7.72	21	1.18
graph18	4	7.86	21	1.25
graph28	7	11.95	21	6.24
graph42	8	16.22	21	23.69
graph58	8	22.43	21	58.67
graph61	11	24.18	21	91.83
graph62	10	23.85	21	62.47
graph95	41	38.50	809	51.01
graph99	9	39.78	167	92.71
graph136	31	54.90	1883	85.22

TAB. 3.3 – Résultats obtenus en utilisant RS-Allocateur et GA-Allocateur

Nous utilisons l'information qu'amasse GNU C pour construire le graphe d'interférence. Dans le graphe, nous enregistrons les conflits entre variables du programme (variables qui sont vivantes en même temps) et entre variables du programme et registres physiques (il y aura un conflit entre une variable et un registre physique quand la variable ne peut pas être assignée à ce registre). Nous utilisons une fonction de coût plus simple car la seule information que nous avons est le nombre de fois que chaque variable est utilisée. Alors, le coût de débordement est calculé de la manière suivante :

$$\sum_i (x_i * u_i)$$

où :

x_i vaut 1 si la variable i est en mémoire et 0 sinon.

u_i est le nombre de fois que la variable i est utilisée.

3.4.2 Analyse de performance

Nous sélectionnons cinq programmes écrits en C. Ces programmes résolvent des problèmes connus. Pour chacun, nous testons différentes données d'entrée. La table 3.4

présente une brève description des programmes de test et des données d'entrée utilisées.

Programme de test	Description	Données d'entrée
flow	algorithme exhaustif pour trouver le flux maximal dans un réseau	réseau de 20 nœuds (reseau20) réseau de 25 nœuds (reseau25)
fulk	algorithme de Ford-Fulkerson pour trouver le flux maximal dans un réseau	réseau de 200 nœuds (reseau200) réseau de 300 nœuds (reseau300) réseau de 400 nœuds (reseau400) réseau de 500 nœuds (reseau500)
queens	algorithme pour resoudre le problème des n reines	12 reines
puzzle	algorithme pour le jeu du 8 puzzle	8 pièces
wan	algorithme de WAN pour résoudre le problème de *cutting-stock* [Wan83]	problèmes aléatoires de différentes tailles : p1, p2, p3, p4, p5, p6

TAB. 3.4 – Programmes test utilisés

La table 3.5 présente les temps de compilation requis par GNU C, par RS-Allocateur et par AG-Allocateur pour les cinq programmes de test. Les temps requis par AG-Allocateur et RS-Allocateur ne sont pas très différents. Par contre, ils sont tous les deux beaucoup plus grand que ceux utilisés par GNU C. Donc, l'utilisation d'un allocateur basé sur une métaheuristique sera viable seulement pour les usagers qui peuvent se permettre un temps de compilation élevé.

Nom	RS-A (s)	AG-A (s)	GNU C (s)
flow	12.6	15.7	0.07
fulk	13.0	17.2	0.10
queens	30.5	45.2	0.16
puzzle	35.6	36.4	0.15
wan	72.1	83.2	0.40

TAB. 3.5 – Temps de compilation requis par RS-Allocateur, GA-Allocateur et GNU C

La table 3.6 présente les temps d'exécution (moyenne de 15 exécutions) en secondes obtenus pour chaque programme de test lorsqu'on compile avec GNU C (option d'optimisation -O2, afin qu'il effectue l'allocation de registres et autres optimisations), GNU C avec RS-Allocateur et GNU C avec AG-Allocateur. La table 3.6 présente aussi l'amé-

lioration apporté par nos approches par rapport à GNU C.

Programme de test	GNU C (s)	RS-A (s)	AG-A (s)	Amélioration RS-A (%)	Amélioration AG-A (%)
flow reseau20	0.90	0.90	0.90	0.00	0.00
flow reseau25	44.88	44.87	44.88	0.02	0.00
fulk reseau200	0.54	0.54	0.54	0.00	0.00
fulk reseau300	1.99	1.99	2.00	0.00	-0.50
fulk reseau400	6.72	6.69	6.69	0.45	0.45
fulk reseau500	12.24	12.20	12.22	0.33	0.16
puzzle 8	13.28	13.22	13.25	0.45	0.23
queens 12	1.22	1.22	1.22	0.00	0.00
wan p1	0.70	0.70	0.70	0.00	0.00
wan p2	0.44	0.43	0.43	0.76	0.76
wan p3	22.98	22.71	22.80	1.20	0.80
wan p4	340.56	333.48	339.19	2.08	0.40
wan p5	227.02	224.85	226.60	0.96	0.19
wan p6	325.30	321.99	323.83	1.02	0.45

TAB. 3.6 – Temps d'exécution des programmes de test

Dans la plupart des cas, les performances obtenues par l'utilisation de RS-Allocateur et AG-Allocateur sont similaires à celles obtenues avec GNU C. En regardant les allocations obtenues par les trois approches, nous constatons que, sauf dans le cas de *wan* compilé avec RS-Allocateur, les allocations sont équivalentes dans tous les cas. Donc, les petites différences de performance sont dues aux variations aléatoires habituelles dans une série d'exécutions d'un même programme.

Dans le cas de *wan*, pour toutes les procédures sauf une, les allocations trouvées par RS-Allocateur sont les mêmes que celles trouvées par les autres approches. Dans la procédure où l'allocation est différente, il y a une variable que GNU C déborde vers la mémoire que RS-Allocateur assigne à un registre. Cette différence dans l'allocation apporte des petits gains de vitesse qui varient entre 0% et 2%. Il est surprenant que les gains obtenus varient pour le même programme lorsqu'il est exécuté avec de données différentes. Pour comprendre ce comportement, nous avons mesuré le nombre de fois que cette procédure est exécutée dans chaque cas (voir la table 3.7). Nous voyons que

les tests où cette procédure est exécutée souvent, sont ceux qui obtiennent des gains de vitesse plus importants. Dans *wan p1* la procédure est exécutée 8647 fois et RS-Allocateur n'obtient pas d'amélioration ; par contre dans *wan p4* la procédure est exécutée 148758 fois et le gain de vitesse est de 2%. Les codes assembleur pour cette procédure, générés par GNU C et par RS-Allocateur, sont présentés à l'annexe B.

Programme de test	Nombre de fois que la procédure est exécute
wan p1	8647
wan p2	10697
wan p3	31741
wan p4	148758
wan p5	100065
wan p6	129135

TAB. 3.7 – Nombre de fois que la procedure est exécutée

3.5 Résumé

À partir des expériences décrites dans ce chapitre, nous pouvons conclure que l'utilisation de méthodes métaheuristiques dans la compilation est viable. Avec GNU C, nous avons obtenu des performances semblables pour l'allocation de registres avec un allocateur standard, avec un allocateur de registres basé sur un AG et avec un allocateur de registres basé sur un RS. Même si les allocateurs basés sur des métaheuristiques ne sont pas plus performants que GNU C sur nos programmes de test, ils permettent d'obtenir des résultats comparables.

Nous croyons qu'une approche plus agressive pourrait mener à des performances plus intéressantes. Nous discutons de telles approches dans les prochains chapitres.

Chapitre 4

Compilateur génétique : une approche unifiée

Dans le chapitre précédent, nous avons illustré que l'utilisation des méthodes métaheuristiques dans la compilation est viable. Dans cette première tentative nous nous sommes contentés de remplacer un allocateur de registres classique par un allocateur basé sur une métaheuristique. La performance obtenue est semblable à celles avec les heuristiques traditionnelles. Toutefois, la puissance des méthodes métaheuristiques nous laisse croire qu'une approche unifiée permettrait de mieux exploiter leurs avantages et pourrait être plus performante. Cette approche unifiée fait l'objet de ce chapitre.

Premièrement, nous décrivons l'idée générale de notre approche et après nous présentons la conception et l'implantation réalisée dans le but de résoudre les problèmes d'allocation de registres et d'ordonnancement d'instructions.

4.1 Les approches traditionnelles

Un compilateur doit résoudre plusieurs problèmes d'optimisation afin de générer du code de bonne qualité. Ces problèmes sont normalement résolus les uns après les autres et les décisions prises à chaque étape ont un impact sur la résolution du problème suivant. Or, l'ordre dans lequel les problèmes sont résolus n'est pas nécessairement fixe. Quelques-uns doivent être résolus dans un ordre spécifique, mais d'autres peuvent être résolus à divers moments de la compilation et il n'est pas clair quel ordre est le meilleur. Par exemple, il faut faire la sélection d'instructions avant l'allocation de registres, puisque le fait de sélectionner les instructions définit de façon implicite le nombre et le type de registres dont chaque instruction a besoin. Par contre, l'ordonnancement d'instructions peut être effectué avant ou après l'allocation des registres. Si l'ordonnancement est fait avant l'allocation, les intervalles de vie des variables et, par conséquent, l'ensemble des allocations réalisables sont changés. Notons aussi que le fait de trouver un meilleur ordonnancement a tendance à allonger les intervalles de vie des variables, ce qui peut empêcher de trouver une bonne allocation ultérieurement (car il faudrait plus de registres). De plus, l'allocation de registres ajoute des instructions et ces instructions ne sont pas considérées lors de l'ordonnancement. Par contre, si l'allocation est fait avant l'ordonnancement, des dépendances fictives sont ajoutées au graphe de dépendances, lesquelles proviennent de la réutilisation des registres. Ainsi, deux instructions qui étaient indépendantes peuvent devenir dépendantes. Étant donné qu'elles utilisent le même registre, les possibilités pour l'ordonnancement d'instructions s'en trouvent diminuées.

Afin de s'attaquer aux difficultés liées à l'interdépendance des problèmes, les compilateurs réalisent certaines optimisations plus d'une fois. Bien que cette approche améliore habituellement la qualité du code généré, elle n'est pas toujours efficace, et la performance de celle-ci dépend beaucoup des caractéristiques du programme source.

À titre d'exemple concret, nous énumérons dans l'ordre les principales optimisations effectués par GNU C [Sta91]. Pour une présentation des différentes optimisations,

voir [Muc97].

- Génération du code intermédiaire. Chaque arbre de syntaxe est traduit en code intermédiaire. Ici, le compilateur choisit la façon d'organiser les boucles. Il optimise, si possible, les énoncés *if* lorsqu'ils ont des conditions qui contiennent des expressions booléennes.
- Optimisation des branchements. Un branchement à l'instruction suivante ou un branchement à un autre branchement est simplifié. Le compilateur convertit certaines parties de code utilisant des branchements en code linéaire.
- Élimination des expressions communes. Les expressions communes sont éliminées et les constantes sont propagées.
- Optimisation de branchement. Celle-ci est nécessaire car la propagation de constantes peut introduire un nouveau branchement à un autre branchement.
- Optimisation des boucles. Le compilateur déplace les expressions invariantes en dehors des boucles. Il effectue le déroulement des boucles.
- Élimination des expressions communes introduites par l'optimisation des boucles.
- Ordonnancement d'instructions. Le compilateur effectue l'ordonnancement d'instructions pour chaque bloc de base.
- Allocation de registres. Le compilateur réalise l'allocation de registres aux niveaux local et global.
- Ordonnancement d'instructions.
- Optimisation de branchement.
- Génération de code : Le compilateur génère le code assembleur pour la fonction. Il effectue des optimisations spécifiques à l'architecture (*peephole optimization*).

Plusieurs des problèmes auxquels les compilateurs doivent faire face sont des problèmes NP-durs. Ils sont résolus à l'aide d'heuristiques qui obtiennent des solutions approximatives. Pour définir ces heuristiques, les concepteurs de compilateurs se basent sur les caractéristiques de la machine et sur des suppositions sur les caractéristiques des programmes sources. Ceci fait en sorte que les heuristiques fonctionnent bien dans

quelques cas et mal dans d'autres, selon que les suppositions étaient vraies au fausses.

Même si dans quelques cas, les heuristiques sont assez bonnes et que le compilateur effectue les optimisations plus d'une fois, les solutions obtenues ne sont optimisées que localement parce que la dépendance entre les différents problèmes n'est pas considérée.

De plus, à toute cette complexité s'ajoute l'impact de l'évolution et des changements continus dans l'architecture des machines, ce qui demande de redéfinir les heuristiques. Par exemple, si le jeu d'instructions d'une machine est modifié, il faut modifier les heuristiques effectuant la sélection d'instructions. Si le *pipeline* de la machine change ou la vitesse relative de la mémoire, il faut récrire les heuristiques liées à l'ordonnancement d'instructions. Si de nouveaux registres sont ajoutés, l'heuristique d'allocation de registres doit être modifiée. Et ainsi de suite.

Pour générer du code performant le compilateur doit résoudre un ensemble de problèmes complexes, dont plusieurs d'entre eux sont interdépendants. Notons que la définition de code performant n'est pas précise parce qu'elle dépend des besoins de l'usager. Par exemple, on peut considérer le temps de compilation, la taille du code objet, le temps d'exécution du code objet ou la quantité de mémoire requise à l'exécution comme étant la caractéristique qui définit la performance du code.

Une approche unifiée qui considère les problèmes simultanément, qui ne fait pas de suppositions sur le code source et qui n'a pas besoin de connaître les détails d'implantation de l'architecture, a des bonnes chances de générer des solutions de meilleure qualité, puisqu'elle élimine les faiblesses des approches traditionnelles. Elle permet de tenir compte automatiquement des interdépendances entre les problèmes. Aussi, en ne faisant aucune supposition sur le programme source, le compilateur pourra avoir des performances plus homogènes sur l'ensemble des programmes. Finalement, étant peu dépendant des caractéristiques de la machine, le compilateur peut s'adapter aux changements apportés à l'architecture de celle-ci. Dans la section suivante nous proposons une approche unifiée possédant ces propriétés.

4.2 L'idée de base

Nous proposons un compilateur qui conserve une structure classique : une partie frontale qui traduit le code source en un code intermédiaire, un optimiseur qui effectue des optimisations sur le code intermédiaire et une partie arrière qui génère le code objet à partir du code intermédiaire optimisé.

Dans notre approche l'optimiseur est basé sur un algorithme génétique (AG-Op). Notre AG-Op reçoit comme entrée le code intermédiaire généré par la partie frontale, un ensemble de caractéristiques de la machine (par exemple, le nombre de registres et le jeu d'instructions) et un critère d'optimisation. En se basant sur ces données, AG-Op fait évoluer le code intermédiaire (à la recherche de la meilleure solution) et fournit à la partie arrière le code intermédiaire optimisé.

La puissance de cette approche réside dans la façon d'effectuer les optimisations. Étant donné les bonnes performances des AGs, en général, pour résoudre des problèmes difficiles, notre intention est de faire résoudre par AG-Op non pas les problèmes séquentiellement mais tous en même temps, en tenant compte des interdépendances. De plus, nous pouvons résoudre avec AG-Op tous les problèmes que nous voulons à la seule condition qu'ils puissent être formulés au niveau du code intermédiaire. Ceci implique que la partie arrière de notre compilateur est plus simple que les parties arrières des compilateurs classiques, puisque davantage d'optimisations sont effectuées par l'optimiseur.

Un des facteurs déterminants pour l'obtention d'une bonne performance des AGs est l'adéquation de la fonction de coût à l'objectif visé. Plus la fonction de coût est bien adaptée, plus un AG a des chances de converger vers des solutions de bonne qualité. Par exemple, si nous voulons obtenir le code objet le plus petit possible, le nombre d'instructions du code objet est une fonction de coût appropriée. Par contre, si notre objectif est de générer le code objet qui a le temps d'exécution le plus petit possible, trouver une fonction de coût adéquate n'est pas simple car celle-ci dépend d'un grand nombre de caractéristiques matérielles (vitesse de la mémoire, profondeur des pipelines,

structure des unités de calcul, etc).

Il y a quelques critères qui sont souvent utilisés afin d'estimer la qualité du code objet. Par exemple, le nombre de *stalls* dans le *pipeline* ou le nombre de débordements en mémoire. Pour établir ces estimations il faut connaître quelques caractéristiques de l'architecture, comme la structure du *pipeline* et le nombre de cycle d'horloge que prend chaque instruction. Cependant, ces informations ne donnent pas une mesure précise. Nous proposons l'utilisation d'une mesure exacte de la qualité du code généré : le temps d'exécution réel du code objet.

L'utilisation du temps réel d'exécution comme fonction de coût offre quelques avantages intéressants. Premièrement, étant donné que notre objectif consiste à minimiser le temps d'exécution du code généré, le temps réel d'exécution est la mesure la plus précise que nous pouvons avoir. De plus, en utilisant cette fonction de coût, nous n'avons besoin que de très peu d'information sur l'architecture de la machine cible, puisque nous n'avons pas besoin d'établir des estimations sur le comportement de la machine.

Un autre avantage de l'utilisation du temps d'exécution est que nous mesurons l'effet réel des optimisations sur la machine cible. Ceci inclut les effets indirects des optimisations qui sont très difficiles à estimer, comme, par exemple, les défauts de pages ou des caractéristiques propriétaires de l'architecture.

Il faut noter que notre système est utilisable avec n'importe quel critère d'optimisation à condition de modifier la fonction de coût d'AG-Op afin de lui permettre d'optimiser le critère désiré.

Nous avons décrit AG-Op comme étant un optimiseur qui travaille sur le code intermédiaire mais, dans la pratique, il faut trouver une représentation du code intermédiaire qui puisse être manipulée par un AG et qui permet d'effectuer toutes les optimisations désirées. Cette représentation doit tenir compte des caractéristiques et des dépendances de tous les problèmes à résoudre. Une fois que cette représentation est déterminée, nous devons définir les opérateurs génétiques correspondants et leur façon d'agir sur la repré-

sentation.

4.2.1 La représentation

Il est possible de concevoir une représentation unique pour tous les problèmes que nous désirons résoudre mais, à notre avis, cette représentation serait trop complexe et les opérateurs génétiques correspondants aussi. De plus, il est fort probable que cette représentation admette beaucoup d'individus irréalisables. La complexité générale de l'AG augmenterait grandement à cause de la représentation.

Par exemple, si nous voulons résoudre les problèmes de sélection d'instructions, d'allocation de registres et d'ordonnancement d'instructions, nous pouvons envisager la représentation suivante. Un individu est une chaîne de nœuds où chaque nœud a quatre éléments :

1. l'instruction machine à utiliser,
2. l'opérande de destination,
3. le premier opérande source,
4. le deuxième opérande source.

L'ordre dans lequel les nœuds apparaissent dans l'individu représente l'ordonnancement d'instructions.

Il est évident qu'avec cette représentation, l'espace de recherche est très grand et qu'il y a beaucoup d'individus irréalisables. De plus, des opérateurs génétiques qui permettent de maintenir la réalisabilité des individus et/ou de maintenir de bonnes caractéristiques d'une génération à l'autre ne sont raisonnablement pas possibles tout en maintenant la simplicité de l'AG.

Enfin, le principal problème avec ce type de représentation est que si nous voulons ajouter une nouvelle optimisation, il faudra redéfinir autant la représentation que les opérateurs. Alors, nous jugeons plus simple et plus naturel de définir un individu comme

la concaténation de n sous-individu, un par problème. Par exemple, si nous voulons résoudre les problèmes de sélection d'instructions, d'allocation de registres et d'ordonnancement d'instructions, nous devons définir des individus capables de représenter les solutions à chaque problème et après nous les concaténons tous pour former l'individu final, tel que montré à la figure 4.1.

Afin de choisir une représentation pour chaque problème, il faut tenir compte des dépendances entre ces derniers. Par exemple, supposons que nous avons une solution pour le problème de sélection d'instructions et une solution pour le problème d'allocation de registres, chacune réalisable et représentée dans un sous-individu. Il est possible que, étant donné l'interdépendance entre les problèmes, cette sélection d'instructions ne soit pas compatible avec cette allocation de registres, et pourtant l'individu formé par la concaténation de ces deux sous-individus est irréalisable. Donc, au moment de définir la représentation pour chaque problème il faut concevoir un mécanisme pour palier cet inconvénient.

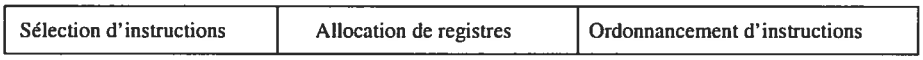


FIG. 4.1 – Exemple d'individu utilisé dans plusieurs optimisations

4.2.2 Les opérateurs génétiques

Les opérateurs de croisement et de mutation sont dépendants de la représentation du problème et ils doivent être définis en tenant compte des caractéristiques de chaque problème. Par conséquent, ils agiront de façon indépendante sur chaque sous-individu.

Par contre, l'opérateur de sélection agit sur l'individu au complet, car ce qui nous intéresse est la mesure de qualité d'un individu composé de solutions à tous les sous-problèmes à la fois. Aussi, nous utilisons la roulette pondérée, telle que définie au chapitre 2.

4.2.3 La fonction de coût

Comme nous l'avons déjà mentionné, la fonction de coût est le temps réel d'exécution du code objet. Pour mesurer ce temps, il faut passer à la partie arrière le code intermédiaire représenté par l'individu afin que le code objet correspondant soit généré. Après le code objet est exécuté et son temps d'exécution est mesuré.

La figure 4.2 présente la structure d'AG-Op. La population est formée d'individus où chaque individu est composé de solutions aux problèmes à résoudre. À chaque génération, l'opérateur de sélection choisit des individus en fonction de leur qualité. Ensuite, chaque individu est subdivisé afin que chaque sous-individu soit traité par les opérateurs correspondants. Finalement, l'individu est reconstruit et il est évalué par la fonction de coût.

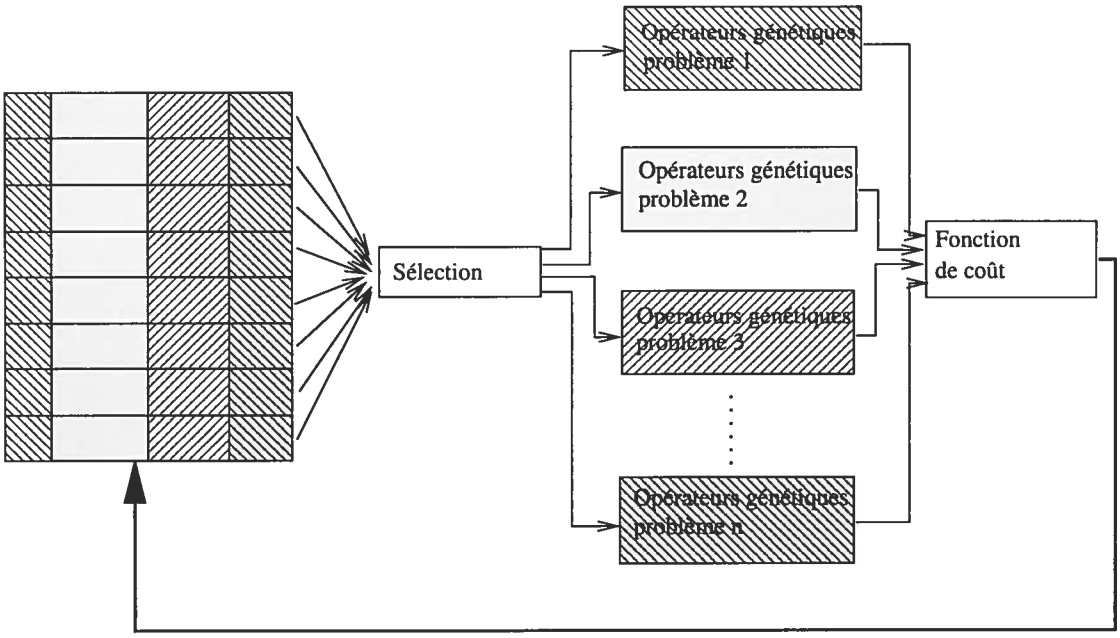


FIG. 4.2 – Structure de notre AG-Op

Pour utiliser notre approche il faut :

1. Déterminer les problèmes à résoudre,

2. définir la représentation des solutions à chaque problème en tenant compte des interdépendances,
3. définir les opérateurs génétiques,
4. établir la fonction de coût,
5. implanter AG-Op et l'intégrer au compilateur.

4.3 La conception

Afin d'utiliser AG-Op nous devons, premièrement, décider quels problèmes seront abordés. Nous avons choisi les problèmes d'allocation de registres et d'ordonnancement d'instructions puisqu'ils ont un impact important dans la qualité du code généré.

Normalement, le problème d'ordonnancement d'instructions a deux instances différentes. Dans la première, l'ordonnancement est réalisé avant l'allocation de registres. Étant donné qu'à ce moment le code intermédiaire suppose l'existence d'un nombre infini de registres, il y a seulement des dépendances réelles entre les instructions et, donc, l'ordonnancement a plus de flexibilité pour déplacer les instructions. Mais il peut trop étendre l'intervalle de vie de quelques variables, nuisant ainsi à l'allocation de registres. Dans la deuxième, l'ordonnancement est réalisé après l'allocation de registres. L'allocation de registres ajoute des dépendances entre les instructions données par la réutilisation de registres et ajoute, aussi, les instructions de chargement et de stockage nécessaires pour déborder les variables vers la mémoire. L'ordonnancement cherche à déterminer un meilleur ordre d'instructions en considérant les nouvelles caractéristiques.

Si l'ordonnancement est réalisé avant l'allocation seulement, les instructions de stockage et de chargement ajoutées par l'allocation de registres ne seront pas considérées. Ces instructions prennent un grand nombre de cycles d'horloge pour être exécutées puisqu'elles font accès à la mémoire. Donc, l'emplacement de ces instructions a un impact important sur la vitesse du code généré.

Par contre, si on réalise l'ordonnancement après l'allocation de registres, le placement des instructions de stockage et de chargement sera amélioré mais l'algorithme d'ordonnancement aura moins de flexibilité puisque les variables seront déjà assignées aux registres et, donc, il y aura des dépendances additionnelles.

Une approche très répandue pour aborder ce problème est de réaliser l'ordonnancement d'instructions deux fois, avant et après l'allocation de registres. Le principal désavantage de cette approche est le même que pour l'ordonnancement avant l'allocation. Il peut trop étendre les intervalles de vie nuisant à l'allocation de registres.

Une approche alternative est de réaliser un ordonnancement unique en même temps que l'allocation. Pour le faire il faut trouver une représentation unique pour les deux problèmes comme celle décrit au début de la section 4.2.1, au prix de toutes les difficultés décrites dans cette section.

Nous optons pour la réalisation d'un ordonnancement d'instructions après l'allocation de registres. Ce choix est basé premièrement sur le fait que l'ordonnancement avant l'allocation peut avoir un impact négatif pour l'allocation et, deuxièmement, sur des raisons techniques. Le compilateur que nous utilisons pour implanter AG-Op ne permet pas de faire un ordonnancement avant l'allocation (voir la section 4.4).

Nous concevons ici le modèle décrit dans la section précédante destiné à résoudre les deux problèmes choisis. Dans cette section, nous décrivons le choix de la représentation du problème, des opérateurs génétiques et de la fonction de coût.

Le problème d'allocation peut être modélisé comme un problème de coloriage de graphe, tandis que le problème d'ordonnancement d'instructions est similaire à d'autres instances du problème d'ordonnancement. Ces problèmes ont été étudiés et résolus à l'aide des méthodes métaheuristiques, ce qui nous permet de s'appuyer sur les résultats de ces recherches pour définir de la représentation du problème et les opérateurs génétiques.

4.3.1 La représentation pour l'allocation de registres

La transformation du problème d'allocation de registres en un problème de coloriage de graphe est direct. Il suffit de considérer les équivalences suivantes.

- Le graphe à colorier est le graphe d'interférence du problème d'allocation.
- Le nombre de couleurs est le nombre de registres disponibles.
  Dans certains cas, quelques variables peuvent être assignés seulement à un sous-ensemble de registres. Cette situation se présente lorsqu'il existe différents types de registres ou lorsque les instructions ont besoin de registres spécifiques. Pour éviter une assignation non désirée il suffit d'ajouter dans le graphe d'interférence un nœud pour chaque registre et ajouter des arêtes entre chaque variable et les registres auxquels elle ne peut pas être assignée. De plus il aura des arêtes entre tous les nœuds qui représentent les registres.

Nous trouvons dans la littérature [FF97, EvdH96, Dav91] deux représentations pour le problème de coloriage de graphe qui s'adaptent parfaitement au problème d'allocation et que nous appelons *représentation directe* et *indirecte*.

- **Représentation directe** : (décrite et utilisée dans le chapitre 3) l'individu est une chaîne a de longueur L, contenant des entiers entre -1 et n-1, où L est le nombre de variables présentes dans le problème d'allocation et n est le nombre de registres de la machine. La valeur a_i de l'individu représente le registre physique auquel la variable i est assignée. Si la valeur est -1, cela signifie que la variable est assignée à la mémoire.
- **Représentation indirecte** : l'individu est une chaîne de longueur L. Chaque élément de l'individu représente une variable. L'AG utilise une fonction heuristique qui assigne les variables aux registres dans l'ordre dans lequel elles apparaissent dans l'individu. La fonction traite séquentiellement les variables dans l'ordre ou elles apparaissent dans l'individu, et lorsque les interférences entre les variables le permettent, elle leur assigne un registre. Sinon la variable est assignée à la mémoire.

Une variante de la représentation indirecte peut être définie comme suit. Les valeurs de l'individu peuvent être positives ou négatives. La fonction heuristique considère seulement les valeurs positives pour trouver l'allocation, les variables représentées par des de valeurs négatives sont assignées directement à la mémoire. Ainsi, c'est l'AG qui détermine de façon directe quelles variables seront assignées à la mémoire et lesquelles seront assignées à des registres.

La fonction heuristique utilisée par l'AG dans la représentation indirecte est vorace. Cette heuristique suppose que chaque couleur a un numéro. Elle considère les variables une à une dans l'ordre donné et leur assigne une couleur faisable. Elle choisit la couleur en cherchant de manière séquentielle en commençant par la couleur ayant le plus petit numéro. Évidemment, puisque le nombre de couleurs (registres) est limité, lorsqu'il n'est pas possible d'assigner une couleur à une variable, celle-ci est assignée à la mémoire.

Ces deux représentations permettent de modéliser parfaitement le problème d'allocation de registres. La principale différence est donnée par le fait que la fonction heuristique de la représentation indirecte cherche toujours à assigner le nombre minimal de registres pour un ordre de nœuds donné. Ceci peut ne pas être l'idéal pour le problème d'allocation de registres. Si le nombre de variables est plus grand que le nombre de registres, minimiser le nombre de registres utilisés ne pose aucun problème (c'est même une bonne idée) puisque ceci permet d'assigner plus de variables aux registres. Par contre, si on a plus de registres que de variables, utiliser le nombre minimal de registres n'est pas nécessaire. Étant donné que la réutilisation de registres ajoute des dépendances entre les instructions, s'il y a assez de registres, il n'y a pas de raison d'ajouter des dépendances qui vont limiter l'ordonnancement. En considérant l'allocation interprocédurale, même s'il y a assez de registres pour une procédure, il est toujours convenable d'essayer de minimiser le nombre de registres et donc, on a plus de registres disponibles pour les autres procédures. Donc, à notre avis, l'utilisation de la représentation indirecte est tout à fait raisonnable, même si, dans quelques cas, elle ajoute des dépendances non-nécessaires.

Dans la section 4.3.4, nous choisissons entre ces deux représentations en considérant la dépendance entre le problème d'allocation de registres et d'ordonnancement d'instructions.

La figure 4.3 présente un exemple de programme et une instance de chacune des représentations décrites, en supposant une machine à cinq registres.

L'individu de la figure 4.3(b) utilise la représentation directe, c'est-à-dire, l'assignation des variables aux registres est directement indiquée dans l'individu. La représentation de la figure 4.3(c) correspond à la représentation indirecte. Donc, pour trouver l'allocation il faut appliquer l'heuristique vorace en considérant les nœuds dans l'ordre de l'individu. Finalement, la figure 4.3(d) présente la représentation indirecte modifiée. Dans l'exemple la variable b est négative, alors, au moment d'appliquer l'heuristique vorace pour générer l'allocation, la variable b ne sera pas considérée et sera débordée directement à la mémoire.

4.3.2 La représentation pour l'ordonnancement d'instructions

Pour le problème d'ordonnancement, la représentation la plus directe consiste en une permutation des numéros de 1 à N, où N est le nombre d'instructions du programme. L'ordre des numéros dans l'individu représente l'ordre dans lequel les instructions seront exécutées. Cependant, les permutations ne sont pas toutes valides, puisqu'il faut tenir compte de la préséance entre les instructions. Il existe trois façons d'attaquer ce problème.

1. Utiliser un critère de pénalité, c'est-à-dire, permettre la génération des solutions irréalisables mais leur assigner un coût très élevé pour éviter que l'AG ne converge vers ces solutions. Cette approche est simple mais pas toujours efficace. Généralement, une approche par pénalité demande d'utiliser des tailles de population très grandes, ce qui se traduit en plus de temps de traitement. Elle est viable malgré tout dans les cas où les solutions irréalisables sont peu probables, ce qui n'est pas

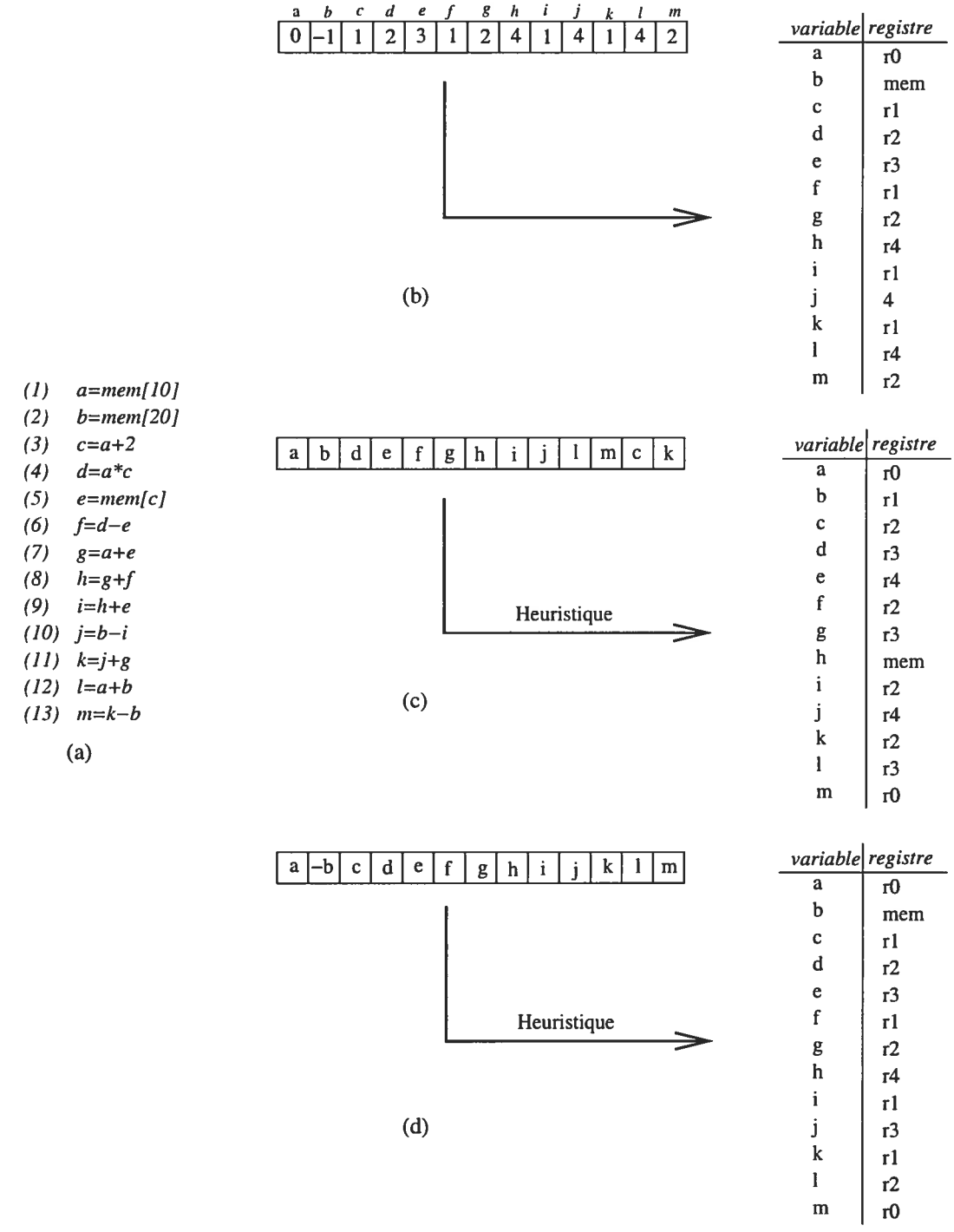


FIG. 4.3 – Exemple de programme et trois représentations possibles pour l'allocation

le cas de l'ordonnancement d'instructions.

2. Utiliser des opérateurs génétiques qui garantissent la réalisabilité des individus générés.
3. Permettre la génération des solutions irréalisables et utiliser un *opérateur de réparation* (opérateur qui transforme une solution qui n'est pas réalisable en une qui l'est) avant d'effectuer l'évaluation.

Pour des raisons techniques, expliquées dans la section 4.3.6, l'ordonnancement des instructions est effectué sur chaque bloc de base du programme, l'individu représentant le problème d'ordonnancement sera constitué de n sous-individus, un par bloc de base.

La figure 4.4 présente des exemples d'ordonnancement réalisables et irréalisables pour le programme de la figure 4.3(a). L'irréalisabilité de l'ordonnancement à la figure 4.4(b) est causée par le fait que l'instruction 5 ne peut pas être exécutée avant l'instruction 3, car il existe une dépendance de données entre elles.

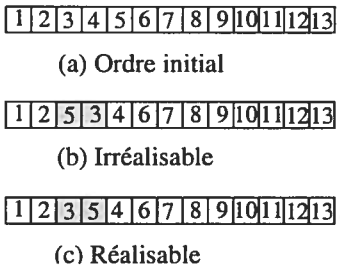


FIG. 4.4 – Exemple d'ordonnancement réalisable et irréalisable

4.3.3 Les opérateurs génétiques

Les opérateurs de croisement (OBX et PBX) décrits dans le chapitre 2 sont applicables autant au problème d'allocation qu'à celui d'ordonnancement. Lors de l'utilisation de ces opérateurs pour le problème d'ordonnancement une **fonction de réparation** doit être utilisée pour transformer les individus irréalisables en réalisables, en les modifiant le

moins possible. Cette fonction parcourt chaque élément de l'individu en vérifiant si l'instruction peut être exécutée en ce moment. Si l'instruction est prête à être exécutée, elle est assignée, si non, elle est mise dans une liste d'attente et elle est assignée dès que possible. Le pseudocode pour la fonction de réparation se trouve à l'annexe C. Par ailleurs, nous avons développé un opérateur de croisement pour le problème d'ordonnancement qui ne génère pas d'individus irréalisables.

Décrivons le croisement spécifique à l'ordonnancement (SSX). Un sous-ensemble des positions est sélectionné chez le premier parent, les éléments trouvés à ces positions sont stockés dans une liste L1 et les éléments non-sélectionnés dans une liste L2 dans l'ordre auquel ils apparaissent chez le deuxième parent. L'enfant est formé en prenant un élément de chaque liste en alternance. Si l'instruction représentée par l'élément choisit peut être exécutée à ce moment (toutes les instructions précédentes ont été déjà assignées), nous assignons cet élément à la prochaine position chez l'enfant. Sinon, nous essayons avec le premier élément de l'autre liste. Un exemple du fonctionnement de cet opérateur est présenté à la figure 4.5.

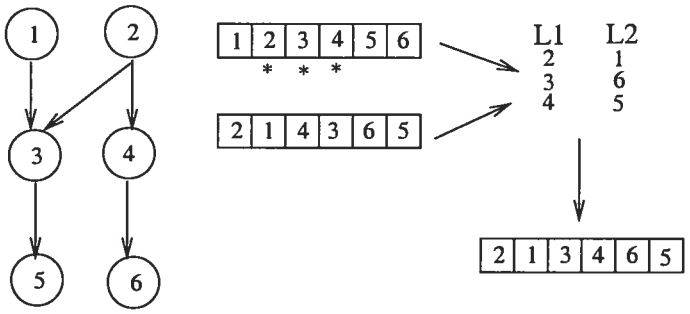


FIG. 4.5 – Opérateur spécifique pour l'ordonnancement

Si nous réalisions l'ordonnancement avant l'allocation de registres, l'opérateur SSX serait sans doute le meilleur candidat puisqu'il ne génère pas d'individus irréalisables. Mais, en considérant l'ordonnancement d'instructions après l'allocation de registres, l'utilisation de SSX perd son sens, puisque les solutions réalisables générées pour SSX peuvent cesser de l'être chaque fois que l'allocation de registres est modifiée (à chaque généra-

tion). Alors, nous choisissons un opérateur de croisement plus classique (PBX ou OBX) et la fonction de réparation pour traiter les individus irréalisables.

Une évaluation empirique qui est présentée au prochain chapitre nous permet de choisir judicieusement les opérateurs de croisement OBX ou PBX pour les deux problèmes.

Les opérateurs de mutation pour l'allocation de registres et pour l'ordonnancement d'instructions consistent en une opération d'échange des éléments entre deux positions choisies aléatoirement.

4.3.4 Le meilleur choix

Si notre but consistait à résoudre les problèmes d'allocation et d'ordonnancement séparément, n'importe quelle des représentations mentionnées serait appropriée. Cependant, notre objectif consiste à les résoudre simultanément. Alors, en choisissant la représentation, nous devons tenir compte de la dépendance entre les problèmes. Nous considérons, aussi, la flexibilité permettant d'ajouter d'autres optimisations (comme l'ordonnancement avant l'allocation ou la sélection d'instructions).

Nous cherchons à former un individu (concaténation des sous-individus pour chaque problème) qui soit toujours réalisable. Ainsi tous les individus dans la population pourront être évalués.

Les problèmes d'allocation de registres et d'ordonnancement d'instructions sont très interdépendants. Ainsi, en modifiant l'allocation, le nombre d'instructions du programme et leurs dépendances changent et les ordonnancements valides ne sont plus les mêmes. De même, si nous ajoutons un ordonnancement avant l'allocation de registres, les modifications qu'il introduit pourraient changer les interférences entre les variables, donc une allocation qui était valide pourrait ne plus l'être et, par conséquent, l'individu serait irréalisable.

Pour l'allocation des registres, nous avons choisi la représentation indirecte modifiée décrite à la section 4.3.1, c'est-à-dire, celle où l'individu contient l'ordre des variables (numéros positifs ou négatifs) et une fonction heuristique détermine l'allocation. Cette représentation nous donne la flexibilité nécessaire pour ajouter un ordonnancement avant l'allocation de registres. Il suffit que les interférences soient recalculées après avoir réalisé le premier ordonnancement et que la fonction heuristique tienne compte de ces nouvelles dépendances pour déterminer l'allocation.

Pour l'ordonnancement, nous choisissons la représentation des individus contenant l'ordre des instructions et utilisant un opérateur de réparation pour s'occuper des irréalisables. Ainsi, la fonction de réparation tient compte des dépendances créées par l'allocation actuelle.

La figure 4.6(a) présente le programme de l'exemple introduit à la figure 4.3 affecté par une allocation de registres et un ordonnancement valides. En modifiant l'allocation (4.6(b)), l'ordonnancement n'est plus valide, puisque l'instruction 7 ne peut pas être exécuté avant l'instruction 4 (puisque dans la nouvelle allocation, les deux instructions ont le même registre de destination). Donc, il est nécessaire d'utiliser la fonction de réparation.

La modification de l'allocation introduit un problème additionnel : en modifiant l'allocation nous ne modifions pas seulement le graphe de dépendances, mais aussi les instructions elles-mêmes. Lorsqu'une variable se fait déborder vers la mémoire, des instructions de chargement et de stockage sont générées. Autrement, elles n'apparaissent pas. Pour faire face à cette difficulté, nous introduisons une généralisation de la représentation choisie pour le problème d'ordonnancement. Nous utilisons des individus de longueur fixe L, où L est le nombre d'instructions en assignant toutes les variables à la mémoire, c'est-à-dire que L est la longueur maximale. Les instructions sont numérotées avec deux séquences distinctes. La première représente, à l'aide des nombres de 0 à N, les instructions qui ne dépendent pas de l'allocation et qui doivent être présentes peu importe l'allocation. La deuxième représente, à l'aide des nombres plus grands que M (où

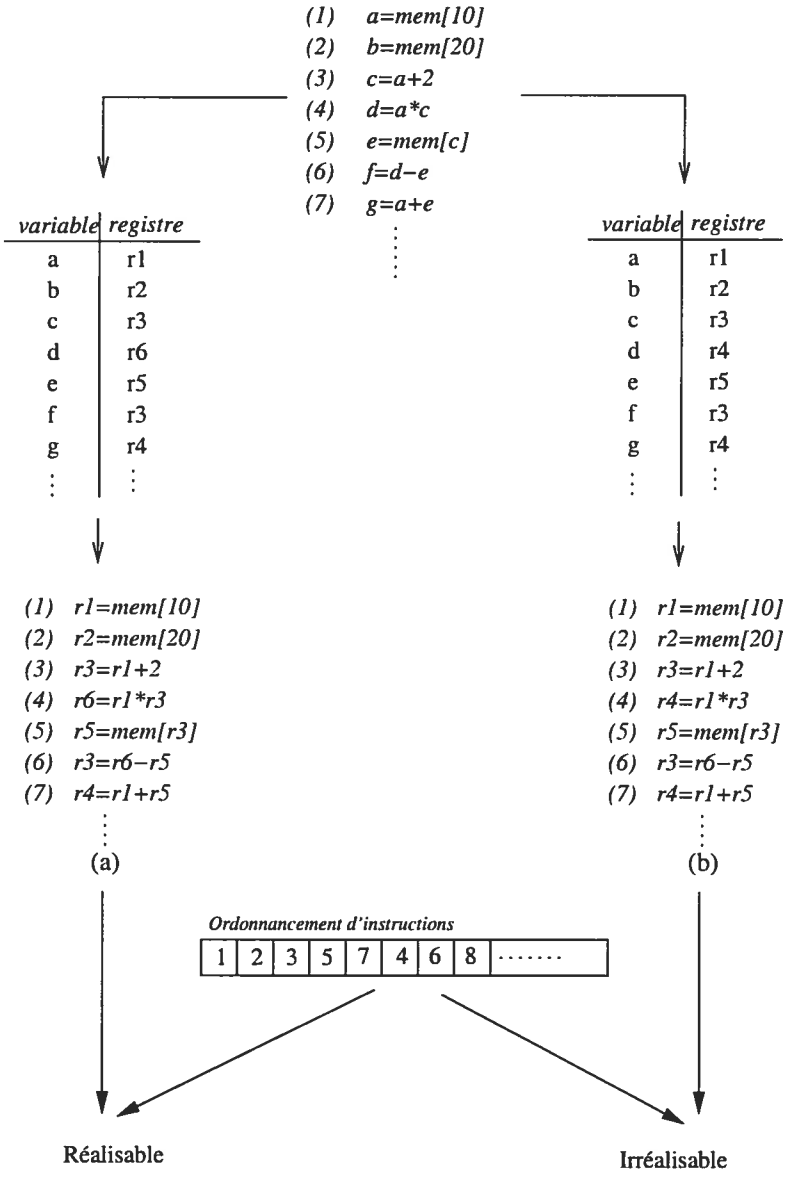


FIG. 4.6 – Exemple d'allocation et d'ordonnancement

$M > N$), les instructions qui dépendent de l'allocation (instructions de déplacement, de stockage et de chargement). Ainsi, nous pouvons éliminer sans difficulté les instructions qui ne sont plus nécessaires après une modification de l'allocation sans changer l'ordre des instructions conservées. Donc, le processus d'évolution n'est pas perturbé puisque l'ordre relatif des instructions dans l'individu est conservé. La figure 4.7 présente un exemple d'un individu dénoté selon cette méthode.

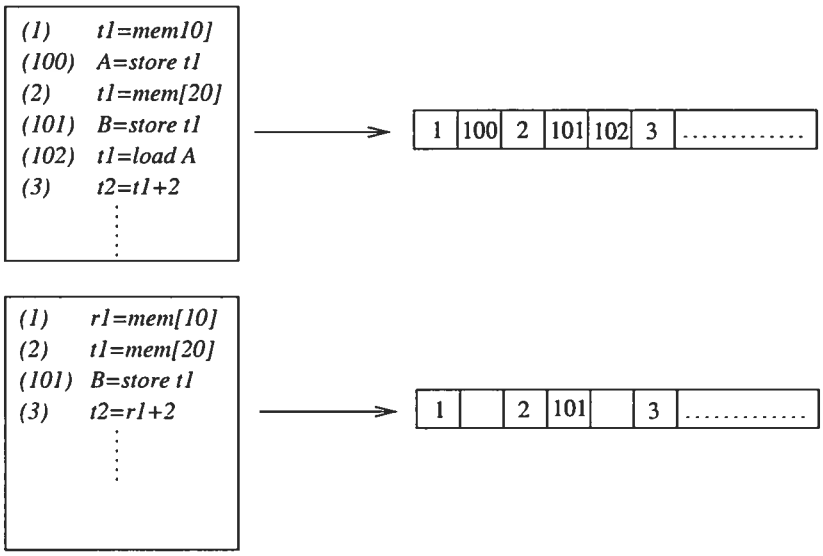


FIG. 4.7 – Individu décrivant l'ordonnancement d'instructions

Avec les représentations choisies pour les deux problèmes, tous les individus (concaténation des individus d'allocation de registres et d'ordonnancement d'instructions) permettent de générer du code objet correct et donc, résoudre les problèmes conjointement.

4.3.5 La fonction de coût

Afin de mesurer la qualité de chaque individu, nous devons générer le programme objet décrit par l'individu (allocation et ordonnancement), l'exécuter et mesurer le temps d'exécution réel. Nous testons deux méthodes pour mesurer le temps d'exécution :

- **externe** : en utilisant un appel au système d'exploitation qui détermine le temps employé par le programme ;
- **interne** : en mesurant le nombre de cycles d'horloge utilisés par le programme. Pour ce faire, il faut instrumenter le programme objet avec des instructions destinées à la collecte des mesures. Des altérations au programme objet ne sont pas désirables, puisque l'AG optimise ainsi un programme qui n'est pas exactement le même que celui à compiler. Cependant, cette approche risque de fournir des mesures plus exactes qu'un appel système.

Suite à une évaluation numérique des deux méthodes, nous avons pu constater que la différence n'est pas importante et nous avons décidé d'utiliser l'appel système car il ne modifie pas le code. La figure 4.8 illustre le fonctionnement de notre système.

Plusieurs exécutions d'un même programme ne donnent pas toujours le même temps d'exécution. Ces différences sont dues aux erreurs de mesures ou aux différences dans l'exécution du programme dû aux caractéristiques non-prévisibles de la machine (comme la mémoire cache). Dans la plupart des cas les différences sont petites, mais parfois ces différences peuvent être importantes, ce qui peut poser un problème. Supposons qu'AG-Op exécute un programme qui n'est pas de bonne qualité. Si, par hasard, il obtient un bon temps d'exécution, AG-Op considérera que cette solution est bonne quand en réalité elle ne l'est pas.

Intuitivement, nous croyons qu'AG-Op possède naturellement le mécanisme pour faire face à ce problème. Supposons que, effectivement, AG-Op se trompe et prend une solution de mauvaise qualité pour bonne. Cette solution aura de bonnes chances de passer à la génération suivante (ou de passer ses caractéristiques aux enfants), alors elle (ou les enfants avec ses caractéristiques) sera évaluée à nouveau. La probabilité de qu'encore une fois ce programme donne un temps d'exécution qui sorte de la moyenne est petite, donc, la solution ne survivra pas à la prochaine génération.

Étant donné l'importance de la fonction de coût, nous ne pouvons pas nous satisfaire de notre intuition. Une approche plus précise serait de faire 5 exécutions du programme dans chaque évaluation, éliminer le minimum et le maximum et considérer la moyenne des 3 exécutions qui restent. Clairement, cette approche élimine le problème mais augmente considérablement le temps de traitement requis pour AG-Op. Donc, si notre intuition est vraie, il n'y a pas de sens à utiliser cette approche.

À la section 5.1.1, nous réaliserons des tests en considérant les deux approches (5 exécutions et 1 exécution) et nous déciderons laquelle utiliser en fonction des résultats numériques. Ces tests confirment notre intuition.

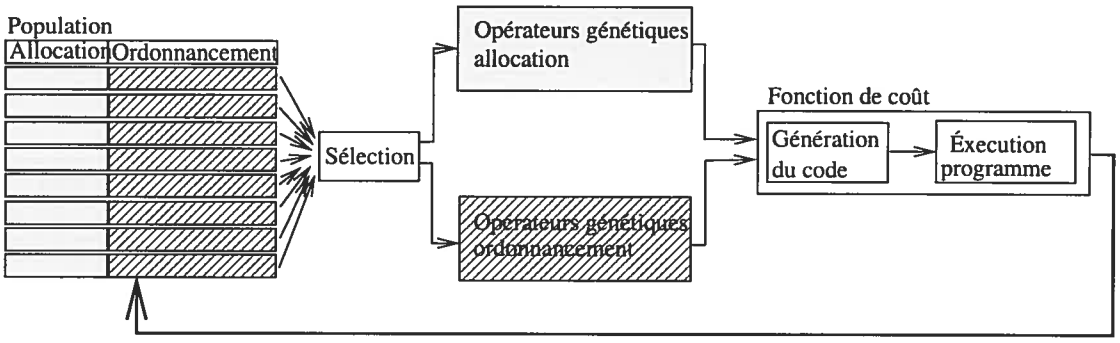


FIG. 4.8 – Structure d'AG-Op pour les problèmes d'allocation de registres et d'ordonnancement d'instructions

4.3.6 Le compilateur

Afin d'implanter AG-Op, il faut un compilateur qui génère un code intermédiaire contenant plusieurs informations (ou à partir duquel nous pouvons obtenir ces informations). Par exemple, si nous nous intéressons aux problèmes d'allocation de registres et d'ordonnancement, nous avons besoin du graphe d'interférence des variables, des registres utilisables par chaque variable et du graphe de dépendances des instructions. Idéalement, le graphe d'interférence doit être disponible pour le programme complet, afin permettre à AG-Op de réaliser une allocation globale.

De plus, le compilateur doit nous permettre d'identifier facilement les instructions introduites par un débordement vers la mémoire et les instructions requises par le programme indépendamment de l'allocation de registres.

La partie arrière doit être simple, c'est-à-dire, elle ne doit pas faire les optimisations effectuées par AG-Op, parce que cela serait redondant ou risquerait d'atténuer le niveau d'optimisation d'AG-Op.

Nous avons besoin de connaître le nombre de registres de la machine et leurs caractéristiques. Il faut noter qu'étant donné que nous utilisons le temps réel d'exécution comme fonction de coût, nous n'avons besoin d'aucun autre renseignement sur les caractéristiques de la machine cible. Par contre, si nous voulions utiliser une autre fonction de coût qui dépendrait, par exemple, du nombre de *stalls* dans le *pipeline* et du nombre d'accès à mémoire, nous aurions besoin de données additionnelles nous permettant de faire ce calcul.

La construction d'un tel compilateur dépasse les objectifs de notre travail. Nous avons opté pour la modification d'un compilateur existant qui s'adapte assez bien à nos besoins. Nous utilisons le compilateur *lcc*, un compilateur C multi-plate-forme (*multitarget*) assez performant écrit par Hanson et Fraser [FH95]. Sa partie arrière très simple ne fait pas d'ordonnancement d'instructions et son allocateur de registre (allocateur à la volée) peut être remplacé facilement [FH92]. Malheureusement, il compile séparément les fonctions, ce qui nous oblige à faire une allocation de registres intraprocédurale au lieu d'une allocation globale. De plus, il gère les blocs de base séparément, ce qui nous oblige à réaliser l'ordonnancement d'instructions à niveau de bloc de base.

Lcc utilise un allocateur à la volée très simple qui gère uniquement les variables temporaires. Comme les intervalles de vie des variables temporaires sont généralement petits et qu'il y a peu de variables temporaires, les débordements vers la mémoire sont rares et l'allocateur obtient normalement une allocation optimale des registres.

Lcc traite les variables de l'usager de façon différente selon l'architecture de la ma-

chine cible. Par exemple, en compilant pour la famille du X86, il met toutes les variables de l'usager en mémoire. Pour les autres architectures, *lcc* assigne tout simplement les n variables les plus référencées à un registre (sans tenir compte du graphe d'interférence), où n est le nombre de registres disponibles dans la machine pour ce type de variables. Comme cette assignation est faite très tôt dans la compilation, il n'est pas possible de faire un ordonnancement avant l'allocation de registres.

Étant donnée que *lcc* effectue l'allocation des variables d'usager et les temporaires de façon différente et en étapes séparées, nous ne pouvons pas traiter les deux types de variables en même temps. Les variables temporaires ne générant pas beaucoup de conflits, nous effectuons seulement l'allocation des variables usager.

Notons que ceci n'est pas optimal, puisqu'il serait mieux de pouvoir faire l'allocation de toutes les variables en même temps. Mais, étant donné que *lcc* fait une allocation très naïve pour les variables usager, une amélioration à ce niveau pourrait avoir un impact important sur la qualité du code généré.

Nous divisons les registres disponibles en deux ensembles disjoints, l'un qui est utilisé par *lcc* pour allouer les variables temporaires et l'autre qui est utilisé par notre AG-Op.

4.4 L'implantation

Nous implantons l'algorithme génétique décrit à la section précédente avec le langage C. Quelques détails importants sur les structures de données utilisées et sur les principales procédures sont donnés à l'annexe C.

Si nous disposions d'un compilateur parfaitement adapté à nos besoins, l'implantation serait directe. Il suffirait de recueillir le code intermédiaire produit par le compilateur, de le modifier en utilisant l'AG et de générer le code objet à partir du nouveau code intermédiaire. Donc, notre système serait aussi simple que celui illustré à la figure 4.9.

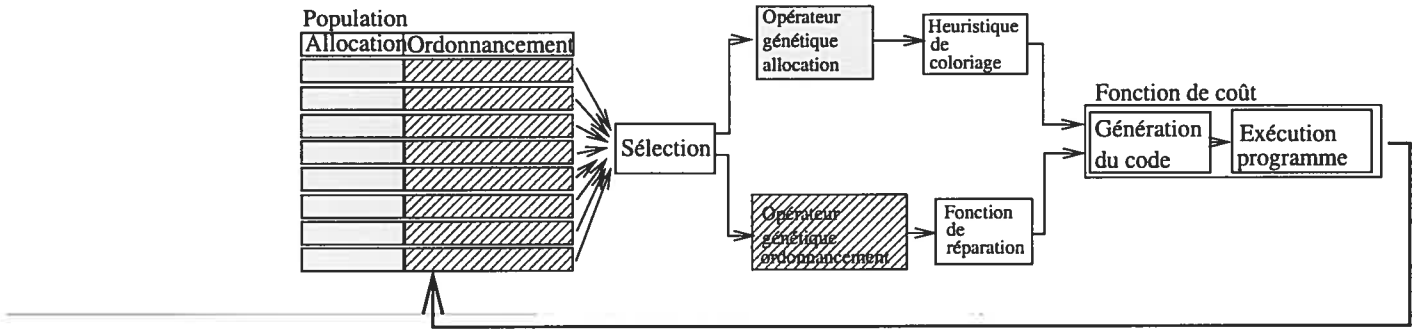


FIG. 4.9 – Structure d'AG-Op avec un compilateur idéal

Comme nous l'avons dit, *lcc* s'adapte assez bien à nos besoins mais il est loin d'être le compilateur idéal. Donc, nous devons nous débrouiller pour adapter *lcc*. Les modifications nécessaires ne sont pas toujours simples et compliquent l'implantation de notre approche.

Nous avons déjà mentionné quelques concessions nécessaires pour pouvoir travailler avec *lcc* :

- l'allocation de registres sera réalisé dans chaque procédure (allocation intraprocédurale) ;
- l'allocation de registres sera appliquée seulement aux variables de l'usager ;
- l'ordonnancement d'instructions sera effectué sur chaque bloc de base ;
- l'ordonnancement d'instructions sera réalisé seulement après l'allocation de registres.

Notre AG-Op requiert le graphe d'interférence entre les variables pour chaque procédure du programme et le graphe de dépendances entre instructions pour chaque bloc de base.

L'allocateur de *lcc* est très simple et n'utilise pas le graphe d'interférence pour guider l'allocation, alors nous devons construire ce graphe. De plus, *lcc* n'est pas conçu pour faire des analyses de flux. Il n'est donc pas évident de construire le graphe d'interférence ni le graphe de dépendances. Nous avons essayé d'obtenir ces données à partir de la

représentation intermédiaire de *lcc* mais celle-ci étant très particulière nous n'avons pas réussi. Cependant, en utilisant le code assembleur comme un genre de code intermédiaire, il a été possible d'obtenir le graphe d'interférence et de dépendances.

Nous avons modifié *lcc* pour qu'il ne fasse pas l'allocation de registres réelle et qu'il génère le code assembleur en supposant qu'il existe un nombre infini de registres. Nous analysons ensuite ce code assembleur pour obtenir le graphe d'interférence. Après, nous générons le code assembleur avec une allocation déjà définie et nous l'analysons afin d'obtenir le graphe de dépendances pour le problème d'ordonnancement d'instructions.

Nous pouvons considérer que nous avons quatre versions de *lcc* différentes :

lcc0 : génère le code assembleur en faisant déborder toutes les variables de l'usager en mémoire et ne fait pas d'ordonnancement,

lcc1 : génère du code assembleur avec un nombre infini de registres et ne fait pas d'ordonnancement,

lcc2 : génère le code assembleur pour une allocation de registres donnée,

lcc3 : génère le code objet pour une allocation de registres et un ordonnancement d'instructions donnés.

De plus, nous avons deux programmes qui, à partir du code assembleur, génèrent les informations suivantes :

Gen-Inter : génère le graphe d'interférence.

Gen-Dep : génère le graphe des dépendances pour chaque bloc de base du programme.

La figure 4.10 présente la structure d'AG-Op utilisé avec *lcc*. La population initiale est générée de façon aléatoire mais tout en considérant le code généré en pire cas (toutes les variables en mémoire et sans ordonnancement) afin de déterminer la longueur du sous-individu d'ordonnancement.

Lcc1 génère le code assembleur en supposant l'existence d'un nombre infini de registres. Ce code est traité par *Gen-Inter* qui obtient le graphe d'interférence. Les opérateurs génétiques sont appliqués. À partir du sous-individu de l'allocation, l'heuristique

de coloriage utilise le graphe d'interférence et l'individu pour obtenir une allocation valide. *Lcc2* génère le code assembleur en fonction de cette allocation et *Gen-Dep* analyse celui-ci afin d'obtenir les graphes de dépendances de chaque bloc de base. Ensuite, la fonction de réparation de l'ordonnancement reçoit les graphes de dépendance et le sous-individu de l'ordonnancement et génère un ordonnancement réalisable. Finalement, *lcc3* génère le code objet voulu.

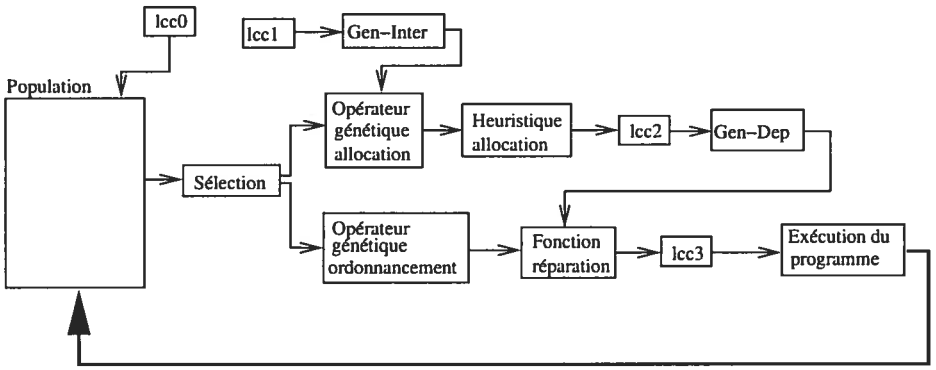


FIG. 4.10 – Structure d'AG-Op avec lcc

Il faut noter que le temps de traitement est plus élevé avec notre système que ce que nous pourrions obtenir en utilisant un compilateur qui génère déjà toutes les données dont nous avons besoin. Cependant, la qualité du code généré est la même et permet à notre implantation d'être validée quant à la performance du code généré.

4.5 Résumé

Dans ce chapitre nous avons défini une approche unifiée pour utiliser les métaheuristiques dans la compilation. Celle-ci permet de tenir compte de l'interdépendance des problèmes en les résolvant simultanément.

Notre approche utilise le temps réel d'exécution comme fonction de coût de la métaheuristique. Ceci permet de simplifier la construction des compilateurs puisque très peu de données sur les caractéristiques de l'architecture sont nécessaires. Pour la même

raison, un compilateur construit de cette façon pourra s'adapter facilement aux changements des architectures.

Notre approche utilise un optimiseur basé sur un AG qui effectue les différentes optimisations au niveau du code intermédiaire. Nous l'avons conçu pour résoudre les problèmes d'allocation de registres et d'ordonnancement d'instructions. Par ailleurs, nous avons défini des représentations pour les deux problèmes qui considèrent ces interdépendances.

Nous avons intégré notre optimiseur à *lcc* afin d'effectuer une analyse de performance.

Chapitre 5

Résultats

La première partie de ce chapitre présente les résultats expérimentaux qui nous ont permis de choisir les différents paramètres de notre AG-Op. La deuxième partie présente les résultats d'une analyse de performance sur notre système en le comparant avec des approches plus classiques pour les problèmes d'allocation de registres et d'ordonnancement d'instructions. Nous avons utilisé une machine SPARC pour nos expérience. Nous justifions ce choix par le fait que l'architecture SPARC n'effectue pas d'ordonnancement d'instructions dynamiquement, ce qui permet de faire une analyse plus directe des résultats.

5.1 Ajustement d'AG-Op

Nous avons quatre décisions à prendre sur les questions suivantes : combien de fois exécuter le programme dans la fonction de coût, quel est l'individu initial, quel est l'opérateur de croisement le plus performant et quels sont les paramètres que nous devons utiliser dans l'AG ? Pour chaque cas, nous effectuons une évaluation empirique du temps de convergence de l'AG et de la qualité de la solution. Dans presque tous les graphiques de cette section l'abscisse représente le coût de la solution, c'est-à-dire le temps d'exé-

cution du programme à compiler, et l'ordonnée indique le nombre de générations. Nous présentons, généralement, la moyenne de la population et le meilleur individu à chaque générations. Les évaluations ont été effectuées sur 3 programmes de test et les résultats sont semblables. Donc, dans la plupart des cas, nous présentons seulement les résultats pour un seul des programmes (*bj*). Les autres résultats sont à l'annexe D.

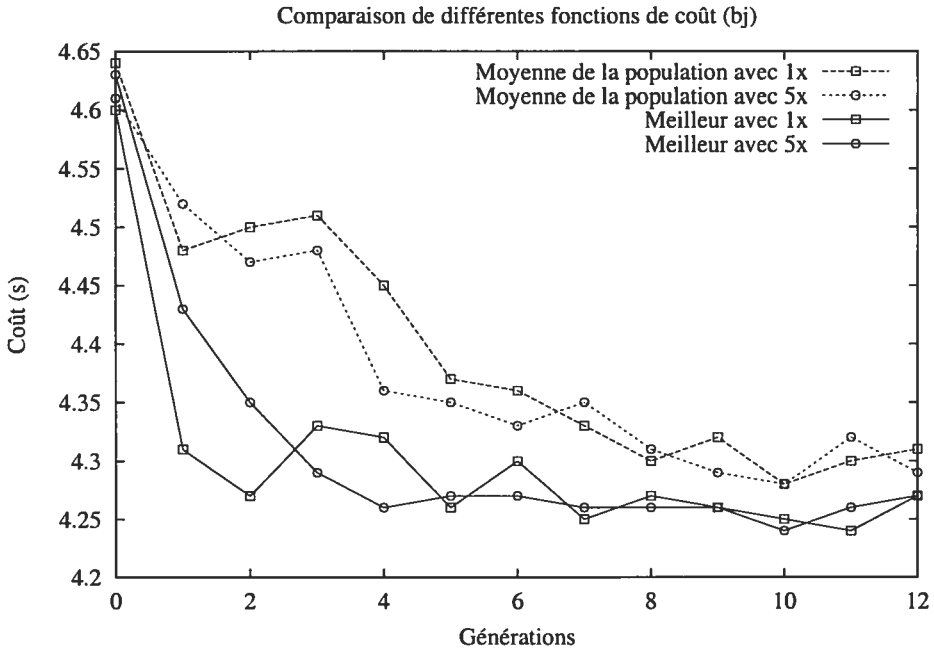


FIG. 5.1 – Convergence d'AG-Op avec les deux fonctions de coût (problème bj)

5.1.1 La fonction de coût

Comme nous avons expliqué dans le chapitre précédant, nous avons deux alternatives pour la fonction de coût.

5x : Mesurer le temps de cinq exécutions du programme, éliminer le minimum et le maximum et considérer comme coût la moyenne des trois qui restent.

1x : Prendre comme coût le temps d'une unique exécution du programme.

Les figures 5.1 et 5.2 présentent les courbes de convergence d'AG-Op en utilisant les deux alternatives sur deux programmes (*bj* et *tfftdp*). Notons que les résultats avec

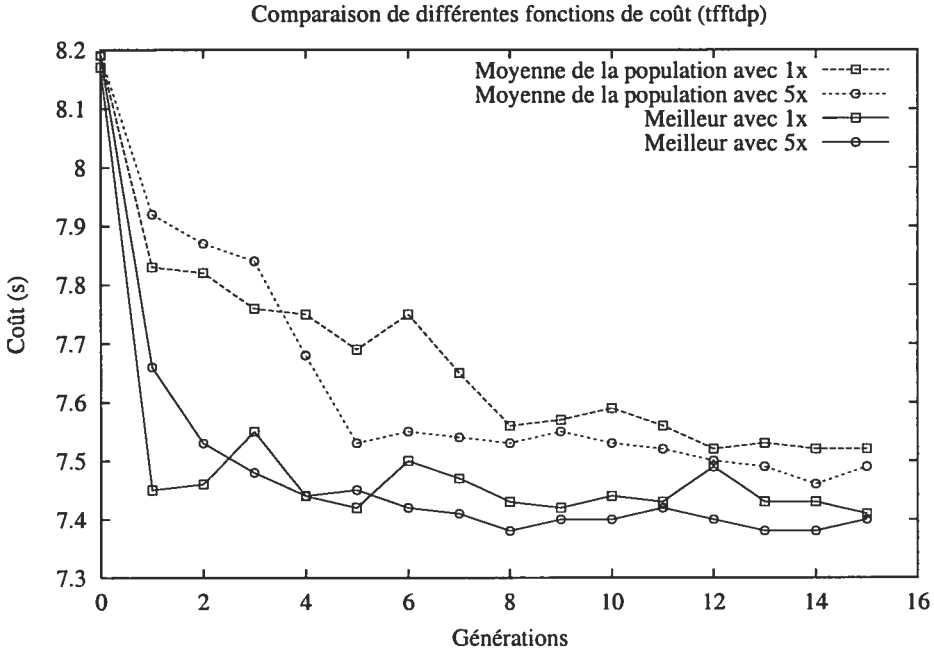


FIG. 5.2 – Convergence d'AG-Op avec les deux fonctions de coût (problème tfftdp)

5x et 1x ne sont pas beaucoup différents. Ainsi, étant donné le temps de traitement additionnel requis par 5x, nous ne croyons pas convenable de l'utiliser. Par conséquent, nous utilisons 1x pour effectuer tous les tests.

5.1.2 L'individu initial

L'individu initial que nous utilisons pour effectuer l'ordonnancement d'instructions représente l'ordre original du programme afin d'être sûr de la réalisabilité de l'ordonnancement initial. Quant à l'individu initial utilisé dans l'allocation de registres, nous testons trois alternatives :

Ini1 toutes les variables sont en mémoire ;

Ini2 n variables choisies au hasard sont assignées à des registres (où n est le nombre de registres) ;

Ini3 l'allocation obtenue à l'aide de l'algorithme de coloriage de graphes.

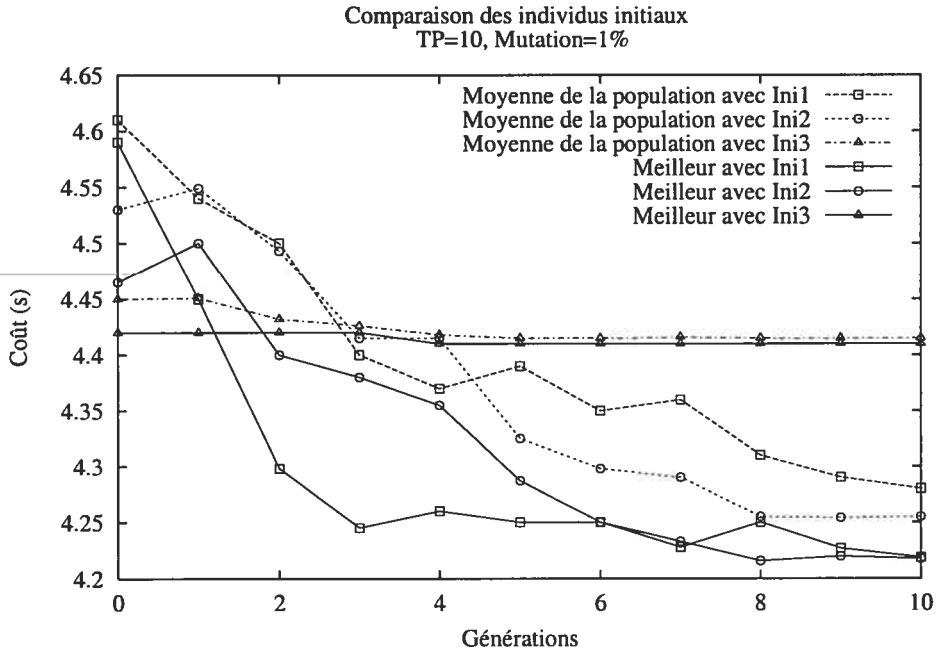


FIG. 5.3 – Convergence d'AG-Op avec les trois individus initiaux (bj)

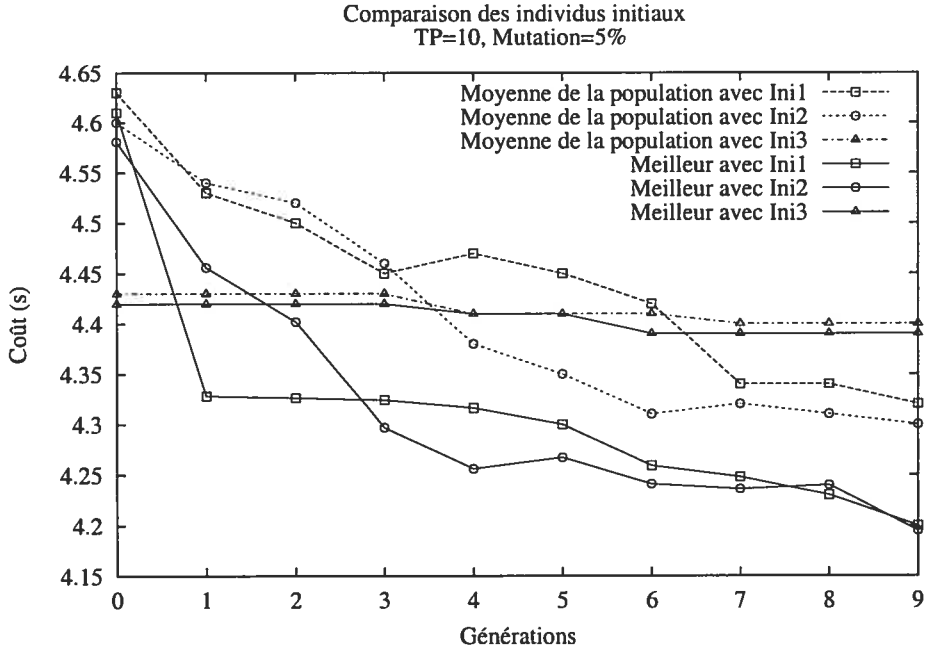


FIG. 5.4 – Convergence d'AG-Op avec les trois individus initiaux (bj)

Les figures 5.3, 5.4, 5.5 et 5.6 présentent les courbes de convergence d'AG-Op obtenues pour le programme *bj* pour les trois types d'individus initiaux et en utilisant différentes tailles de population et différentes probabilités de mutation. Notons qu'en utilisant les individus 1 et 2, la qualité obtenue est semblable. Par contre, AG-Op a une convergence un peu plus lente avec l'individu initial 1. En utilisant l'individu initial 3, AG-Op n'est pas capable d'améliorer la solution initiale, parce que l'allocation donnée par un algorithme de coloriage de graphe est déjà un optimum local et AG-Op n'est pas capable de sortir de cet optimum. Par conséquent, nous choisissons l'individu initial 2.

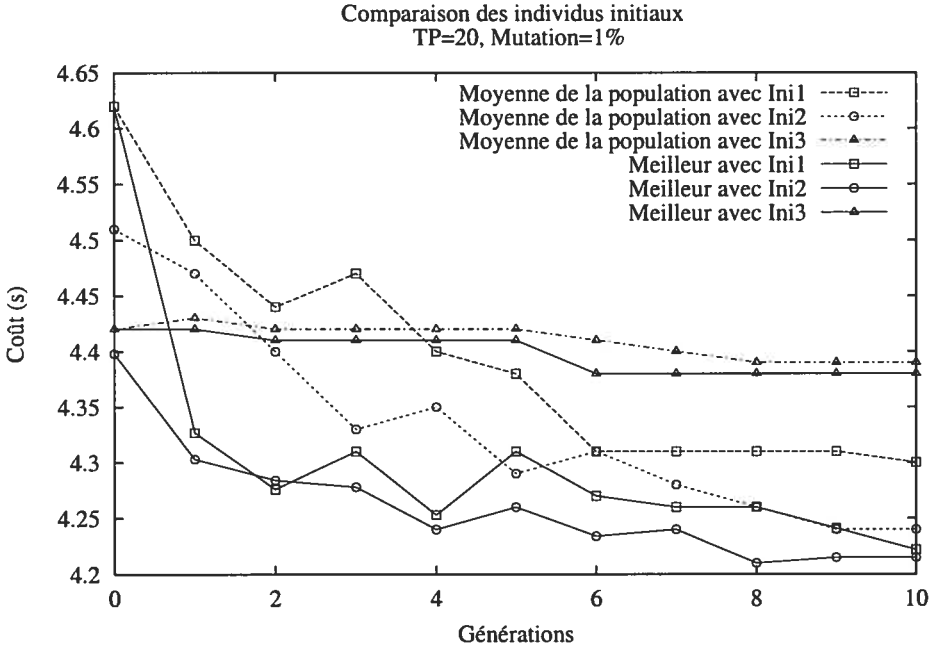


FIG. 5.5 – Convergence d'AG-Op avec les trois individus initiaux (bj)

À première vue, il semble étonnant qu'AG-Op commençant avec l'allocation donnée par un algorithme de coloriage de graphe n'arrive pas à améliorer la solution initiale. Dans les figures 5.3, 5.4, 5.5 et 5.6, on observe, aussi, qu'en utilisant l'individu initial 3, la moyenne de la population est toujours très proche du meilleur individu. Donc nous croyons que tous les individus sont très rapprochés de la solution initiale qui est déjà un optimum local. Pour vérifier ceci nous avons regardé les caractéristiques de la

population à différentes générations durant l'exécution d'AG-Op. Nous avons constaté que la population est très homogène, c'est-à-dire, les individus se ressemblent beaucoup les uns aux autres. Ce manque de variabilité génétique est sûrement dû au fait qu'AG-Op commence avec un optimum local et, donc, il converge rapidement vers cette solution. Une augmentation de la probabilité de mutation (figures 5.4 et 5.6) ne change pas ce comportement de façon importante.

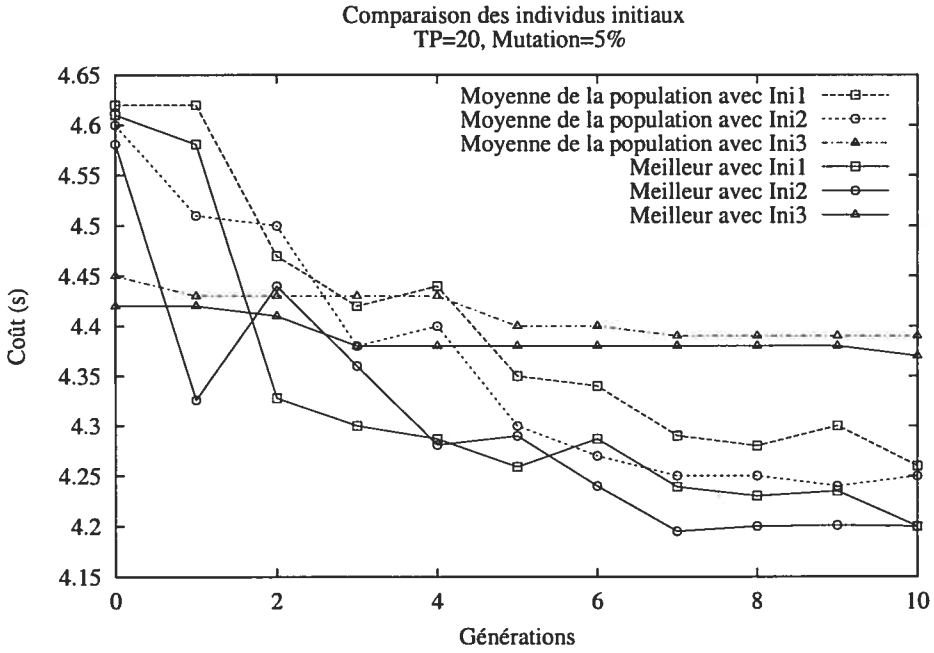


FIG. 5.6 – Convergence d'AG-Op avec les trois individus initiaux (bj)

5.1.3 Les opérateurs de croisement

Nous faisons l'évaluation numérique des opérateurs de croisement associés à chaque problème de façon indépendante. Premièrement, nous évaluons les opérateurs de croisement associés au problème d'allocation de registres. Par la suite, ayant choisi un opérateur de croisement pour le problème d'allocation de registres, nous évaluons les opérateurs associés à l'ordonnancement d'instructions. Dans les deux cas nous évaluons les opérateurs OBX et PBX décrits au chapitre 2. Les figures 5.7 et 5.8 illustrent la

convergence d'AG-Op pour le problème d'allocation de registres sur le programme *bj*. Nous utilisons deux différentes tailles de population pour mieux tester le comportement des opérateurs. Avec toutes les tailles de population, la qualité de la solution obtenue et le nombre de générations requises sont similaires mais nous pouvons constater que l'opérateur OBX est un peu plus performant que PBX.

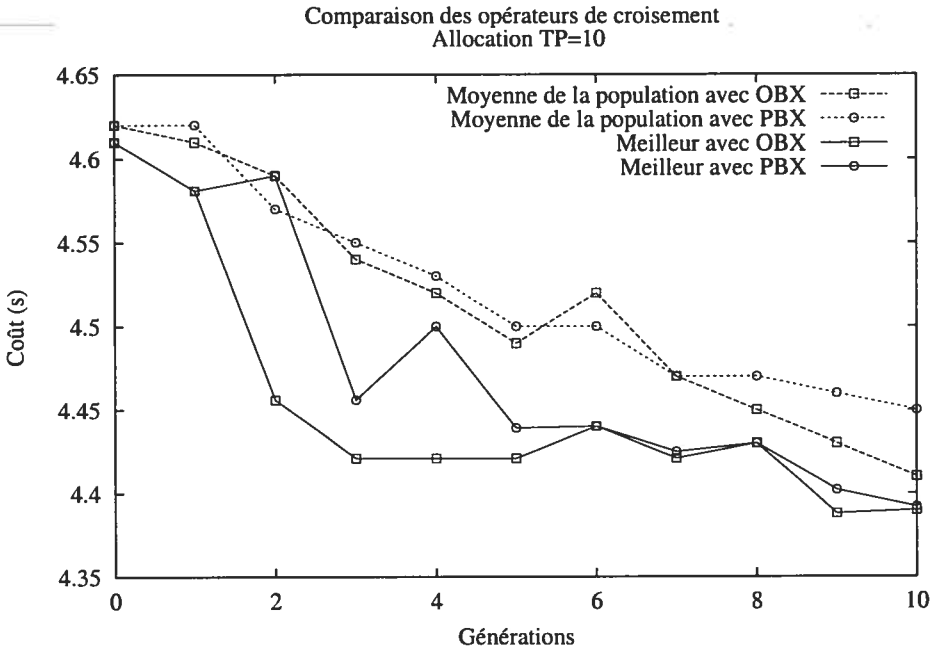


FIG. 5.7 – Convergence d'AG-Op avec différents opérateurs de croisement pour l'allocation de registres (bj)

Les figures 5.9 et 5.10 illustrent les résultats obtenus par AG-Op en utilisant l'opérateur OBX pour l'allocation de registres et les opérateurs OBX et PBX pour l'ordonnancement d'instructions avec différentes tailles de population. Les deux opérateurs obtiennent des performances semblables. Tout comme les performances obtenues dans le problème d'allocation, OBX a une convergence un peu plus rapide. Ainsi, nous choisissons cet opérateur.

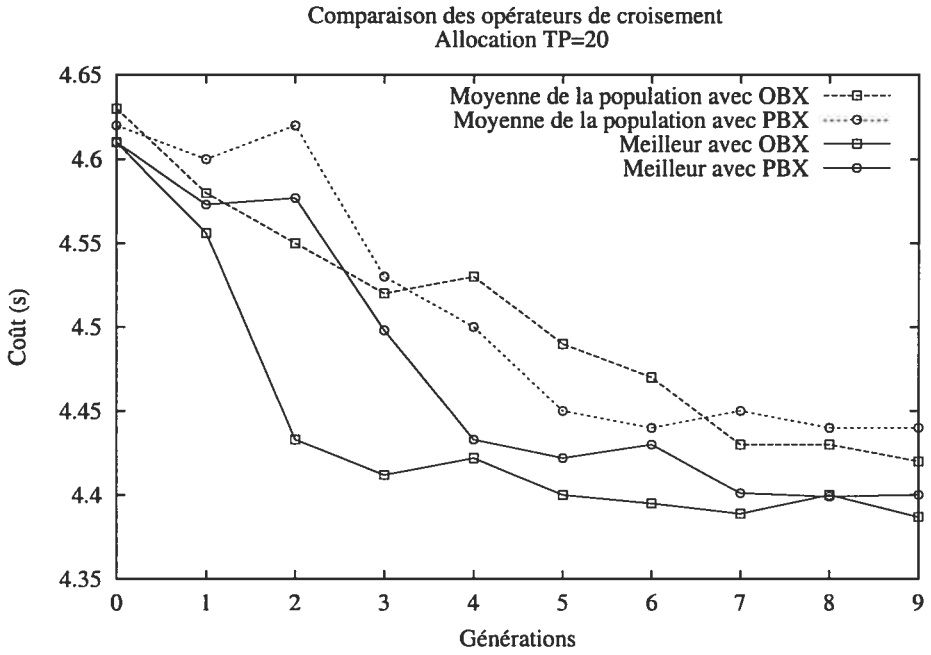


FIG. 5.8 – Convergence d'AG-Op avec différents opérateurs de croisement pour l'allocation de registres (bj)

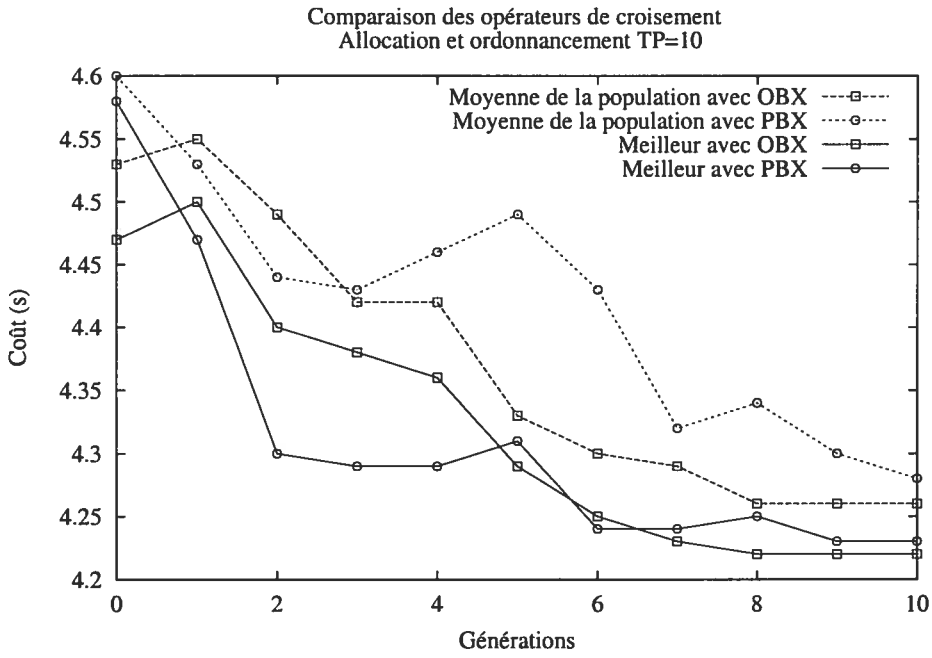


FIG. 5.9 – Convergence d'AG-Op avec différents opérateurs de croisement pour l'ordonnancement d'instructions (bj)

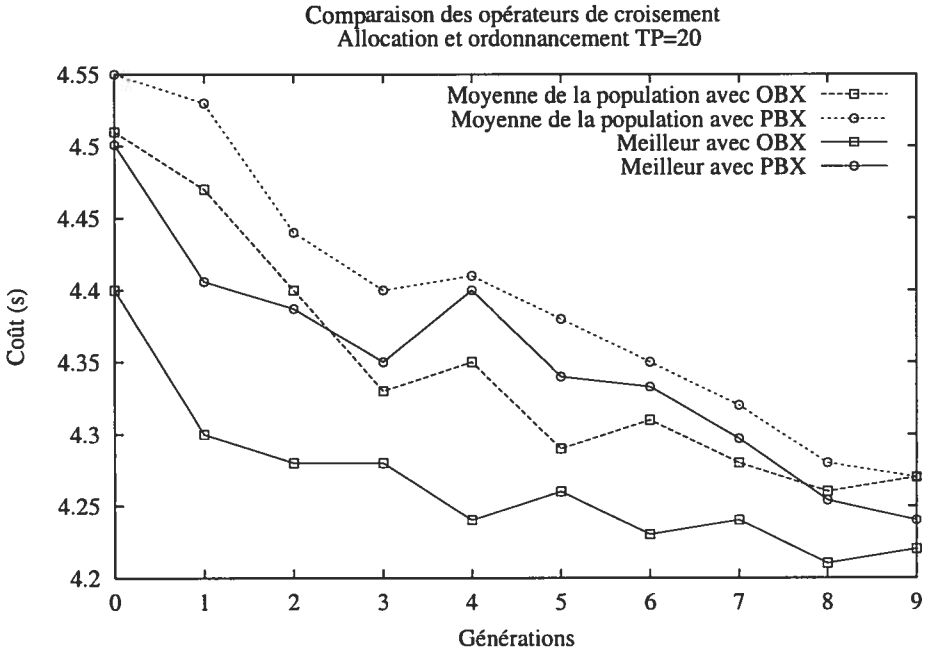


FIG. 5.10 – Convergence d'AG-Op avec différents opérateurs de croisement pour l'ordonnancement d'instructions (bj)

5.1.4 Les paramètres

Nous cherchons l'ensemble de paramètres menant à la meilleure performance, c'est-à-dire, ceux avec lesquels AG-Op génère du code de meilleure qualité dans un temps de compilation court (le plus petit nombre de générations). Afin de trouver ces paramètres, nous testons différentes tailles de population, probabilités de mutation et pourcentages de sélection.

Tel que nous avons fait au chapitre 3, nous avons réalisé quelques tests préliminaires afin de connaître l'ordre de grandeur des valeurs que nous devions tester. De ces tests nous avons obtenu les informations suivantes :

1. Les tailles de population proche de *0.5*MAXVAR*, *MAXVAR* et *2*MAXVAR*, (où *MAXVAR* est le nombre maximal de variables à allouer dans chaque procédure)

semble approprié. Nous avons été surpris de ces valeurs puisqu'elles sont petites et dépendent uniquement de la taille du problème d'allocation. Mais, en testant des tailles plus grandes la qualité de la solution ne s'améliore pas.

2. AG-Op n'est pas très sensible à la probabilité de mutation donc, nous testons les valeurs typiques.
3. L'utilisation d'un pourcentage de sélection différent de zéro semble améliorer la qualité de la solution.

La taille de la population. Nous observons à la figure 5.11 la convergence d'AG-Op en utilisant les tailles de population de *0.5*MAXVAR*, *MAXVAR* et *2*MAXVAR*. Cette fois-ci, l'ordonnée indique le nombre individus évalués. La qualité de la solution est similaire dans tous les cas. Donc, il n'est pas nécessaire d'utiliser une population de grande taille. Nous choisissons une taille de population de *0.5*MAXVAR*.

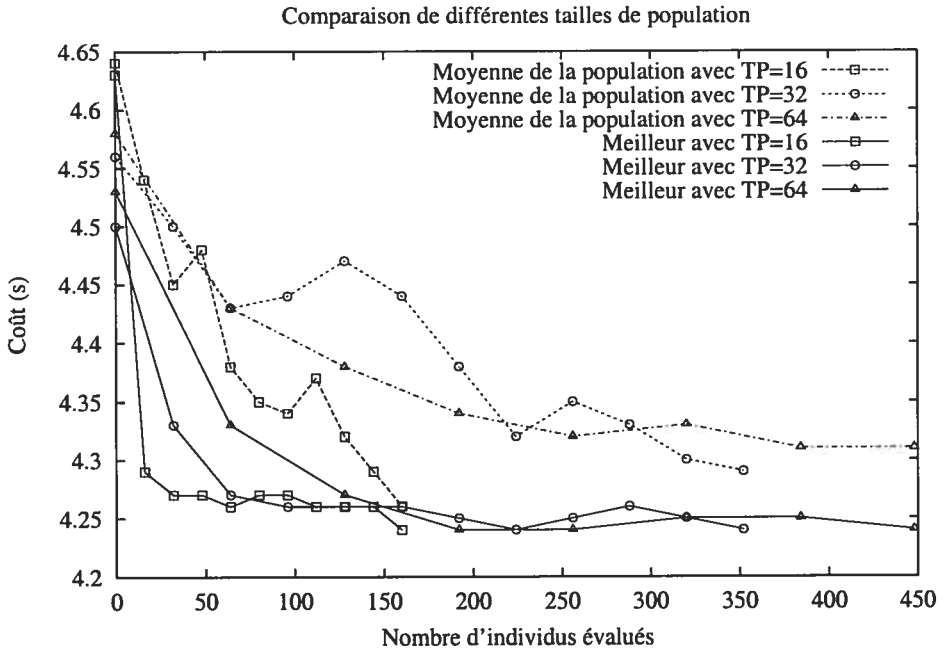


FIG. 5.11 – Convergence d'AG-Op en utilisant différentes tailles de population (bj)

La probabilité de mutation. Généralement la probabilité de mutation est très petite (environ 0.01). Cependant, dans le cas où il existerait trop d'optimums locaux,

une probabilité plus élevée peut être plus adéquate. Alors, nous testons les valeurs 0.01, 0.05, 0.1 et 0.2. La figure 5.12 montre la convergence d'AG-Op en utilisant ces quatre valeurs. Considérant la qualité de la solution et le nombre de générations, nous choisissons la probabilité de mutation 0.1.

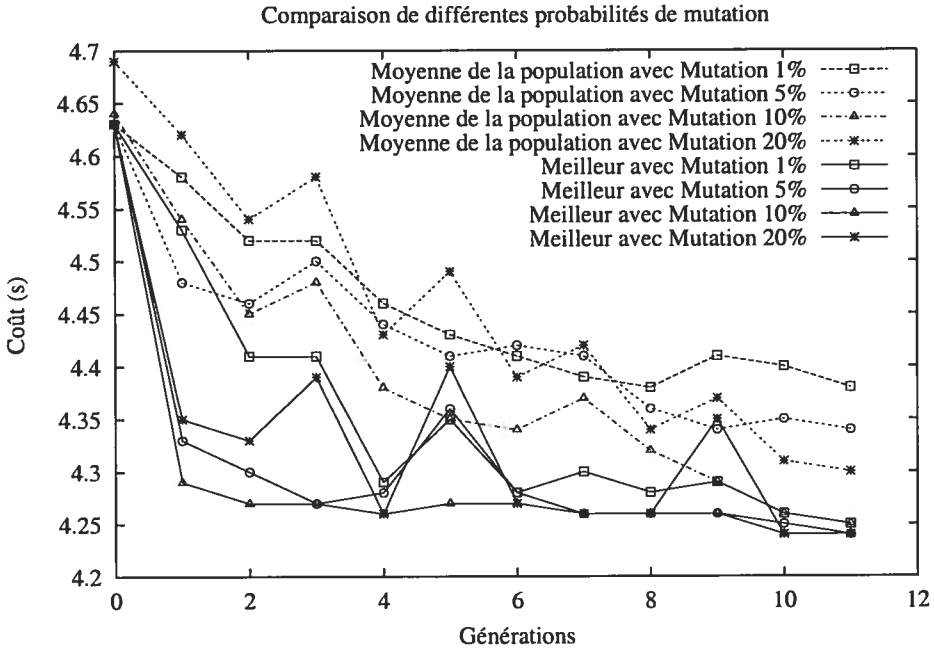


FIG. 5.12 – Convergence en utilisant différents pourcentages de mutation

Le pourcentage de sélection. Généralement, avec une petite taille de population, il n'est pas conseillé d'utiliser un pourcentage de sélection élevé car cela pourrait faire en sorte que la variabilité génétique ne soit pas suffisante. Par contre, dans plusieurs cas, il est profitable d'utiliser un pourcentage de sélection différent de zéro puisque ceci assure que les meilleurs individus survivent d'une génération à l'autre. La figure 5.13 présente les courbes de convergence en utilisant les pourcentages 0, 0.2 et 0.4. Avec le pourcentage le plus élevé, la convergence est trop lente (l'algorithme nécessite presque le double de générations pour arriver à la même solution). Cela s'explique par la similarité entre les générations. En utilisant les pourcentages 0 et 0.2, nous n'observons pas beaucoup de différence. Malgré tout, en utilisant le pourcentage de sélection de 0.2, la qualité de la solution est un peu supérieure. Alors, nous choisissons cette valeur comme pourcentage

de sélection.

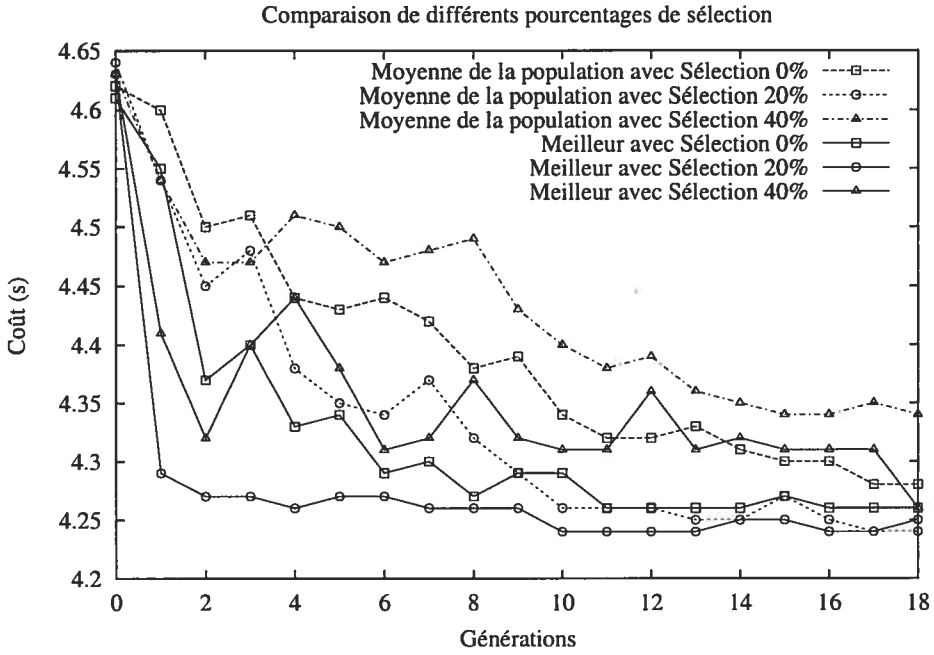


FIG. 5.13 – Convergence en utilisant différents pourcentages de sélection

Il faut noter que, même s'il y a des variations dans la performance d'AG-Op avec différents paramètres, ces différences ne sont pas très importantes. Alors, nous pouvons dire que notre AG-Op n'est pas très sensible aux paramètres utilisés. Cette caractéristique est très importante car nous faisons l'ajustement des paramètres à l'aide de quelques programmes de test seulement mais notre AG-Op doit s'exécuter sur d'autres programmes aussi.

5.2 Analyse de performance

La machine SPARC utilisée pour les expériences décrites ci-dessous est une Ultra-1, avec un processeur UltraSparc-I de 167 MHz et 768 Mo de mémoire vive. Nous utilisons la fonction du système d'exploitation *ptime* pour mesurer le temps d'exécution. Nous ajoutons le temps usager et le temps système. Tous les résultats présentés sont des

moyennes sur cinq exécutions. Dans tous les cas, la déviation standard est plus petite que 0.005 secondes.

Pour choisir les programmes de test, nous avons cherché parmi les programmes de test librement disponibles. Premièrement, nous nous intéressons aux programmes de test qui utilisent un grand nombre de variables dans la même procédure (le nombre de variables est plus grand que le nombre de registres disponibles). Ainsi, l'allocation de registres est plus intéressante. Deuxièmement, nous avons ajouté des programmes typiques de différentes tailles afin d'évaluer l'impact de notre approche sur un ensemble plus hétérogène de programmes. Les programmes de test utilisés sont décrits à la table 5.1. La table 5.2 indique le nombre de procédures, le nombre de variables utilisées dans chaque procédure, le nombre de blocs de base et la taille du plus grand bloc de base pour chaque programme de test.

Nous parlons souvent dans cette section de *pourcentage d'amélioration* d'un compilateur ou d'une approche A par rapport à un autre compilateur ou approche B. Cette valeur est dénotée par A/B et elle est calculée comme suit :

$$A/B = ((R_B - R_A) * 100)/R_B$$

ou :

R_A : le résultat obtenu pour l'approche A.

R_B : le résultat obtenu pour l'approche B.

Pour commencer notre analyse, nous comparons (table 5.3) les temps d'exécution de chaque programme de test en le compilant avec GNU C (version 2.95.2 avec l'option d'optimisation -O) et *lcc*. Nous pouvons constater que GNU C est beaucoup plus performant que *lcc*. La troisième colonne de la table 5.3 montre le pourcentage d'amélioration de GNU C par rapport à *lcc*. Ceci est principalement dû au fait que GNU C effectue des optimisations que *lcc* ne réalise pas (optimisations de boucle, optimisation de branchement, élimination des expressions communes, etc), ce qui rend la comparaison difficile. Par conséquent, notre AG-Op, qui utilise *lcc*, ne peut pas être comparé à GNU C. Alors

Nom	Description
bj	Programme qui joue au black-jack afin de mesurer la performance de la machine
bubble	GNU *bubble sort*
dhr	Dhrystone
fft	Implantation de FFT
fib	Calcul de nombres de fibonacci
flow	Implantation de l'algorithme exhaustif pour calculer le flux maximal d'un réseau
fulk	Implantation de l'algorithme de Ford-Fulkerson pour calculer le flux maximal d'un réseau
hanoi	Programme résolvant le problème des tours de Hanoi
heap	GNU *heapsort*
heapsort	Programme qui exécute plusieurs fois un heapsort afin de mesurer la performance de la machine
insertion	GNU *insertion sort*
merge	GNU *merge sort*
arithm	Programme qui effectue différentes opérations arithmétiques à l'intérieur de boucles
queens	Programme qui résout le problème des n reines
quick	GNU *quick sort*
selection	GNU *selection sort*
shuffle	Vérification d'un générateur de nombres aléatoires
tfftdp	Autre implantation de FFT
whet	Whetstone simple précision

TAB. 5.1 – Description des programmes de test utilisés

nous ne l'utilisons plus comme base de comparaison dans nos prochaines analyses.

Notons la grande différence entre les résultats pour le problème *arithm*, ceci s'explique par le fait que *arithm* fait beaucoup de calculs, mais il ne fait rien avec le résultat. GNU C s'aperçoit de ce comportement et il élimine presque toutes les opérations arithmétiques réalisées dans les corps des boucles.

La table 5.4 présente les temps de compilation requis par *lcc* et par AG-Op pour tous les programmes de test. Clairement, AG-Op prend beaucoup plus de temps à compiler que *lcc*. Comme nous l'avons expliqué dans le chapitre précédent, le fait d'utiliser *lcc* comme base pour notre AG-Op entraîne des coûts additionnels à ceux qui sont propres

Nom	Nombre de procédures	Nombre de variables utilisées par procédure	Nombre total de blocs de base	Taille du bloc de base le plus grand
arithm	1	19	12	139
bj	5	32-6-2-2-6	562	119
bubble	2	3-1	16	8
dhr	13	7-2-2-2-2-1-1-1-0-0-0-0-0	138	23
fft	5	3-14-3-3-3	119	154
fib	3	2-0-0	18	18
flow	10	4-4-3-3-3-3-2-1-0-0	150	25
fulk	10	4-4-4-4-3-3-3-2-1-0	168	35
hanoi	3	2-1-0	26	23
heap	3	3-2-1	28	10
heapsort	2	21-5	70	94
insertion	2	3-1	16	5
merge	4	4-1-1-0	31	9
queens	1	18	93	171
quick	3	3-1-0	26	6
selection	2	4-1	17	8
shuffle	3	8-6-6	68	112
tfftdp	2	16-10	83	130
whet	7	14-9-1-0-0-0-0	248	51

TAB. 5.2 – Caractéristiques des programmes de test utilisés

à notre approche. Étant donné que la fonction de coût est le temps réel d'exécution, notre AG-Op ne peut pas compiler dans un temps plus court que :

$$G * TP * Te$$

où :

G est le nombre de générations de l'AG-OP ;

TP est la taille de la population ;

Te le temps d'exécution de chaque programme généré.

La quatrième colonne de la table 5.4 donne le calcul de ce temps minimal en utilisant comme temps d'exécution le temps de chaque programme compilé avec *lcc*. La cinquième colonne de la table 5.4, indique le temps d'exécution d'AG-Op (génération de

Programme de test	GNU C (s)	lcc (s)	GNU C/lcc (%)
arithm	0.1	13.4	99.3
bj	3.4	4.4	22.7
bubble	4.1	6.3	34.9
dhr	1.4	2.5	44.0
fft	2.9	3.9	25.6
fib	2.0	2.1	4.8
flow	7.3	13.6	46.3
fulk	3.6	5.6	35.7
hanoi	24.9	29.9	16.7
heap	2.6	3.9	33.3
heapsort	13.1	14.4	9.0
insertion	2.7	3.6	25.0
merge	2.5	3.1	19.4
queens	0.3	0.4	25.0
quick	1.7	2.0	15.0
selection	2.3	2.8	17.9
shuffle	29.3	38.6	24.1
tfftdp	6.3	7.5	16.0
whet	19.6	20.9	6.2

TAB. 5.3 – Temps d'exécution des programmes de test compilés avec GNU C et *lcc*.

la population, application des opérateurs génétiques, applications des fonction de réparation, etc). Le temps additionnel est le temps requis pour générer les codes assembleurs (que nous utilisons comme représentation intermédiaire) et pour construire les graphes d'interférence et de dépendances. Ce temps pourrait être diminué de façon importante en utilisant un compilateur qui génère une représentation intermédiaire contenant les informations requises par notre AG-Op.

L'utilisation d'AG-Op est viable seulement pour les usagers qui peuvent se permettre de troquer un temps de compilation élevé pour un gain sur le temps d'exécution du code. Maintenant, nous portons notre attention à l'analyse de la performance de notre AG-Op quant à la qualité du code généré.

La performance de notre système est comparée à celles des approches existantes

Programme de test	lcc (s)	AG-Op (s)	Fonction de coût (s)	Opérateurs génétiques (s)
arithm	0.03	1241.1	1201.5	21.0
bj	0.02	3064.2	708.3	148.2
bubble	0.03	1260.0	626.2	27.7
dhr	0.01	784.6	250.7	32.6
fft	0.03	617.6	271.5	43.8
fib	0.03	539.2	212.4	29.6
flow	0.03	3590.5	1363.4	30.5
fulk	0.03	1148.6	557.9	27.7
hanoi	0.03	6573.0	2992.9	31.5
heap	0.03	959.1	393.2	29.9
heapsort	0.03	5072.3	1587.4	101.6
insertion	0.03	896.3	364.2	28.9
merge	0.03	869.7	312.9	30.3
queens	0.03	608.3	34.5	76.4
quick	0.02	686.0	202.6	26.5
selection	0.02	695.3	284.1	29.1
shuffle	0.02	4079.3	1544.2	28.1
tfftdp	0.03	894.6	600.2	28.9
whet	0.02	4871.4	1464.0	32.7

TAB. 5.4 – Temps de compilation obtenus par *lcc* et AG-Op.

pour les problèmes d'allocation de registres et d'ordonnancement d'instructions. Pour l'allocation de registres, nous utilisons l'allocateur de *lcc* et un allocateur par coloriage de graphe tel que décrit par Appel [App97]. L'heuristique utilisée par cet allocateur afin de décider quel nœud doit se faire déborder vers la mémoire est basée sur le nombre de fois que la variable est utilisée et sur le nombre d'arêtes du nœud. Ainsi, l'allocateur privilégie pour faire le débordement des variables qui interfèrent avec beaucoup d'autres et qui sont peu utilisées.

Pour l'ordonnancement d'instructions nous avons implanté un ordonnancement par liste [Muc97]. Nous avons testé trois heuristiques afin de choisir le nœud qui sera assigné :

1. le nœud qui est le premier dans l'ordre original du bloc de base (ordre original) ;
2. le nœud qui a la latence la plus grande ;

3. le nœud qui maximise le parallélisme, c'est-à-dire le nœud qui maximise le nombre de nœuds qui seront ensuite prêts à être exécutés.

Comme le montre la figure 5.5, la troisième heuristique obtient les meilleurs résultats dans la plupart de cas. Nous choisissons cette heuristique pour le reste de notre analyse.

Programme de test	Heuristique 1 (s)	Heuristique 2 (s)	Heuristique 3 (s)
arithm	13.3	13.0	12.7
bj	4.4	4.3	4.3
bubble	6.3	6.0	5.9
dhr	2.5	2.6	2.6
fft	3.8	3.8	3.8
fib	2.1	2.1	2.1
flow	13.6	13.4	13.2
fulk	5.6	5.7	5.8
hanoi	30.0	30.0	30.2
heap	3.9	4.0	3.4
heapsort	13.6	13.1	12.5
insertion	3.6	3.6	3.6
merge	3.2	3.0	3.0
queens	0.4	0.4	0.4
quick	2.0	2.0	2.0
selection	2.8	2.7	2.3
shuffle	38.6	38.8	38.4
tfftdp	7.8	7.6	7.6
whet	20.9	20.6	20.4

TAB. 5.5 – Performance des différentes heuristiques pour l'ordonnancement d'instruction par liste

Nous avons modifié *lcc* de façon à pouvoir contrôler le nombre de registres disponibles. Nous compilons les programmes de test en supposant la présence de 4, 8 et 12 registres. Ceci nous permet d'analyser la sensibilité des différentes approches au nombre de registres disponibles. Pour compiler avec 12 registres, nous devons diminuer le nombre de registres utilisés pour les variables temporaires (comme nous avons expliqué dans le chapitre 4, nous utilisons des ensembles disjoints pour les variables de l'usager et les temporaires). Afin que nos comparaisons soient valides, nous utilisons le même nombre de registres pour les temporaires dans tous les cas.

Pour la suite de ce chapitre, nous désignons chaque programme de test par le nom du test donné à la table 5.1 suivi du nombre de registres utilisés. Par exemple, le programme de test *bj* compilé avec 8 registres est désigné par *bj8*.

Les courbes des figures 5.14 et 5.15 illustrent la convergence d'AG-Op pour les programmes de test *bj8* et *tfftdp8*. Nous présentons la meilleur solution obtenue par AG-Op et nous fixons comme point de référence les courbes constantes représentant les temps d'exécution en utilisant :

- l'allocateur de *lcc* et un ordonnancement par liste (lcc+sch) ;
- un allocateur par coloriage de graphe et un ordonnancement par liste (color+sch).
- le temps d'exécution en pire cas, soit avec toutes les variables en mémoire et sans ordonnancement d'instructions.

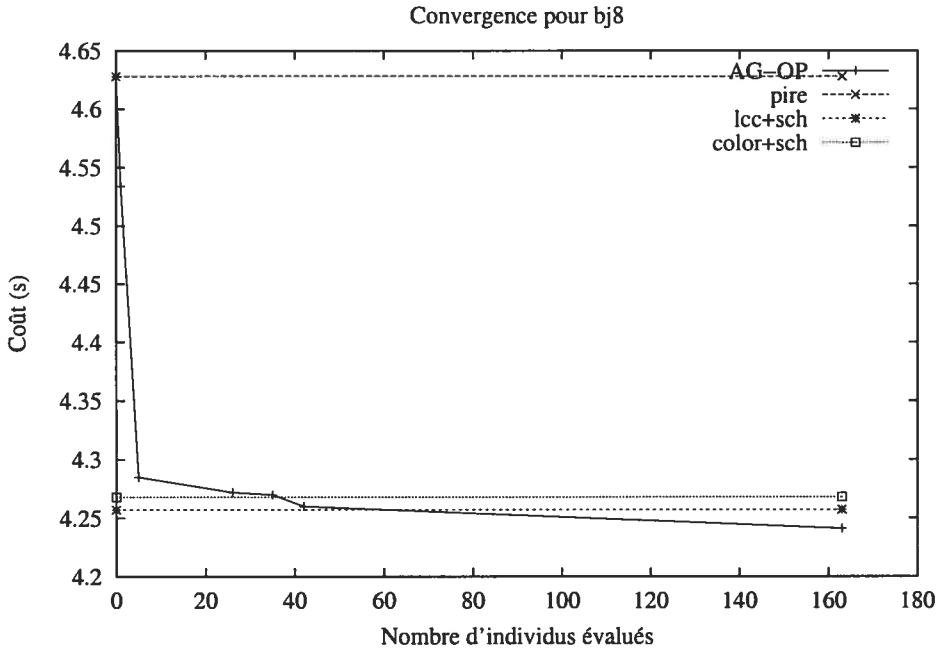


FIG. 5.14 – Convergence d'AG-Op sur *bj8*

D'après les graphiques des figures 5.14 et 5.15, AG-Op converge rapidement à des valeurs proches de celles obtenues par les approches traditionnelles. Ce comportement est semblable pour tous les programmes de test.

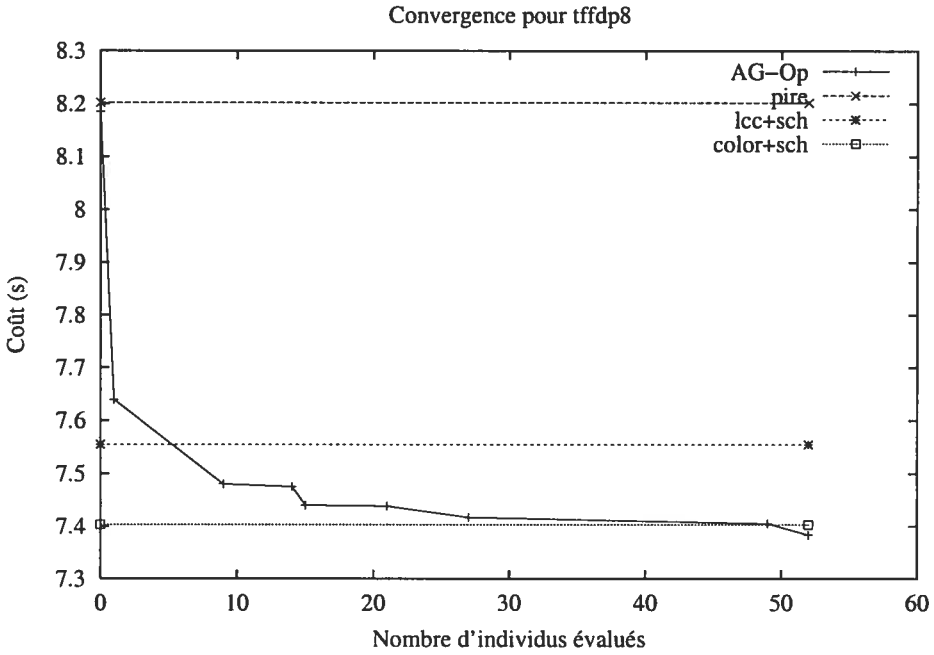


FIG. 5.15 – Convergence d'AG-Op sur *fftdp8*

Dans les programmes de test où le nombre de variables utilisées est plus petit ou égal au nombre de registres disponibles (voir la table 5.2), toutes les approches d'allocation de registres (*lcc*, coloriage et AG-Op) obtiennent une allocation équivalente (toutes les variables sont allouées à un registre mais pas nécessairement au même registre) ou même identique. Il faut noter que, pour ces programmes de test, il n'est pas raisonnable d'utiliser comme taille de population 0.5 fois le nombre de variables (tel que nous l'avons établi dans l'ajustement de paramètres). Cette taille de population peut être suffisante pour le problème d'allocation de registres mais risque d'être nettement trop petite pour le problème d'ordonnancement d'instructions. Alors, pour tous ces programmes de test, nous utilisons une taille de population de 10. Cette valeur provient d'une suite de tests empiriques.

La table 5.6 présente les temps d'exécution en secondes pour cet ensemble de programmes de test compilés avec :

1. lcc+sch : allocation de registres obtenue par *lcc* et ordonnancement d'instructions

Programme de test	lcc+sch (s)	color+sch (s)	AG-Op (s)	AG/lcc+sch (%)	AG/color+sch (%)
bubble4	5.7	5.6	5.0	12.3	10.7
bubble8	5.7	5.6	5.0	12.3	10.7
bubble12	5.7	5.7	5.0	12.3	12.3
dhr8	2.6	2.6	2.6	0.0	0.0
dhr12	2.6	2.6	2.6	0.0	0.0
fib4	2.1	2.1	1.7	19.0	19.0
fib8	2.1	2.0	1.7	19.0	15.0
fib12	2.0	2.1	1.7	15.0	19.0
flow4	13.2	13.3	12.9	2.3	3.0
flow8	13.2	13.2	12.9	2.3	2.3
flow12	13.2	13.3	13.0	1.5	2.3
fulk4	5.8	5.8	5.5	5.2	5.2
fulk8	5.8	5.8	5.5	5.2	5.2
fulk12	5.8	5.8	5.5	5.2	5.2
hanoi4	30.2	30.2	29.8	1.3	1.3
hanoi8	30.2	30.2	29.8	1.3	1.3
hanoi12	30.2	30.2	29.9	1.0	1.0
heap4	3.6	3.6	3.5	2.8	2.8
heap8	3.6	3.6	3.5	2.8	2.8
heap12	3.6	3.6	3.5	2.8	2.8
insertion4	3.6	3.6	3.6	0.0	0.0
insertion8	3.6	3.6	3.6	0.0	0.0
insertion12	3.6	3.6	3.6	0.0	0.0
merge4	3.0	3.0	3.0	0.0	0.0
merge8	3.0	3.0	3.0	0.0	0.0
merge12	3.0	3.0	3.0	0.0	0.0
quick4	2.0	2.0	2.0	0.0	0.0
quick8	2.0	2.0	2.0	0.0	0.0
quick12	2.0	2.0	2.0	0.0	0.0
selection4	2.4	2.4	2.4	0.0	0.0
selection8	2.4	2.4	2.4	0.0	0.0
selection12	2.4	2.4	2.4	0.0	0.0
shuffle8	37.2	37.1	37.3	-0.3	-0.5
shuffle12	37.2	37.2	37.2	0.0	0.0

TAB. 5.6 – Temps d'exécution obtenus pour les programmes de test qui utilisent peu de variables

par liste ;

2. color+sch : allocation de registres par coloriage de graphe et ordonnancement d'instructions par liste ;
3. AG-Op.

La table 5.6 inclut aussi les pourcentages d'améliorations obtenus par AG-Op par rapport à lcc+sch et color+sch.

Notons que les temps de la deuxième colonne de la table 5.6 ne correspondent pas exactement aux temps présentés dans la table 5.3. La table 5.3 présente les temps d'exécution des programmes de test en les compilant avec *lcc* sans aucune modification. Par contre, la version de *lcc* utilisée pour mesurer les temps montrés dans la deuxième colonne de la table 5.6 a deux ensembles de registres différents, un pour allouer les variables d'usager et un autre pour allouer les variables temporaires. L'ensemble utilisé pour les temporaires est plus petit que celui utilisé par le *lcc* original. Ceci peut provoquer des débordements additionnels vers la mémoire. De plus, les valeurs indiquées dans la table 5.6 correspondent à l'approche lcc+sch ; c'est-à-dire que l'ordonnancement d'instructions par liste a été appliqué. L'ordonnancement par liste est basé sur une heuristique, donc il n'est pas sûr que son application améliorera la qualité de la solution, d'ailleurs, dans quelque cas elle pourrait détériorer la qualité de la solution initiale.

Notons que les performances de lcc+sch et color+sch sont presque identiques. Nous expliquons cette similarité par le fait que les allocations de registres générées sont les mêmes (ou presque) et, donc, que l'algorithme d'ordonnancement d'instructions obtient les mêmes résultats dans les deux cas. Bien que AG-Op obtient aussi des allocations de registres équivalentes, on note, cependant, une amélioration par rapport à lcc+sch et color+sch. Cette amélioration varie passablement d'un programme de test à l'autre.

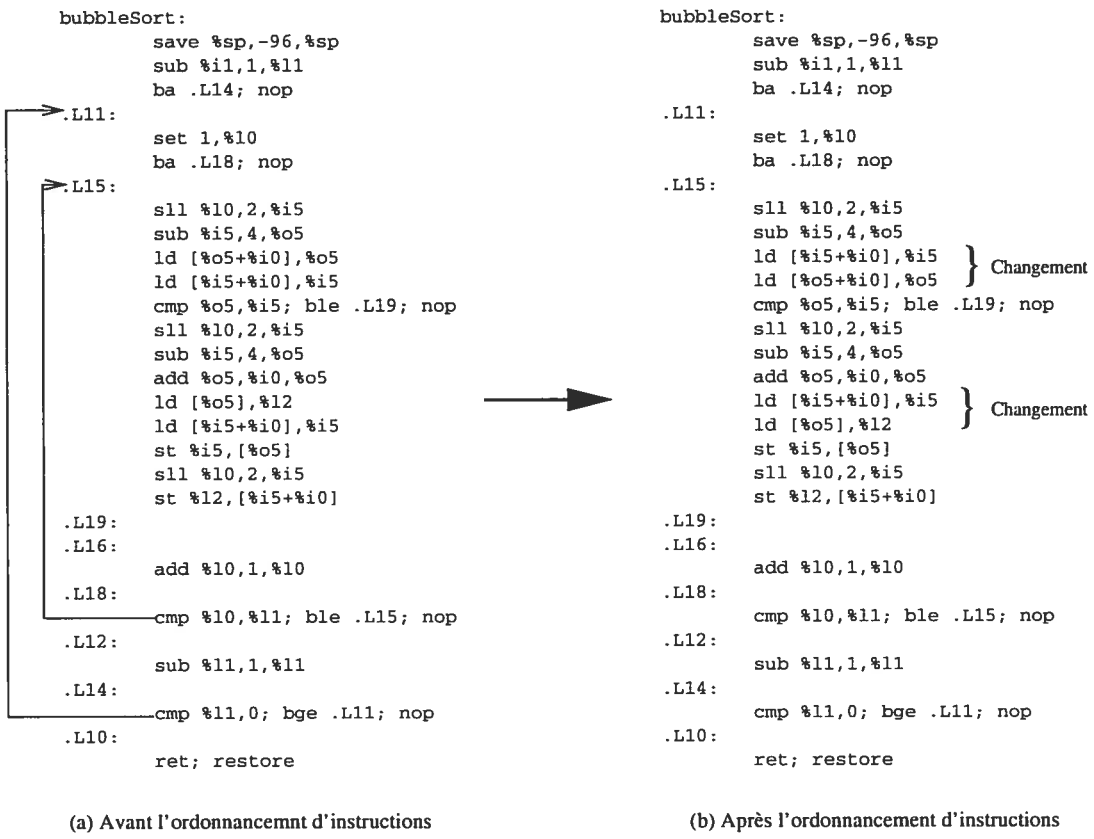
Voici comment nous expliquons ce phénomène. Si nous consultons à nouveau la table 5.2, nous pouvons constater que les blocs de base les plus grands dans les programmes de test *fib*, *flow*, *fulk* et *hanoi* (18, 25, 35 et 23, respectivement) sont plus grands que ceux dans les autres programmes de test présentés dans la table 5.6. Ceci

explique que AG-Op génère une amélioration plus intéressante dans ces cas que dans d'autres cas comme *insertion*, *merge*, *quick* et *selection*, qui ont des tailles de blocs de base de 5, 9, 6 et 8, respectivement. Lorsqu'un bloc de base est plus grand, la possibilité qu'un bon ordonnancement ait un impact sur le temps d'exécution augmente.

Cependant, ce comportement n'est pas vérifié pour *shuffle* puisque la taille du plus grand bloc de base est de 112, et alors AG-Op n'arrive pas à faire mieux que les autres approches. En analysant la structure de *shuffle* avec plus d'attention, nous constatons qu'en général ses blocs de base sont de petite taille. Il y a seulement un bloc de base de taille importante (112) et les instructions dans ce bloc de base sont très dépendantes. Ainsi, il y a très peu de flexibilité pour l'ordonnancement d'instructions.

Néanmoins, une très bonne performance d'AG-Op est observée pour *bubble*, où l'amélioration est 12%. Le bloc de base le plus grand de ce programme de test contient 8 instructions, tout comme *selection* pour lequel AG-Op n'obtient pas de bons résultats. La même situation se présente avec *heap* (blocs de base de 10 instructions) où AG-Op obtient une meilleure performance que pour *merge* (bloc de base de 9 instructions). Pour expliquer cette situation, il faut situer l'endroit du code où se trouve le bloc de base le plus grand.

Les figures 5.16 et 5.17 présentent les codes assembleur pour les fonctions où se trouve le bloc de base le plus grand pour *bubble* et *selection* respectivement avant et après l'ordonnancement d'instructions. Nous observons que pour *bubble* le bloc de base plus grand est dans le corps d'une boucle double, tandis que dans *selection* le bloc de base le plus grand se trouve dans une boucle simple. Ceci explique que les optimisations effectuées sur l'ordre des instructions ont un impact plus grand dans *bubble* que dans *selection*. Il faut noter que, même si l'ordonnancement d'instructions n'induit pas de grands changements dans le code (seulement deux instructions sont changées), il est quand même responsable de la différence de 12% dans l'amélioration de la vitesse du code généré dans *bubble*. Nous trouvons ce même comportement avec *heap* et *merge*.



(a) Avant l'ordonnancemnt d'instructions

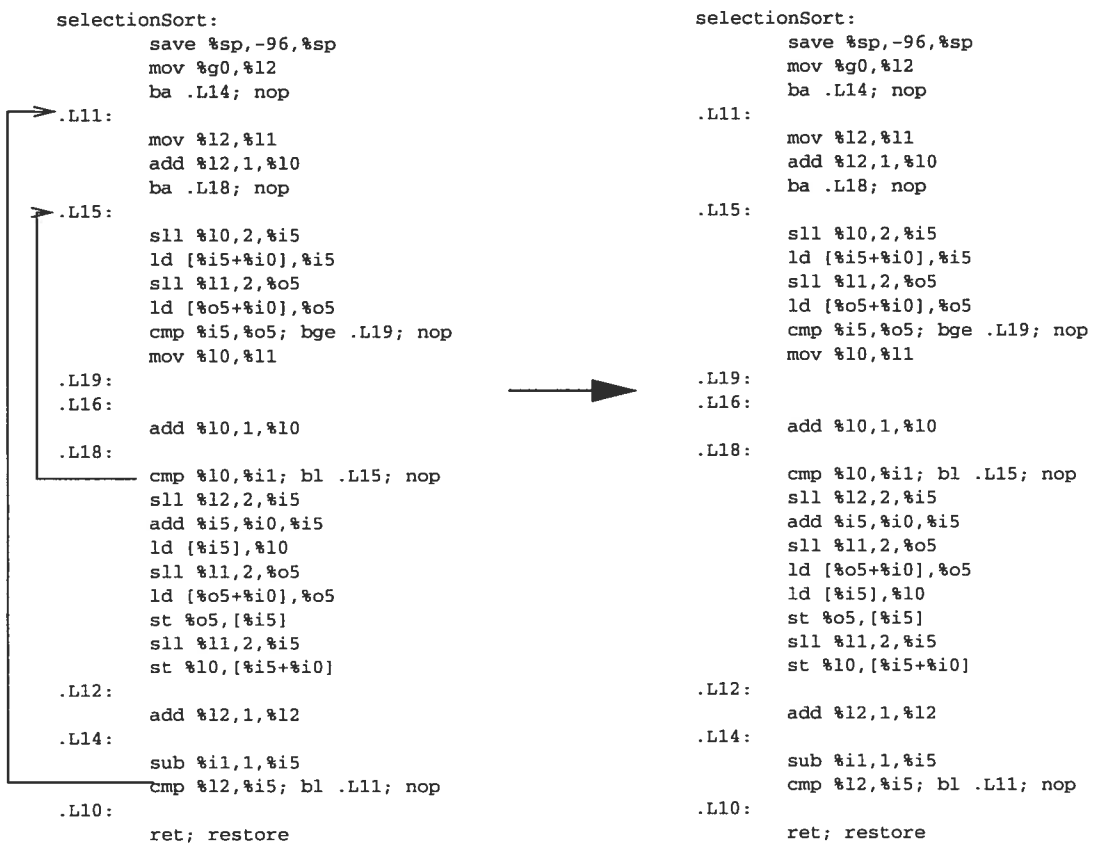
(b) Après l'ordonnancement d'instructions

FIG. 5.16 – Code assembleur pour *bubble*

Dans quelques cas, AG-Op entraîne une dégradation de performance mais les pertes sont très petites (moins que 1%) par rapport aux gains qu'il permet d'obtenir (jusqu'à 19%).

Pour les autres programmes de test, c'est-à-dire ceux où le nombre de variables est plus grand que le nombre de registres, la table 5.7 compare la performance des algorithmes d'allocation de registres de *lcc* et de coloriage de graphe. Nous voyons que dans presque la moitié des cas, *lcc* est meilleur que l'allocateur par coloriage de graphe. Puisqu'il nous est impossible de déterminer lequel est le meilleur en général, nous comparons les deux approches avec notre AG-Op.

La table 5.8 présente les temps d'exécution pour les programmes de test compilés avec toutes les approches, nous présentons aussi les résultats obtenus en utilisant la



(a) Avant l'ordonnancemnt d'instructions

(b) Après l'ordonnancement d'instructions

FIG. 5.17 – Code assembleur pour *selection*

politique du pire cas, c'est-à-dire, toutes les variables sont débordées vers la mémoire et sans ordonnancement d'instructions.

Pour nous aider à analyser les résultats, la figure 5.18 indique le pourcentage d'amélioration avec chaque approche par rapport au pire cas pour tous les programmes de test. Nous observons que les gains sont très importants dans tous les cas, atteignant jusqu'à 50% pour *heapsort* avec 8 registres. Par contre, les différences entre les méthodes d'optimisation ne sont pas très importantes, sauf dans le cas de *arithm*.

Nous pouvons constater que AG-Op obtient une meilleure performance pour *arithm* (le code source d'*arithm* est à l'annexe E) que pour les autres programmes de test. Mieux encore, les approches classiques (lcc+sch et color+sch) ont des performances qui se

Programme de test	lcc (s)	color (s)	color/lcc (%)
arithm4	13.4	13.4	0.0
arithm8	13.3	13.3	0.0
arithm12	13.4	13.4	0.0
bj4	4.5	4.4	2.2
bj8	4.4	4.4	0.0
bj12	4.4	4.4	0.0
dhr4	2.6	2.6	0.0
fft4	3.9	4	-2.6
fft8	3.8	3.8	0.0
fft12	4	3.8	5.0
heapsort4	16	15.5	3.1
heapsort8	13.6	13.7	-0.7
heapsort12	14	13.5	3.6
queens4	0.4	0.4	0.0
queens8	0.4	0.4	0.0
queens12	0.4	0.4	0.0
shuffe4	39.8	38.6	3.0
tfftdp4	7.7	7.7	0.0
tfftdp8	7.6	7.4	2.6
tfftdp12	7.4	7.3	1.4
whet4	21	21.9	-4.3
whet8	21	21.9	-4.3
whet12	19.5	19.6	-0.5

TAB. 5.7 – Résultats obtenus avec l'allocateur de *lcc* et avec l'allocateur par coloriage

rapprochent du pire cas. Ce comportement est dû au fait que ce programme de test a une structure qui trompe les heuristiques classiques. Dans *arithm*, il y a des variables qui sont beaucoup référencées et d'autres, peu. Les heuristiques traditionnelles pour l'allocation de registres donnent la priorité aux variables en fonction du nombre de fois où elles sont référencées. Ces approches fonctionnent normalement bien. Toutefois, à l'exécution, les variables peu référencées de *arithm* sont plus souvent accédées que les autres. Ainsi, les heuristiques traditionnelles donnent priorité aux mauvaises variables. En utilisant le temps réel d'exécution comme fonction de coût AG-Op, converge facilement vers la solution la plus appropriée.

Programme de test	pire (s)	lcc+sch (s)	color+sch (s)	AG-Op (s)
arithm4	13.4	12.7	12.7	10.6
arithm8	13.4	12.7	12.7	9.3
arithm12	13.4	12.8	12.8	10.6
bj4	4.6	4.4	4.3	4.2
bj8	4.6	4.3	4.3	4.2
bj12	4.6	4.3	4.3	4.2
dhr4	3.3	2.6	2.6	2.6
fft4	4.4	3.8	3.9	3.8
fft8	4.4	3.8	3.7	3.8
fft12	4.4	3.8	3.8	3.7
heapsort4	25.9	15.3	15.0	14.5
heapsort8	25.9	12.5	13.2	12.3
heapsort12	25.9	13.1	12.9	12.2
queens4	0.6	0.4	0.4	0.4
queens8	0.6	0.4	0.4	0.4
queens12	0.6	0.4	0.4	0.4
shuffe4	40.6	38.6	37.1	37.5
tfftdp4	8.2	7.7	7.7	7.6
tfftdp8	8.2	7.6	7.4	7.4
tfftdp12	8.2	7.4	7.3	7.2
whet4	22.8	20.4	21.5	20.5
whet8	22.8	20.4	21.4	20.4
whet12	22.6	19.1	19.1	19.1

TAB. 5.8 – Temps d'exécution en compilant avec toutes les approches

De plus, si nous comparons les performances obtenues par les trois approches, nous constatons qu'il n'y a pas une d'entre elles qui soit toujours la meilleure. Pour *fft4*, *queens4* et *whet4*, lcc+sch est le plus performant ; pour *dhr4*, *fft8* et *shuffle4*, color+sch est meilleur et pour les autres, AG-Op est le plus performant. Cependant, lorsque AG-Op n'est pas le plus performant il est quand même très proche du meilleur.

Nous voyons aussi qu'il n'existe pas de relation entre les différences de performance obtenues et le nombre de registres disponibles pour l'allocation de registres. Ceci n'est pas vraiment étonnant car la complexité du problème d'allocation de registres ne dépend

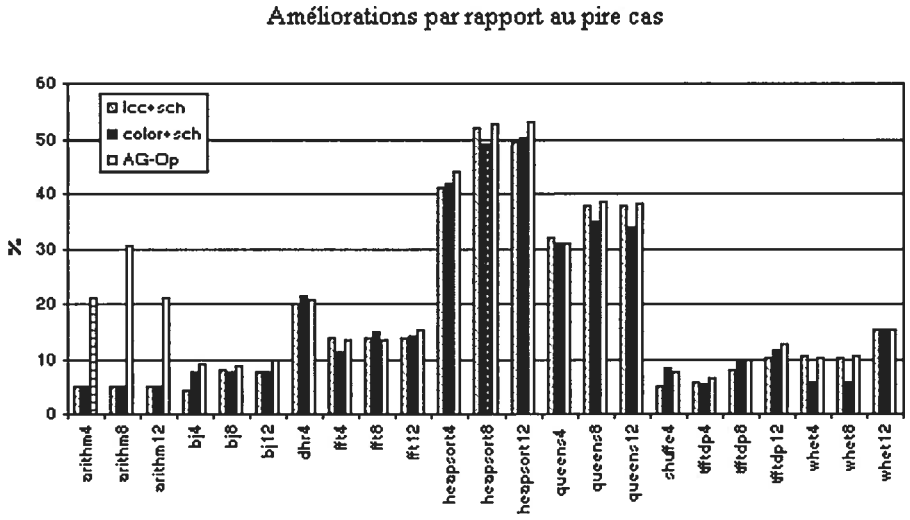


FIG. 5.18 – Pourcentage d'amélioration par rapport au pire cas

pas autant du nombre de registres que de la relation entre le nombre de registres et le nombre de variables. Si nous considérons le nombre total de combinaisons possibles pour le problème d'allocation de registres (soit m^n où m est le nombre de registres et n est le nombre de variables) nous avons une meilleure idée de la complexité pour chaque programme de test. Ainsi, si nous analysons les programmes de test par rapport à cette mesure de complexité, nous observons une corrélation intéressante. La table 5.9 présente les améliorations d'AG-Op par rapport à lcc+sch et à color+sch ; ces améliorations sont illustrées aux figures 5.19 et 5.20 respectivement. Dans ces graphiques, les programmes de test sont triés en ordre croissant de leur complexité (m^n). Nous voyons que, pour les problèmes de plus petite complexité, les approches traditionnelles sont meilleures que AG-Op. Pour les problèmes de plus grande complexité, AG-Op entraîne toujours une amélioration par rapport aux approches traditionnelles, qui varie entre 0% et 26%. Ce comportement correspond très bien au comportement habituel des AGs, c'est-à-dire que leur performance ne sort pas de l'ordinaire sur les problèmes de petite taille mais deviennent particulièrement performants lorsque la taille du problème augmente.

La table 5.10 présente les améliorations obtenues par AG-Op par rapport au pire

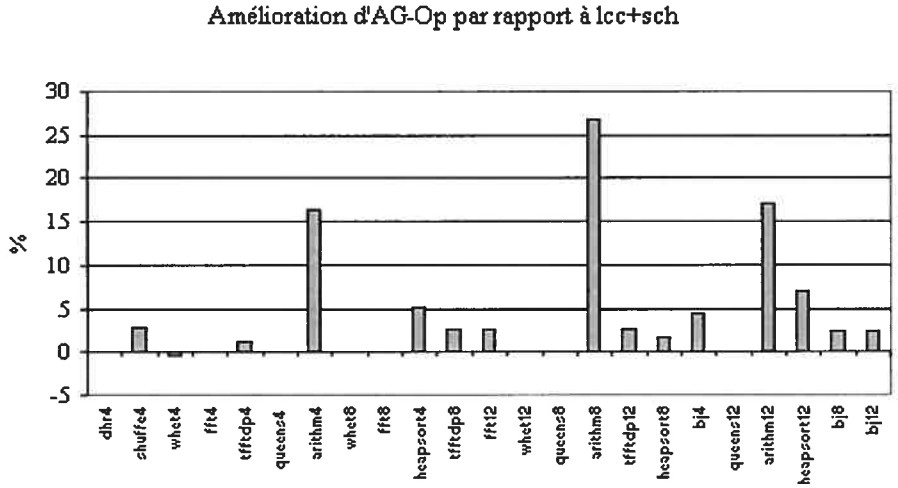


FIG. 5.19 – Pourcentage d'amélioration par rapport à lcc et l'ordonnancement par liste

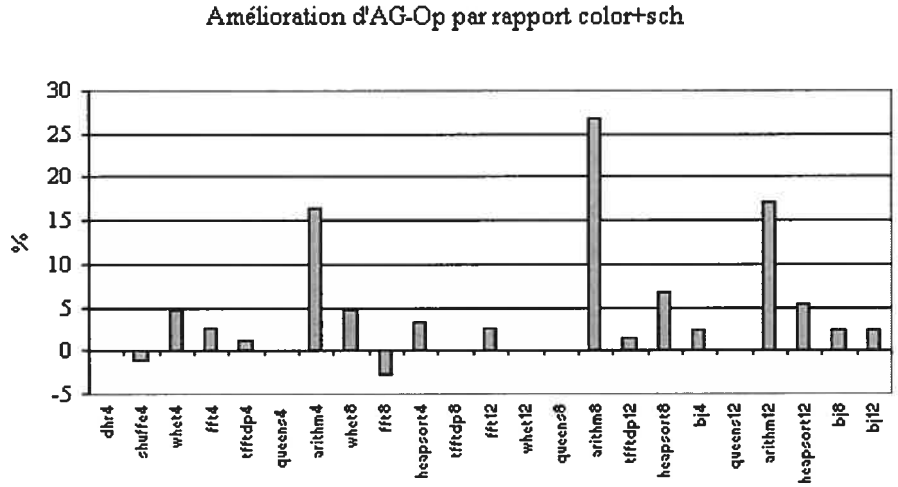


FIG. 5.20 – Pourcentage d'amélioration par rapport au coloriage et l'ordonnancement par liste

Programme de test	AG-Op/lcc+sch (%)	AG-Op/color+sch (%)
arithm4	16.5	16.5
arithm8	26.8	26.8
arithm12	17.2	17.2
bj4	4.5	2.3
bj8	2.3	2.3
bj12	2.3	2.3
dhr4	0.0	0.0
fft4	0.0	2.6
fft8	0.0	-2.7
fft12	2.6	2.6
heapsort4	5.2	3.3
heapsort8	1.6	6.8
heapsort12	6.9	5.4
queens4	0.0	0.0
queens8	0.0	0.0
queens12	0.0	0.0
shuffe4	2.8	-1.1
tfftdp4	1.3	1.3
tfftdp8	2.6	0.0
tfftdp12	2.7	1.4
whet4	-0.5	4.7
whet8	0.0	4.7
whet12	0.0	0.0

TAB. 5.9 – Amélioration obtenue par AG-Op par rapport à lcc+sch et color+sch

cas. Les résultats sont triés par pourcentage d'amélioration.

Nous voyons que les meilleures performances sont obtenues pour les programmes de test *queens* et *heapsort*. Ces programmes de test ont la caractéristique d'avoir un très petit nombre de procédures (1 et 2, respectivement) et un grand nombre de variables. Donc, l'allocation de registres a la possibilité d'avoir un impact plus important que dans les programmes plus modulaires qui ont beaucoup de fonctions avec peu de variables chacune. Ce comportement s'observe sur tous nos programmes de test. AG-Op obtient ses meilleures performances sur les programmes de test qui ont peu de procédures et qui

Programme de test	pire(s)	AG-Op (s)	AG/pire (%)
tfftdp4	8.2	7.6	7.3
shuffe4	40.6	37.5	7.6
bj8	4.6	4.2	8.7
bj4	4.6	4.2	8.7
bj12	4.6	4.2	8.7
tfftdp8	8.2	7.4	9.8
whet4	22.8	20.5	10.1
whet8	22.8	20.4	10.5
tfftdp12	8.2	7.2	12.2
fft4	4.4	3.8	13.6
fft8	4.4	3.8	13.6
whet12	22.6	19.1	15.5
fft12	4.4	3.7	15.9
dhr4	3.3	2.6	21.2
arithm4	13.4	10.6	20.9
arithm12	13.4	10.6	20.9
arithm8	13.4	9.3	30.6
queens4	0.6	0.4	33.3
queens12	0.6	0.4	33.3
queens8	0.6	0.4	33.3
heapsort4	25.9	14.5	44.0
heapsort8	25.9	12.3	52.5
heapsort12	25.9	12.2	52.9

TAB. 5.10 – Amélioration obtenue par AG-Op par rapport au pire cas

utilisent beaucoup de variables. Par la même occasion, le fait que le bloc de base soit plus grand permet à l'ordonnancement d'instructions d'apporter des améliorations plus importantes.

Même si de nos jours les programmes sont de plus en plus modulaires, le comportement d'AG-Op demeure très intéressant parce que le compilateur a toujours l'option de créer des grands blocs de base et de grandes procédures en effectuant quelques optimisations avant l'allocation de registres et l'ordonnancement d'instructions. Par exemple, le déroulement des boucles et l'*inlining* de fonctions. Ces optimisations peuvent être incluses à AG-Op ou bien être effectuées à l'avance. De plus, certains programmes générés

par d'autres programmes (par exemple le compilateur Gambit-C pour Scheme [Fee98]) ont tendance à avoir des grosses fonctions.

5.3 Résumé

D'après l'analyse de performance effectué sur AG-Op, nous pouvons affirmer :

- Le temps de traitement requis par AG-Op est beaucoup plus grand que le temps requis par les approches traditionnelles. Ainsi, l'utilisation d'AG-Op est viable seulement pour les usagers qui peuvent se permettre un temps de compilation élevé.
- Pour la plupart des programmes de test, AG-Op obtient des améliorations de performances par rapport aux méthodes traditionnelles. Ces gains varient entre 0% et 26%.
- Les gains les plus importants d'AG-Op se retrouvent pour des programmes qui utilisent un grand nombre de variables, qui ont un petit nombre de procédures et qui ont des grands blocs de base.

Chapitre 6

Conclusions et travaux futurs

6.1 Contributions

Les résultats de cette recherche permettent d'établir que l'utilisation de méthodes métaheuristiques en compilation est viable, profitable et qu'elle offre des avantages intéressants.

La viabilité

Les deux allocateurs de registres (un basé sur un algorithme génétique et l'autre sur un recuit simulé) développés et testés dans le chapitre 3 ont généré des résultats comparables par rapport à l'approche traditionnelle utilisée par GNU C. À la lumière de ces résultats, nous pouvons établir qu'il est viable d'utiliser les méthodes métaheuristiques pour résoudre ce type de problèmes.

Les résultats obtenus avec l'approche unifiée pour l'utilisation de méthodes métaheuristiques en compilation, telle que définie dans le chapitre 4 et testée pour les problèmes d'allocation de registres et d'ordonnancement d'instructions dans le chapitre 5, confirment la viabilité de l'approche unifiée. L'optimiseur basé sur un algorithme géné-

tique a une performance compétitive avec celle des approches plus réputées pour résoudre les problèmes choisis.

Le profit

L'analyse de performance de notre optimiseur basé sur un algorithme génétique, présenté dans le chapitre 5, démontre que l'utilisation de notre approche est profitable. L'optimiseur testé optimise seulement deux problèmes (allocation de registres et ordonnancement d'instruction) et obtient des améliorations intéressantes par rapport aux approches classiques :

- par rapport au pire cas (toutes les variables sont en mémoire et sans ordonnancement d'instructions), les gains de vitesse varient entre 7% et 53%.
- par rapport à l'allocation de registres par coloriage de graphe et ordonnancement d'instructions par liste, les gains de vitesse varient entre -2.7% et 26%.
- par rapport à l'allocation de registres de *lcc* et ordonnancement d'instructions par liste, les gains de vitesse varient entre -0.5% et 26%.

Les gains plus importants se retrouvent pour les programmes de test qui utilisent un grand nombre de variables, qui ont peu de procédures et qui ont des blocs de base plus grands. Ceci est dû au fait que dans ces programmes les optimisations ont un impact plus global.

Les avantages

La performance : Le premier avantage de l'utilisation de métaheuristiques en compilation est le gain de vitesse du code généré. Ces améliorations viennent principalement de trois sources :

- les métaheuristiques sont plus puissantes que les heuristiques pour résoudre des problèmes complexes
- la résolution de plus d'un problème à la fois permet de tenir compte des interdé-

pendances

- l'utilisation du temps réel d'exécution comme fonction de coût permet d'avoir une mesure précise de la qualité de la solution

De plus, ces gains sont obtenus en limitant l'optimiseur à deux problèmes ; si nous ajoutons plus de problèmes nous pouvons nous attendre à des gains de performance supérieurs.

L'extensibilité : La structure de l'optimiseur présentée dans le chapitre 4 est conçue de façon générale, c'est-à-dire qu'elle permet de résoudre plusieurs problèmes simultanément. Nous l'avons testé pour les problèmes d'allocation de registres et d'ordonnancement d'instructions, mais il est possible d'y ajouter plusieurs autres problèmes sans changer le fonctionnement de l'optimiseur. De plus, la représentation du problème pour l'allocation et l'ordonnancement a été défini, elle aussi, de la façon la plus générale possible. Donc, il n'y a pas de changements nécessaires pour ajouter des problèmes comme la sélection d'instructions ou l'ordonnancement d'instructions avant l'allocation de registres.

L'homogénéité : L'optimiseur basé sur une métaheuristique, grâce à l'utilisation du temps réel comme fonction de coût, a un comportement plus homogène que les heuristiques pour tout l'ensemble des programmes de test. Dans les résultats présentés dans le chapitre 5, nous observons que pour un programme test qui dévie de la normale, les heuristiques classiques n'arrivent pas à trouver une bonne solution ; par contre, la métaheuristique a une très bonne performance.

La simplicité : En utilisant le temps réel d'exécution comme fonction de coût notre optimiseur n'a besoin que de très peu d'informations sur la machine. Considérant la complexité croissante des architectures actuelles cette caractéristique est très intéressante parce que la construction du compilateur devient beaucoup plus simple. De même, le compilateur pourra s'adapter facilement aux changements des architectures.

L'utilisation du temps réel d'exécution permet, aussi, de mesurer les effets secondaires de différentes optimisations, ce qui n'est pas possible avec les heuristiques traditionnelles.

Le coût : Un désavantage de notre approche est le temps de compilation. Le temps de traitement requis par AG-Op est beaucoup plus grand que le temps requis par les approches traditionnelles. Ainsi, l'utilisation d'AG-Op est viable seulement pour les usagers qui peuvent se permettre un temps de compilation élevé.

6.2 Travaux futurs

Ce travail est un des premiers à considérer l'utilisation de méthodes métaheuristiques en compilation et il indique que cette utilisation est viable et avantageuse. Il faut voir ce travail comme le point de départ pour beaucoup de futures recherches. Nous envisageons des travaux futurs qui sont décrits brièvement ci-dessous.

Résoudre les problèmes plus globalement

Notre prototype résout les problèmes d'allocation de registres et d'ordonnancement d'instructions localement. L'allocation de registres est effectuée dans chaque procédure et l'ordonnancement d'instructions dans chaque bloc de base. Ceci limite les possibilités d'optimisation de notre approche. Ainsi, nous croyons fondamental de généraliser notre prototype afin de pouvoir effectuer des optimisations plus globales.

Ajouter d'autres optimisations

Nous pouvons ajouter d'autres optimisations à notre approche. La sélection d'instructions et un ordonnancement avant l'allocation de registres sont les premiers candidats, puisque leur impact sur la qualité du code généré est très importante. Étant donné que nous avons considéré ces optimisations dans la définition de la représentation des pro-

blèmes d'allocation de registres et d'ordonnancement d'instructions après l'allocation de registres, nous pouvons les utiliser tel que définis. Il faudra trouver des représentations et des opérateurs génétiques appropriés pour les nouvelles optimisations.

Tester d'autres représentations

Même si intuitivement les choix des représentations pour les problèmes d'ordonnancement d'instructions et d'allocation de registres semblent les plus appropriés, nous pouvons concevoir d'autres représentations afin de les comparer. Parfois une différence dans la représentation utilisée pour les métaheuristiques implique des différences importantes dans la performance de la métaheuristiques soit pour la qualité de la solution obtenue soit pour le temps de traitement requis.

Compiler vers d'autres architectures

Il serait intéressant de compiler vers d'autres architectures, comme l'architecture Intel, qui a un ensemble de registres plus petit et un jeu d'instructions plus complexe, ce qui fait que les problèmes d'allocation de registres et sélection d'instructions ont un impact plus important sur la qualité du code généré.

Tester d'autres métaheuristiques

Nous pouvons implanter d'autres méthodes métaheuristiques. Celles qui sont intéressantes en fonction de ses bonnes performances sont la méthode de recherche tabou, les méthodes hybrides (tabou et recherche locale, par exemple) et les algorithmes génétiques parallèles. Il est possible d'utiliser les mêmes représentations pour les problèmes et la même fonction d'évaluation. Il serait intéressant de réaliser une analyse comparative des différentes méthodes puisqu'il n'est pas clair laquelle est la meilleure.

Etudier les types de données d'entrée requis

Notre approche mesure le temps réel d'exécution de chaque programme généré par la métaheuristique ; pour ce faire, elle doit connaître les données d'entrée du programme. Bien sur, l'usager ne veut pas compiler ses programmes avec chaque ensemble de données d'entrée. L'idéal serait de compiler le programme en utilisant des ensembles petits mais représentatifs du comportement des données avec lesquelles le programme sera utilisé. Trouver cette sorte de données d'entrée pourrait apporter des améliorations intéressantes à notre approche. On pourrait diminuer le temps de compilation en assurant une bonne performance dans les cas réels.

Construire un compilateur

La construction d'un compilateur qui génère une représentation intermédiaire avec toutes les informations nécessaires pour l'approche unifiée permettrait de simplifier l'implantation, diminuer le temps compilation et faciliter l'inclusion d'autres problèmes.

Un tel compilateur devrait :

- Générer un code intermédiaire pour le programme au complet permettant ainsi de réaliser les optimisations plus globalement.
- Avoir au niveau du code intermédiaire l'information de flux du programme pour pouvoir obtenir les différentes données requises pour résoudre chaque problème. Par exemple, le graphe d'interférence de variables, le graphe de dépendances d'instructions et le graphe des appels de fonctions.

Étendre notre approche à la compilation dynamique

Une approche semblable à celle proposée peut tirer profit des compilations multiples dans un système de compilation dynamique. Pour optimiser une bibliothèque, le compilateur maintiendrait d'un compilation à l'autre une population et il profiterait chaque

fois qu'une procédure est exécutée pour évaluer un individu différent de la population. Ainsi, après quelque temps on aurait des procédures optimisées en considérant les données d'entrée habituelles pour cette procédure.

Bibliographie

[App96] Andrew W. Appel. http ://www.cs.princeton.edu/~appel/graphdata, 1996.

[App97] Andrew W. Appel. *Modern Compiler Implementation in ML : Basic Techniques.* Cambridge University Press, 1997.

[AS87] Andrew W. Appel and Kenneth J. Supowit. Generalization of the Sethi-Ullman Algorithm for Register Allocation. *Software - Practice and Experience*, pages 417–421, 1987.

[AU78] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1978.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. *ACM SIGPLAN Notices*, pages 275–284, 1989.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *ACM SIGPLAN Notices*, pages 311–321, 1992.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 428–455, 1994.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill Code Minimization Via Interference Region Spilling. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[Bea91] Steven J. Beaty. Genetic Algorithms and Instruction Scheduling. *24th Annual International Symposium on Microarchitecture*, pages 206–211, 1991.

[Bea93] S. J. Beaty. Genetic Algorithms Versus Tabu Search for Instruction Scheduling. *International Conference on Neural Network and Genetic Algorithms*, pages 496–501, 1993.

[BGG+89] D. Bernstein, D. Q. Goldin, M. C. Golumbici, H. Krawczykand Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263, 1989.

[BGS98] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Integrated Instruction Scheduling and Register Allocation Techniques. *Languages and Compilers for Parallel Computing*, pages 247–262, 1998.

[CAC+82] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation Via Coloring. *Computer Languages*, pages 47–57, 1982.

[CH90] F. C. Chow and J. L. Hennessy. The Priority-based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 501–536, 1990.

[CHdW87] M. Chams, A. Hertz, and D. de Werra. Some Experiments with Simulated Annealing for Coloring Graphs. *European Journal of Operational Research*, pages 260–266, 1987.

[CK91] David Callahan and Brian Koblenz. Register Allocation Via Hierarchical Graph Coloring. *Conference on Programming Language Design and Implementation*, pages 192–203, 1991.

[CLG02] Josep M. Codina, Josep Llosa, and Antonio González. A Comparative Study of Modulo Scheduling Techniques. *16th International Conference on Supercomputing*, pages 97–106, 2002.

[CS98] K. D. Cooper and L. T. Simpson. Live Range Splitting in a Graph Coloring Register Allocator. *Lecture Notes in Computer Science*, pages 174–187, 1998.

[CSS98] Keith Cooper, Philip Schielke, and Devika Subramanian. An Experimental Evaluation of List Scheduling. Technical Report TR98-326, Rice University, 1998.

[CW94] Arthur L. Corcoran and Roger L. Wainwright. A Parallel Island Model Genetic Algorithm for the Multiprocessor Scheduling Problem. *Selected Areas in Cryptography*, pages 483–487, 1994.

[Dav91] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[DMC91] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System : An Autocatalytic Optimizing Process. Technical Report TR.03-IEEE-SMC 91-016, Politecnico di Milano, 1991.

[EDA96] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing Register Requirements of a Modulo Schedule Via Optimum Stage Scheduling. *International Journal of Parallel Programming*, pages 103–132, 1996.

[EvdH96] A. Eiben and J. van der Hau. Graph Coloring with Adaptive Genetic Algorithm. Technical Report TR 96-11, Leiden University, 1996.

[Fee98] M. Feeley. http ://www.iro.umontreal.ca/~gambit/, 1998.

[FF97] C. Fleurent and J. A. Ferland. Genetic and Hybrid Algorithms for Graph Coloring. *Annals of Operations Research*, pages 437–461, 1997.

[FH92] Christopher W. Fraser and David R. Hanson. Simple Register Spilling in a Retargetable Compiler. *Software - Practice and Experience*, pages 85–99, 1992.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler : Design and Implementation.* Benjamin/Cummings Pub. Co., 1995.

[FL98] Martin Farach and Vincenzo Liberatore. On Local Register Allocation. *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–573, 1998.

[FRC93] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Re-Scheduling, and Open-Shop Scheduling Problems. *5th International Conference on Genetic Algorithms*, pages 375–382, 1993.

[GA96] Lal George and Andrew W. Appel. Iterated Register Coalescing. *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 208–218, 1996.

[GDC92] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic Algorithms, Noise, and the Sizing of Populations. *Complex Systems*, pages 333–362, 1992.

[GH88] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. *2nd International Conference on Supercomputing*, pages 442–452, 1988.

[GH99] Phillippe Galinier and Jin-Kao Hao. Hybrid Evolutionary Algorithms for Graph Coloring. *Journal of Combinatorial Optimization*, pages 379–397, 1999.

[GL95] F. Glover and M. Laguna. Tabu Search. *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150, 1995.

[GM86] Philip B. Gibbons and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. *SIGPLAN Symposium on Compiler Contruction*, pages 11–16, 1986.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

[Hol75] John H. Holland. *Adaptation in Natural Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, second edition, 1996.

[Huf93] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[JAMS91] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by Simulated Annealing : An Experimental Evaluation ; Part II, Graph Coloring and Number Partitioning. *Operations Research*, pages 378–406, 1991.

[KF96] Steven M. Kurlander and Charles N. Fischer. Minimum Cost Interprocedural Register Allocation. *Symposium on Principles of Programming Languages*, pages 230–241, 1996.

[KGV83] S. Kirpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science 220*, pages 671–680, 1983.

[KPF95] Steven M. Kurlander, Todd A. Proebsting, and Charles N. Fischer. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 740–776, 1995.

[Kri96] F. Kri. Modelos Paralelos de Algoritmos Genéticos. Master's thesis, University of Santiago of Chile, 1996.

[LFK97] Vincenzo Liberatore, Martin Farach, and Ulrich Kreme. Hardness and Algorithms for Local Register Allocation. Technical Report DCS-TR-332, Case Western Reserve University, 1997.

[LGAT96] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Global Register Allocation Based on Graph Fusion. *Languages and Compilers for Parallel Computing*, pages 246–265, 1996.

[LGAT00] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-Based Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 431–470, 2000.

[LPU95] A. Leung, K. Palem, and C. Ungureanu. Run-Time versus Compile-Time Instruction Scheduling in Superscalar (RISC) Processors : Performance and Tradeoffs. Technical Report TR1995-699, New York University, 1995.

[Lue96] Guei-Yuan Lueh. Issues in Register Allocation by Graph Coloring. Technical Report CMU-CS-96-171, School of Computer Science, 1996.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[NG93] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1993.

[Nis98] Andy P. Nisbet. GAPS : Genetic Algorithm Optimised Parallelisation. *7th Workshop on Compilers for Parallel Computing*, pages 172–183, 1998.

[NP95] Cindy Norris and Lori L. Pollock. An Experimental Study of Several Cooperative Register Allocation and Instruction Scheduling Strategies. *28th Annual International Symposium on Microarchitecture*, pages 169–179, 1995.

[Pin95] M. Pinedo. *Scheduling : Theory, Algorithms, and Systems.* Prentice-Hall, Englewood Cliffs, 1995.

[PL00] Martin Pelikan and Fernando G. Lobo. Parameter-Less Genetic Algorithm : A Worst-Case Time and Space Complexity Analysis. *Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 370, 2000.

[Pot96] Jean-Yves Potvin. Genetic Algorithms for the Traveling Salesman Problem. *Annals of Operations Research*, pages 339–370, 1996.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 895–913, 1999.

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling : an Algorithm for Software Pipelining Loops. *27th Annual International Symposium on Microarchitecture*, pages 63–74, 1994.

[SBS94] A. Shahid, M.S.T. Benten, and S.M. Sait. GSA : Scheduling and Allocation Using Genetic Algorithm. *European Design Automation Conference (EURO-DAC94)*, pages 84–89, 1994.

[SdRS97] A. Schoneveld, J.F. de Ronde, and P.M.A. Sloot. Task Allocation by Parallel Evolutionary Computing. *Journal of Parallel and Distributed Computing*, pages 91–97, 1997.

[Sta91] Stallman. GNU C User and Porting Guide. Technical report, MIT, 1991.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-Scan Register Allocation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[Tom67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, pages 25–33, 1967.

[VAL94] R. Vaessens, E. Aarts, and J. Lenstra. Job-Shop Scheduling by Local Search. Technical Report COSOR 94-05, Eindhoven University of Technology, 1994.

[VG99] Madhavi Gopal Valluri and R. Govindarajan. Evaluating Register Allocation and Instruction Scheduling Techniques in Out-of-Order Issue Processors. *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 78–83, 1999.

[Wal86] D. Wall. Global Register Allocation at Link Time. *7th SIGPLAN Symposium on Compiler Construction*, pages 264–275, 1986.

[Wan83] P. Wang. Two Algorithms for Constraint Two-Dimensional Cutting Stock Problems. *Operations Research*, pages 573–586, 1983.

[WLH00] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal Instruction Scheduling Using Integer Programming. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2000.

[WW96] K. Williams and S. William. Genetic Compilers : A New Technique for Automatic Parallelisation. *2nd European School of Parallel Programming Environments (ESPPE 96)*, pages 27–30, 1996.

[ZLAV01] J. Zalamea, J. Llosa, E. Ayguad'e, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. *34th International Symposium on Microarchitecture (MICRO-34)*, pages 160–169, 2001.

Annexe A

Résultat de l'ajustement de paramètres

Dans cette section nous présentons les résultats obtenus dans le processus d'ajustement de paramètres d'AG-Allocateur et de RS-Allocateur. Pour chaque paramètre testé nous présentons le nombre d'individus évalués (*Ind*) et le coût de la meilleure solution trouvé par l'algorithme (*Coût*).

A.1 AG-Allocateur

La table A.1 présente les résultats obtenus en utilisant l'opérateur de mutation d'échange (mutation 1).

Graphe	Mut=1%		Mut=5%		Mut=10%	
	Ind	Coût	Ind	Coût	Ind	Coût
g-96	7900	4037	5500	4547	3400	4037
graph42	9800	1028	6876	987	4201	925
graph61	11900	1015	11400	1006	13400	771
graph99	8600	2697	7410	2420	5470	2152

TAB. A.1 – Résultats opérateur de mutation 1 : échange

La table A.2 présente les résultats obtenus en utilisant l'opérateur de mutation de descente (mutation2) avec une taille de population de 100 individus.

Graphe	Mut=1%		Mut=5%		Mut=10%	
	Ind	Coût	Ind	Coût	Ind	Coût
g-96	19602	15	5743	48	3600	2
graph42	3957	21	3642	18	2946	18
graph61	6017	21	5694	21	4026	21
graph99	5046	167	4527	172	3872	167

TAB. A.2 – Résultats opérateur de mutation 2 : descente. TP=100

La table A.3 présente les résultats obtenus en utilisant l'opérateur de mutation de descente (mutation2) avec une taille de population de 50 individus.

Graphe	Mut=1%		Mut=5%		Mut=10%		Mut=50%		Mut=100%	
	Ind	Coût	Ind	Coût	Ind	Coût	Ind	Coût	Ind	Coût
g-96	11228	2552	13623	1258	14864	21	5400	2	12089	15
graph42	8651	956	8025	873	9627	16	4403	18	9683	21
graph61	7998	1082	8111	946	8932	36	4870	21	9987	16
graph99	12267	2354	10628	2160	14560	172	6786	167	14970	171

TAB. A.3 – Résultats opérateur de mutation 2 : descente. TP=50

La table A.4 présente les résultats obtenus en utilisant l'opérateur de croisement OBX et PBX.

Graphe	OBX		PBX	
	Ind	Coût	Ind	Coût
g-96	3862	21	7600	21
graph42	3498	18	4291	21
graph61	5226	0	5100	212
graph99	3604	172	4803	196

TAB. A.4 – Résultats opérateur de croisement

La table A.5 présente les résultats obtenus en utilisant différentes tailles de population : n, $n/2$ et $2n$ où n est le nombre de nœuds du graphe.

Graphe	TP=n		TP=n/2		TP=2n	
	Ind	Coût	Ind	Coût	Ind	Coût
g-96	3900	21	8668	25	16992	25
graph42	3504	21	4951	34	10564	21
graph61	4871	21	5605	21	12067	18
graph99	4002	167	7964	178	14672	167

TAB. A.5 – Résultats pour différentes tailles de population

A.2 RS-Allocateur

La table A.6 présente les résultats obtenus en utilisant différentes températures initiales et finales. Pour la température initiale nous utilisons $5n$ et $10n$, où n est le nombre de nœuds du graphe.

Graphe	Ti=5n,Tf=0,1		Ti=10n,Tf=0,1		Ti=5n, Tf=0,01		Ti=10n, Tf=0,01	
	Ind.	Coût	Ind.	Coût	Ind	Coût	Ind.	Coût
g-96	32500	4	37850	8	43800	4	36650	4
graph42	15942	8	18700	12	19870	21	17945	8
graph61	30500	11	35151	11	49550	11	31100	11
graph99	33257	31	35482	36	37025	45	36725	31

TAB. A.6 – Résultats pour différentes températures

Annexe B

Code assembleur généré par GNU C et RS-Allocateur

Dans cette section nous présentons les codes assembleur pour la procédure de *wan* pour laquelle RS-Allocateur trouve une allocation de registres différente de GNU C. Cette différence est résponsable des gains de vitesse du code généré que varient entre 0% et 2% dépendant des données fournies au programme.

GNU C

```
.globl existe
    .type   existe,@function
existe:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    pushl %edi
    pushl %esi
    pushl %ebx
    movb $0,-2(%ebp)
    movl Lk,%ebx
    .p2align 4,,7
.L368:
    movl 8(%ebp),%eax
    flds 4(%ebx)
    flds 4(%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L360
    movl 8(%ebp),%eax
    flds (%ebx)
    flds (%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L360
    movb $0,-1(%ebp)
    xorl %edx,%edx
    leal 12(%ebx),%edi
    movl 8(%ebp),%esi
    addl $12,%esi
    movl n,%ecx
    decl %ecx
    .p2align 4,,7
.L365:
    movl (%edi,%edx,4),%eax
    cmpl (%esi,%edx,4),%eax
    je .L364
    movb $1,-1(%ebp)
.L364:
    incl %edx
    cmpl %ecx,%edx
    ja .L362
    cmpb $0,-1(%ebp)
    je .L365
.L362:
    cmpb $0,-1(%ebp)
    jne .L360
    movb $1,-2(%ebp)
.L360:
    movl 1212(%ebx),%ebx
    testl %ebx,%ebx
    je .L358
    cmpb $0,-2(%ebp)
    je .L368
```

RS–Allocateur

```
.globl existe
    .type   existe,@function
existe:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    pushl %edi
    pushl %esi
    pushl %ebx
    movb $0,-2(%ebp)
    movl Lk,%ebx
    movl %ebx,-8(%ebp)
    .p2align 4,,7
.L368:
    movl -8(%ebp),%ebx
    movl 8(%ebp),%eax
    flds 4(%ebx)
    flds 4(%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L360
    movl -8(%ebp),%ebx
    movl 8(%ebp),%eax
    flds (%ebx)
    flds (%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L360
    movb $0,-1(%ebp)
    xorl %edx,%edx
    leal 12(%ebx),%edi
    movl 8(%ebp),%esi
    addl $12,%esi
    movl n,%ecx
    decl %ecx
    .p2align 4,,7
.L365:
    movl (%edi,%edx,4),%eax
    cmpl (%esi,%edx,4),%eax
    je .L364
    movb $1,-1(%ebp)
.L364:
    incl %edx
    cmpl %ecx,%edx
    ja .L362
    cmpb $0,-1(%ebp)
    je .L365
.L362:
    cmpb $0,-1(%ebp)
    jne .L360
    movb $1,-2(%ebp)
.L360:
    movl 1212(%ebx),%ebx
    movl %ebx,-8(%ebp)
    testl %ebx,%ebx
    je .L358
    cmpb $0,-2(%ebp)
    je .L368
```

GNU C
(suite)

```
.L358:
    cmpb $0,-2(%ebp)
    jne .L370
    cmpl $0,f
    je .L370
    movl f,%ebx
    .p2align 4,,7
.L382:
    movl 8(%ebp),%eax
    flds 4(%ebx)
    flds 4(%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L374
    movl 8(%ebp),%eax
    flds (%ebx)
    flds (%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L374
    movb $0,-1(%ebp)
    xorl %edx,%edx
    leal 12(%ebx),%edi
    movl 8(%ebp),%esi
    addl $12,%esi
    movl n,%ecx
    decl %ecx
    .p2align 4,,7
.L379:
    movl (%edi,%edx,4),%eax
    cmpl (%esi,%edx,4),%eax
    je .L378
    movb $1,-1(%ebp)
.L378:
    incl %edx
    cmpl %ecx,%edx
    ja .L376
    cmpb $0,-1(%ebp)
    je .L379
.L376:
    cmpb $0,-1(%ebp)
    jne .L374
    movb $1,-2(%ebp)
.L374:
    movl 1212(%ebx),%ebx
    testl %ebx,%ebx
    je .L370
    cmpb $0,-2(%ebp)
    je .L382
.L370:
    movzbl -2(%ebp),%eax
    popl %ebx
    popl %esi
    popl %edi
    movl %ebp,%esp
    popl %ebp
    ret
```

RS-Allocateur
(suite)

```
.L358:
    cmpb $0,-2(%ebp)
    jne .L370
    cmpl $0,f
    je .L370
    movl f,%ebx
    .p2align 4,,7
.L382:
    movl -8(%ebp),%ebx
    movl 8(%ebp),%eax
    flds 4(%ebx)
    flds 4(%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L374
    movl -8(%ebp),%ebx
    movl 8(%ebp),%eax
    flds (%ebx)
    flds (%eax)
    fucompp
    fnstsw %ax
    andb $68,%ah
    xorb $64,%ah
    jne .L374
    movb $0,-1(%ebp)
    xorl %edx,%edx
    leal 12(%ebx),%edi
    movl 8(%ebp),%esi
    addl $12,%esi
    movl n,%ecx
    decl %ecx
    .p2align 4,,7
.L379:
    movl (%edi,%edx,4),%eax
    cmpl (%esi,%edx,4),%eax
    je .L378
    movb $1,-1(%ebp)
.L378:
    incl %edx
    cmpl %ecx,%edx
    ja .L376
    cmpb $0,-1(%ebp)
    je .L379
.L376:
    cmpb $0,-1(%ebp)
    jne .L374
    movb $1,-2(%ebp)
.L374:
    movl 1212(%ebx),%ebx
    movl %ebx,-8(%ebp)
    testl %ebx,%ebx
    je .L370
    cmpb $0,-2(%ebp)
    je .L382
.L370:
    movzbl -2(%ebp),%eax
    popl %ebx
    popl %esi
    popl %edi
    movl %ebp,%esp
    popl %ebp
    ret
```

Annexe C

L'implantation d'AG-Op

Nous présentons dans cette section quelque détailles importants sur l'implantation d'AG-Op.

Les structures de donnés pour les individus : Un individu est formé de deux sous-individus, un pour le problème d'allocation de registres et l'autre pour le problème d'ordonnancement d'instruction.

```
struct individu
{
        Ind1 IndAlloc ;   /* L'individu pour l'allocation de registres */
        Ind2 IndSch ;     /* L'individu pour l'ordonnancement d'instructions */
        double cout ;
}Individu ;
```

L'individu utilisé pour l'allocation de registres est une chaîne de nombres entiers.

```
struct ind1
{
        short int *chromo ; /* Un chaîne de nombres entiers */
        double cout ;       /* Le coût de la solution */
        int longueur ;      /* La longueur de l'individu */
}Ind1 ;
```

L'individu utilisé pour l'ordonnancement d'instruction est forme pour un vecteur d'individus simples (*ind1*).

```
struct ind2
{
        Ind1 *Bb;          /* Un vecteur d'individus */
        double cout;       /* Le coût de la solution */
        int *longeurBb;    /* La longueur de chaque bloc de base */
        int NombreBb;      /* Le nombre de blocs de base */
}Ind2;
```

La structure de donné pour la population :

```
struct population
{
   Individu *habitants;     /* vecteur avec les habitants */
   int TaillePopulation;
}Population;
```

La structure de donné pour l'algorithme génétique : Étant donné que AG-Op effectue l'optimisations dans chaque procédure du programme, nous avons besoins d'un AG par chaque procéedure. Alors, nous definisons la structure AG et utilisons un liste d'AGs.

```
typedef struct ag
{
        char *nom;                  /* Le nom de la fonction */
        Individu IndInitial;        /* L'individu initial */
        Population Pob;             /* La population */
        Mat *Inferference;          /* Le graphe d'interference pour le problème */
                                    /* d'allocation de registres*/
        Mat **Dependance;           /* Une liste de graphes de depéndence, un pour */
                                    /* chaque bloc de base */
        int nVar;                   /* Le nombre de variables dans le procedure */
        double *Ref;                /* Le nombre de fois que chaque variable*/
                                    /* est reférence */
                                    /* Les paramètres de l'AG */
        int Generations;                    /* Nombre de générations */
        int TaillePop;                      /* La taille de la population */
        double Selection;                /* Le pourcentache de sélection */
        double ProbMutation;                /* La probabilité de mutation */
}AG;

typedef struct listag
{
        AG *Genetic;                /* Une liste d'AGs, une pour chaque procedure */
        int nAG;                    /* Le nombre de procedures du programme */
}ListAG;
```

Le pseudocode pour la fonction de coût :

```
EvaluerPopulation()
{
    Pour chaque individu ind[i]  de la population
    /* ind[i].IndAlloc = sous-individu allocation */
    /* ind[i].IndSch = sous-individu ordonnancement */
    {
        /* génère l'allocation de registres à partir du sous-individu */
        /* pour le problème d'allocation de registres */
            allocation=HeuristiqueVorace(ind[i].IndAlloc);

        /* génère du code assembleur avec une allocation donne */
            code_assembleur=lcc2(allocation);

        /* génère le graphe de dependence */
            graphedep=Gen_Dep(codeassembleur);

        /* répare l'ordonnancement d'instructions */
            scheduling=Fonctionreparation(ind[i].IndSch,graphedep);

        /* génère le code objet à partir d'une allocation et un ordonnancemnt données*/
            codeobjet=lcc3(allocation, scheduling);

        /* mesure le temps de exécution du programme */
            t=TempExecution(executer(codeobjet));
    }
}
```

Le pseudocode pour la fonction de réparation pour l'ordonnancement d'instruction :

```
FonctionReparation(Ind1 ind, Graph_dependence G)
{
    PasPret= liste;

    IndSch=ind.IndSch;
    l=ind.IndSch.longueur;

    Pour i=0 à l {
        Pour chaque element n ∈ PasPret     /* Pour chaqe instructions en attante */
                                            /* verifier si peut être exécuté */
            Si (estPret(n,G)){              /* si oui, l'assigner */
                PasPret=PasPret-{n};
                final[k++]=n;
            }
        m=IndSch[i];                        /* Pour chaque instructions dans l'individu */
        Si estPret(m,G);                    /* si elle peut être exécuté */
            final[k++]=m;                   /* l'assigner */
        sinon                               /* sinon */
            PasPret=PasPret ∪ {m}           /* la faire attendre dans PasPret */
    }
}

ou
estPret(n,G) determine si l'instruction n peut etre exécute. Elle
verifie si tous les instruction dependant ont été déja assignées.
```

Le pseudocode pour les opérateurs de croisement PBX :

```
PBX(Ind1 pere1, Ind1 pere2)
{
    Pos=ensemble;
    Elem=ensemble;

    l=p1.longueur;          /* longeur du individu */
    n=random(1,l);          /* Nombre de position fixes */

    Pour i=0 à n {
        m=random(0,1);                              /* Position fixe*/
        Elem=Elem ∪ {pere1.chromo[m]};
        Pos=Pos ∪ {m};
        enfant[m]=pere1.chromo[m];     /* Mettre dans les positions fixes chez */
    }                                  /* l'enfant les valeur de le même positions */
                                       /* chez le premier parent */
    k=0;
    Tant que (i<l) {   /* Completer les autres positions chez l'enfant */
        Si (i ∈ Pos)   /* Si est un position fixe, l'ignorer */
            i++;
        sinon
            Si (pere2.chromo[k] ∈ Elem)     /* Si l'element est déjà dans */
                k++;                        /* l'enfant, l'ignorer */
            sinon
                enfant[i++]=pere2.chromo[k++];   /* Mettre dans l'enfant la valeur du */
    }                                            /* deuxime parent */
}
```

Le pseudocode pour les opérateurs de croisement OBX :

```
OBX(Ind1 pere1, Ind1 pere2)
{
    Pos=ensemble;
    Elem=queue FIFO;

    l=p1.longueur;          /* longeur du individu */
    n=random(1,l);          /* Nombre de position fixes */

    Pour i=0 à n {
        m=random(0,1);                              /* Position fixe*/
        Elem=Elem ∪ {pere1.chromo[m]};              /* Mettre valeur de position fixes du */
                                                    /* premier parent dans Elem */
    }
    Pour i=0 à n
        Si pere2.chromo[i] ∈ Elem       /* Si la valeur chez le deuxime parent */
            Pos=Pos ∪ {i}               /* est dans Elem metre la position dans Pos */
    k=0;
    Tant que (j<l) {                        /* Pour chaque position chez l'enfant */
        Si j ∈ Pos                          / * Si la position est fixe */
            h[j++]=Enlever(Elem);           /* prendre le premièr element de Elem */
        sinon
            Si pere2.chromo[k] ∈ Elem   /* si la valeur chez le deuxieme */
                k++;                    /* parent est dans Elem l'ignorer */
            sinon
                h[j++]=pere2.chromo[k++]    /* Mettre dans l'enfant la valeur */
    }                                       /* du deuxime parent */
}
```

Annexe D

Résultat de l'ajustement de paramètres pour AG-Op

Dans cette section nous présentons les résultats obtenus dans le processus d'ajustement de paramètres d'AG-Op. Pour chaque paramètre testé nous présentons le nombre d'individus évalués (*Ind*) et le coût de la meilleure solution trouvée par l'algorithme (*Coût*).

La table D.1 présente les résultats obtenus en utilisant différents individus initiaux avec une taille de population de 10 individus et une probabilité de mutation de 1%.

Programme de test	Ini1		Ini2		Ini	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	99	4.22	76	4.22	100	4.41
heapsort	101	13.56	92	13.09	125	13.7
fftdp	96	7.62	80	7.5	110	7.41

TAB. D.1 – Résultats obtenus pour les différents individus initiaux. TP=10, Mut=1%

La table D.2 présente les résultats obtenus en utilisant différents individus initiaux avec une taille de population de 10 individus et une probabilité de mutation de 5%.

Programme de test	Ini1		Ini2		Ini3	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	92	4.2	89	4.2	100	4.39
heapsort	97	13.16	85	13.03	121	13.75
fftdp	83	7.59	72	7.49	105	7.44

TAB. D.2 – Résultats obtenus pour les différents individus initiaux. TP=10, Mut=5%

La table D.3 présente les résultats obtenus en utilisant différents individus initiaux avec une taille de population de 20 individus et une probabilité de mutation de 1%.

Programme de test	Ini1		Ini2		Ini3	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	186	4.22	158	4.21	201	4.38
heapsort	232	13.21	190	13.02	205	13.74
fftdp	181	7.5	154	7.41	222	7.4

TAB. D.3 – Résultats obtenus pour les différents individus initiaux. TP=20, Mut=1%

La table D.4 présente les résultats obtenus en utilisant différents individus initiaux avec une taille de population de 20 individus et une probabilité de mutation de 5%.

Programme de test	Ini1		Ini2		Ini3	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	196	4.2	148	4.2	190	4.37
heapsort	207	13.95	98	12.6	194	13.74
fftdp	112	7.5	72	7.4	208	7.4

TAB. D.4 – Résultats obtenus pour les différents individus initiaux. TP=20, Mut=5%

La table D.5 présente les résultats obtenus en utilisant différents opérateurs de croisement pour le problème d'allocation de registres avec une population de taille 10 individus.

Programme de test	OBX		PBX	
	Ind	Coût	Ind	Coût
bj	93	4.39	99	4.39
heapsort	69	13.5	70	13.67
fftdp	53	7.9	67	7.9

TAB. D.5 – Résultats obtenus pour les différents opérateurs de croisements pour le problème d'allocation de registres. TP=10

La table D.6 présente les résultats obtenus en utilisant différents opérateurs de croisement pour le problème d'allocation de registres avec une population de taille 20 individus.

Programme de test	OBX		PBX	
	Ind	Coût	Ind	Coût
bj	154	4.39	184	4.4
heapsort	148	13.52	153	13.69
fftdp	126	7.84	135	7.91

TAB. D.6 – Résultats obtenus pour les différents opérateurs de croisements pour le problème d'allocation de registres. TP=20

La table D.7 présente les résultats obtenus en utilisant différents opérateurs de croisement pour le problème d'ordonnancement d'instructions avec une population de taille 10 individus.

Programme de test	OBX		PBX	
	Ind	Coût	Ind	Coût
bj	90	4.2	98	4.23
heapsort	65	12.9	79	13
fftdp	57	7.5	71	7.54

TAB. D.7 – Résultats obtenus pour les différents opérateurs de croisements pour le problème d'ordonnancement d'instructions. TP=10

La table D.8 présente les résultats obtenus en utilisant différents opérateurs de croisement pour le problème d'ordonnancement d'instructions avec une population de taille 20 individus.

Programme de test	OBX		PBX	
	Ind	Coût	Ind	Coût
bj	148	4.2	192	4.24
heapsort	120	12.99	160	13.11
fftdp	97	7.56	131	7.61

TAB. D.8 – Résultats obtenus pour les différents opérateurs de croisements pour le problème d'ordonnancement d'instructions. TP=20

La table D.9 présente les résultats obtenus en utilisant différentes tailles de population : $n/2$, n et $2n$, où n est le nombre maximal de variables du programme.

Programme de test	0.5MAXVAR		MAXVAR		2MAXVAR	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	163	4.24	352	4.24	448	4.24
heapsort	99	13.06	230	13.06	284	13
fftdp	76	7.6	160	7.7	183	7.89

TAB. D.9 – Résultats obtenus pour les différentes tailles de population

La table D.10 présente les résultats obtenus en utilisant différents probabiltés de mutation.

Programme de test	Mut=1%		Mut=5%		Mut=10%	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	174	4.25	170	4.24	163	4.24
heapsort	114	12.87	116	12.58	101	12.31
fftdp	90	7.62	87	7.5	79	7.41

TAB. D.10 – Résultats obtenus pour les différentes probabilités de mutations

La table D.11 présente les résultats obtenus en utilisant différents pourcentages de sélection.

Programme de test	Selec=0%		Selec=20%		Selec=40%	
	Ind	Coût	Ind	Coût	Ind	Coût
bj	174	4.26	163	4.24	258	4.26
heapsort	116	12.53	99	12.28	229	13.01
fftdp	87	7.39	85	7.4	167	7.6

TAB. D.11 – Résultats obtenus pour les différents pourcentages de sélection

Annexe E

Code source pour le programme de test *arithm*

Dans cette section nous présentons le code source du programme de test *arithm*. La structure de ce programme fait que les heuritiques traditionelles d'allocation de registres se trompent ; par contre AG-Op réussit à trouver une bonne allocation.

```
#include <stdio.h>

main()
{
        FILE *fp ;
        int i,j,n,m ;
        int a0,a1,a2,a3,a4,a5,a6,a7 ;
        int b0,b1,b2,b3,b4,b5,b6,b7 ;

        a0=a1=a2=a3=a4=a5=a6=a7=1 ;
        b0=b1=b2=b3=b4=b5=b6=b7=1 ;

        fp=fopen("nn","r") ;
        fscanf(fp,"%d %d",&n,&m) ; /* n est 100 fois plus grand que m */
                                   /* m=10 n=1000        */
        for (i=0 ;i<n ;i++)
        {
                a0=i*n ;
                a1=a0+n/8 ;
                a2=a1*3-(a2*i) ;
                a3=a3+5*a2 ;
                a4=a2+a3*a1 ;
                a5=a4/2 ;
                a6=a5-a0 ;
                a7=a7+a2*a6 ;
                a0=i*n ;
                a1=a0+n/8 ;
                a2=a1*3-(a2*i) ;
                a3=a3+5*a2 ;
                a4=a2+a3*a1 ;
                a5=a4/2 ;
                a6=a5-a0 ;
                a7=a7+a2*a6 ;
                a0=i*n ;
                a1=a0+n/8 ;
                a2=a1*3-(a2*i) ;
                a3=a3+5*a2 ;
                a4=a2+a3*a1 ;
                a5=a4/2 ;
                a6=a5-a0 ;
                a7=a7+a2*a6 ;
                a0=i*n ;
                a1=a0+n/8 ;
                a2=a1*3-(a2*i) ;
                a3=a3+5*a2 ;
                a4=a2+a3*a1 ;
                a5=a4/2 ;
                a6=a5-a0 ;
                a7=a7+a2*a6 ;
        }

        for (j=1 ;j<m ;j++)
        {

                b0=i*m ;
                b1=b0+n/8 ;
                b2=b1*3-(b2*i) ;
                b3=b3+5*b2 ;
                b4=b2+b3*b1 ;
                b5=b4/2 ;
                b6=b5-b0 ;
                b7=b7+b2*b6 ;
        }

        b7=b1+b2+b3+b4+b5+b6+b7-(a1+a2+a3+a4+a5+a6+a7+a0)-b0 ;
}
```