Une approche génétique et source à source de l'optimisation de code

Serge Guelton, Sébastien Varrette

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Une approche génétique et source à source de l'optimisation de code

Serge Guelton[1] et Sebastien Varrette[2]

[1] Institut Télécom, Télécom Bretagne, Info/HPCAS, Plouzané, France
[2] CSC research unit, University of Luxembourg, Luxembourg

Résumé

Développement et maintenance de codes numériques performants requièrent généralement des optimisations manuelles, au détriment de la maintenabilité. Les compilateurs source à source offrent une solution intéressante à ce problème en séparant le code métier des optimisations. Reste que la recherche des bonnes transformations parmi l'ensemble disponible, la sélection de leur ordre et point d'application exigent l'exploration d'un domaine trop vaste pour espérer obtenir un résultat optimal à partir de modifications manuelles.
Cet article propose un environnement adaptatif qui automatise l'exploration des combinaisons de transformations possibles en couplant un compilateur source à source et un algorithme génétique. Une validation expérimentale donnant des résultats particulièrement encourageants est proposée en fin d'article.

1. Introduction

Tout programmeur d'application scientifique est confronté à un problème crucial : garder un code clair et maintenable tout en garantissant une exécution efficace sur les machines cibles. Deux aspects bien souvent antagonistes ! Un code propre évite la redondance, sépare les concepts par unités de compilation et reste indépendant de l'architecture cible ; au contraire la recherche de performance nécessite des transformations complexes qui obscurcissent le code source (*déroulement* de boucle, *expansion* de procédure, *instructions spécifiques* ...) et dépendent de la cible. Idéalement, cette phase d'optimisation est entièrement déléguée à un compilateur.
Le paramétrage des transformations et leur ordre d'application reste néanmoins un problème complexe non résolu par les compilateurs actuels, au profit d'un ordre d'application statique et de décisions basées sur des heuristiques donnant de bons résultats dans la plupart des cas. La solution consiste alors à tester manuellement un ensemble de transformations que l'on estime rentable. Commence alors un travail d'optimisation manuel qui conduira à un code généralement plus performant, rarement optimal et souvent illisible. On peut utiliser un compilateur source à source afin de séparer cette phase d'optimisation du code initial : on fournit un ensemble de fichiers sources et une série de transformations à appliquer en différents points, et un nouvel ensemble de sources est généré. On peut ensuite le traiter de façon standard. L'exploration manuelle de la totalité de l'espace des transformations étant impossible, on procède généralement à tâtons, ou par expérience.
C'est dans ce contexte que cet article propose une approche adaptative et automatique de cette phase exploratoire en vue de générer un code optimisé pour une cible donnée. Nous commencerons par fournir des pointeurs sur les transformations de code les plus courantes, puis nous vérifierons les limitations de l'approche des compilateurs actuels, justifiant ainsi une approche plus fine (§3). L'une des contributions de cet article est de détailler au §4 un algorithme génétique qui étend une approche antérieure et permet d'optimiser les performances de l'exécution d'un code compilé par le biais de manipulations du code source. La section 5 revient sur l'implémentation de cette approche tandis que le §6 décrit les diverses expériences menées pour

la valider. Cette étude se conclura au §7 avec une ouverture sur les perspectives offertes par cet article.

2. Contexte & Motivations

Depuis l'avènement des langages de programmation de haut niveau, les recherches dans le domaine de la compilation n'ont cessé de trouver des optimisations améliorant les performances du code compilé. De nombreuses études ont ainsi permis d'identifier un grand nombre des transformations applicables. L'ensemble de ces opérations, leurs effets ainsi que leurs contextes d'application sont notamment résumés dans [2]. Déterminer les séquences de transformations optimales qui minimisent le temps d'exécution pour un programme donné est un problème NP-complet [11]. De fait, les compilateurs actuels disposent d'un ordre d'application statique : les décisions sont basées sur des heuristiques qui donnent de bons résultats dans la plupart des cas, mais sans garantie d'optimalité. À ce niveau, le fonctionnement et les choix du compilateur GNU `gcc` sont bien documentés [12][13].

Améliorer les performances des compilateurs nécessite donc soit des connaissances ingénieurs pointues pour manuellement opérer les restructurations de code amenant de meilleurs résultats, soit le développement d'outils d'analyse automatique. Dans cette optique, plusieurs approches s'appuyant sur des algorithmes évolutionnistes ont vu le jour. Les algorithmes évolutionnistes (*Evolutionary Algorithm* – EA en anglais) sont des heuristiques basées sur la théorie de l'évolution de Darwin.Ils définissent une approche dynamique et adaptative permettant d'explorer une *population* de solutions (et non une seule). Une solutions possible *i.e.* un membre de la population est appelé *individu*. Chaque itération d'un EA implique une sélection compétitive qui exclut progressivement les "mauvaises" solutions à travers l'*évaluation* d'une valeur de *fitness* qui indique la qualité d'un individu en tant que solution du problème. Par ailleurs, un ensemble d'opérateur génétiques (mutation et recombinaison *i.e.* cross-over typiquement) sont appliqués à chaque génération pour créer de nouveaux individus qui viendront remplacer toute ou partie de la population. Il existe plusieurs modèles d'algorithmes évolutionnistes, les plus populaires étant les algorithmes génétiques (AG) [4].

Dans le domaine de la compilation, on retrouve les approches génétiques dans l'optimisation de la taille des binaires générés [3] ou dans la recherche automatique des meilleurs options de compilation. Ce dernier cas de figure est traité par l'environnement Acovea [10] pour `gcc` ou Cole [6] dans un cadre plus générique. Sur l'optimisation des séquences de transformation à appliquer à un code donné[8], une approche évolutionniste est proposée dans [9, 5]. Nous aurons l'occasion d'y revenir au §4 et d'expliquer la contribution de cet article dans ce domaine.

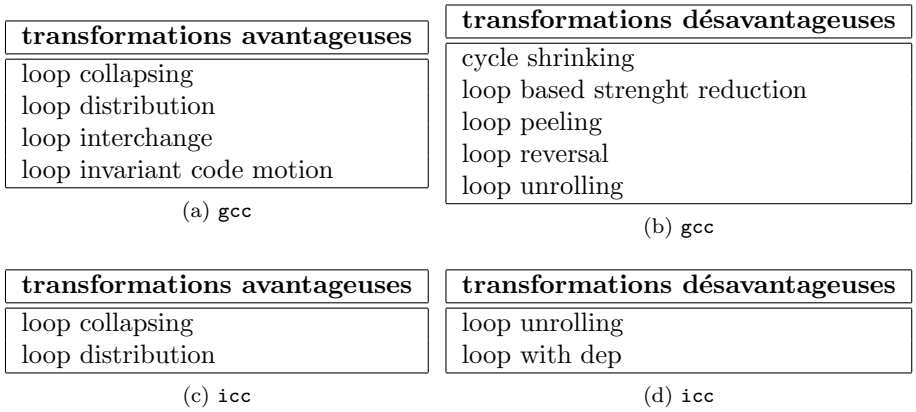
3. Limitations de l'approche statique

3.1. Comparaison des approches statiques & dynamiques

Pour montrer les limitations des compilateurs actuels et l'intérêt d'utiliser des transformations de code en amont du compilateur, nous avons chercher à vectoriser (génération d'instructions vectorielles) des cas tests d'après ce protocole :

1. sélection d'un ensemble représentatif de transformations connues ;
2. pour chacune d'elles, sélection d'un cas où elle s'applique avec profit ;
3. compilation du source non transformé et observation de la vectorisation ;
4. compilation du source transformé et observation de la vectorisation.

Les compilateurs `Intel icc` version 11.0 et `GNU gcc` version 4.3.2 ont été examinés. Le lecteur pourra se référer à `https://freia.enstb.org/resources-related-projects` pour obtenir les sources de l'expérience.



(a) `gcc` (b) `gcc`

(c) `icc` (d) `icc`

FIG. 1: Effets des transformations sur la vectorisation des codes.

Les résultats (vectorisation ou non) sur une quinzaine de transformations sont donnés dans le tableau 1 où l'on remarque que :
– le compilateur `Intel` est plus avancé que celui de `GNU` concernant la vectorisation ;
– les transformations appliquées judicieusement (ici sur décision du développeur) peuvent permettre au compilateur de prendre de meilleures décisions ;
– certaines transformations pénalisent le compilateur.
Plus exactement, les transformations liées à des déroulements de boucle rendent la vectorisation plus difficile tandis que d'autres transformations de boucles facilitent l'analyse des dépendances. Enfin, certaines transformations sont visiblement déjà appliquées par les compilateurs et n'ont pas d'effet visible dans notre cas.

3.2. Complexité de l'approche dynamique

Pour illustrer la difficulté du choix de transformations de code, et donc de l'automatisation de ce choix, nous utilisons un code factice qui nécessite plusieurs transformations pour être vectorisable par `gcc`.

```
void foo(double A[SIZE], double B[SIZE][SIZE]) {
   int i,j;
l0: for(i=0;i<SIZE;i++)
l1:  for(j=0;j<SIZE;j++)
       A[i] = B[j][i] + get(A,i);
}
```

Les labels `l0,l1` ont été ajoutés pour identifier les boucles par la suite. La fonction `get` est définie dans un source à part et a un corps très simple :

```
double get(double f[SIZE],int i) { return f[i]; }
```

L'optimisation de **foo** nécessite un *inlining* inter-procédural, une détection de réduction, de la scalarisation et de l'échange de boucles et est donc relativement complexe : la lisibilité du code y est privilégiée à l'efficacité.
Si l'utilisateur fournit un ensemble T de transformations avec leur jeu de paramètres, et sous l'hypothèse que l'ordre d'application de ces transformations importe, l'ordre de grandeur du coût de parcours de l'ensemble des solutions possibles est de $|\mathsf{Search\_Domain}| \simeq |\mathsf{T}|!$. Une première amélioration serait de ne garder à chaque itération que les K meilleurs éléments de l'itération précédente, conduisant alors à $|\mathsf{Search\_Domain}| \simeq (\mathsf{K}) * |\mathsf{T}|^2$ Cette approche permet de limiter l'explosion combinatoire, mais en procédant ainsi on suit des solutions locales qui pourraient nous écarter des solutions globales. Pour résoudre ce problème, on propose d'utiliser un algorithme génétique.

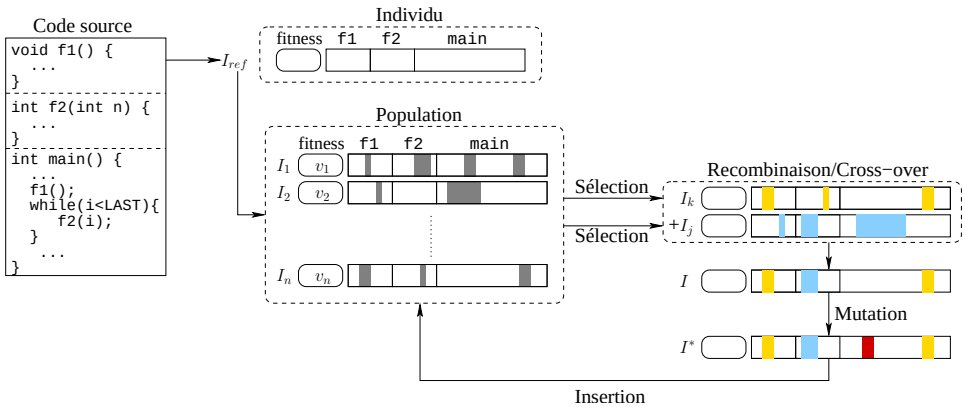


FIG. 2: Illustration de l'AG stationnaire proposé pour l'optimisation dynamique d'un code source.

4. Approche dynamique : Optimisation des performances par algorithme génétique

On s'intéresse dans cet article aux algorithmes génétiques (AG) non-structurés dits stationnaires[1] [1] dont le pseudo code est fournit dans l'algorithme 1.

Algorithme 1 : Pseudo-code d'un algorithme génétique stationnaire.

Entrées : p_c : probabilité d'appliquer le cross-over
p_m : probablité d'appliquer la mutation
Sortie : meilleur individu issu du processus génétique
`pop` ← `Generate_Initial_Population()` ;
`Evaluate(Pop)` ; *//Évaluation de la valeur de fitness de chaque individu*
while *!* `StopCondition()` **do**
  `parents` ← `SelectionFrom(pop)` ; *//Typiquement par tournoi binaire*
  `offspring` ← `Recombine(parents,` p_c`)` ; *//Cross-over*
  `offspring` ← `Mutate(offspring,` p_m`)` ; *//Mutation*
  `Evaluate(offspring)` ;
  //Remplacement des pires individus de `pop` *par ceux de* `offspring`
  `InsertionInto(pop, offspring)` ;
end
return `BestIndividualIn(pop)` ;

Dans le cadre de l'optimisation de codes et pour pallier aux limites de l'approche statique évoquées au §3, une alternative dynamique proposée dans la littérature consiste à utiliser un algorithme génétique [11, 9]. Dans [11], l'auteur cherche à optimiser le temps d'exécution d'un code parallèle Fortran en trouvant l'ordre d'application de cinq transformations de restructuration. Cette approche est reprise dans [9] pour proposer un environnement interactif de compilation et d'amélioration de code appelé VISTA. Ce système offre un *profiler* d'exécution, restreint au nombre d'instructions opérées, et surtout la recherche des séquence de transformations globales optimisant le code exécuté. Chacune d'entre elles correspond à un gène pour les individus de l'algorithme génétique qui ont tous une taille fixe (14 dans [9]). Les générations successives permettent ainsi de trouver la séquence optimale de transformations à appliquer.
Parmi les problèmes ouverts qui concluaient l'étude [9], les auteurs s'interrogeaient sur la possibilité d'optimiser directement les blocs qui composent un programme. C'est précisément l'une des contributions de cet article. En effet, notre approche se concentre sur un code source et cherche à optimiser les modules *i.e.* les fonctions qui

[1] *steady-state Genetic Algorithm* (ssGA) en anglais.

le composent au lieu de rechercher une séquence valide de transformations globales à appliquer. Un ensemble beaucoup plus hétérogène de transformations est ainsi possible. L'AG stationnaire utilisé pour ces travaux est illustré dans la figure 2 et suit l'algorithme 1. Il est spécifié par ses éléments de base :

Individu : un individu I est un code source composé de *modules*[2], correspondant à la définition d'une fonction identifiée par son prototype. Dans la suite, nous faisons l'hypothèse **H1** que l'ensemble des modules est défini statiquement pour l'ensemble des individus des populations considérées. Si cela n'empêche pas les modifications internes à chaque module[3], cette hypothèse permet de garantir la cohérence des opérations de recombinaison détaillées par la suite.
Néanmoins une telle modélisation des individus et donc d'un code source va classiquement au-delà des modules. Nous considérons donc également des sous-modules définis comme des blocs d'instructions (une boucle `for` ou `while`, un bloc `if...else` etc.). Enfin, la granularité la plus fine est obtenue au niveau des instructions de haut niveau qui composent chaque sous-module. Cette classification hiérarchique sera utilisée dans la suite de cet article.

Population : un ensemble de $\mathtt{n}$ individus.

Initialisation de la population : processus de génération définissant les $\mathtt{n}$ premiers individus qui serviront à alimenter la première génération de l'algorithme. Dans le cadre de cet article, le source à optimiser est analysé pour en extraire les modules qui le composent. Cette décomposition fournit un individu référence $\mathrm{I}_{\mathtt{ref}}$ qui est ensuite dérivé en $\mathtt{n}$ individus par l'application de mutations aléatoires sur $\mathrm{I}_{\mathtt{ref}}$ de sorte que l'hypothèse (H1) est respectée.

Évaluation des individus / *fitness* : caractérisation de la qualité d'un individu. Dans le cadre de l'optimisation de code étudiée ici, on s'intéresse au temps d'exécution du code compilé sur la machine cible.

Sélection : sélection des individus qui serviront de parents aux opérateurs génétiques (recombinaison et mutation). Plusieurs approches sont possibles (roulette, tournoi etc). Dans cet article, un tournoi binaire[4] est utilisé pour la sélection d'un parent. Il s'agit de tirer au hasard deux individus dans la population et de choisir le gagnant d'un tournoi basé sur la valeur de fitness.

Recombinaison / Cross-over : opérateur génétique assurant la création d'un individu I "nouveau-né" à partir des parents I_1 et I_2. Bien que l'implémentation de chaque module puisse différer, les types de modules sont communs aux deux parents par l'hypothèse (H1). L'idée consiste alors de composer I de sorte que pour chaque module, un tirage aléatoire et uniforme détermine celui de I_1 et I_2 qui sera choisi.

Mutation : l'opérateur génétique le plus important puisqu'il consiste à générer un nouvel individu I^* dérivant de I par l'application des transformations de code mentionnées §2. De fait, ces modifications s'effectuent à différentes granularités (modules ou sous-modules, instruction). À chaque niveau et sous réserve que cela soit possible, une transformation est choisie aléatoirement et est appliquée à un élément aléatoire de l'individu.

5. Implémentation de l'approche dynamique avec compilateur source à source

Afin de valider l'approche génétique pour l'optimisation hors ligne d'un programme, il est nécessaire de pouvoir manipuler l'ensemble des codes sources pour en dériver

[2] les chromosomes dans la terminologie des algorithmes évolutionnistes.

[3] L'objectif de l'AG est évidemment de faire évoluer le code source et d'analyser l'impact des modifications d'implémentation de chaque module.

[4] *Binary tournament selection* en anglais.

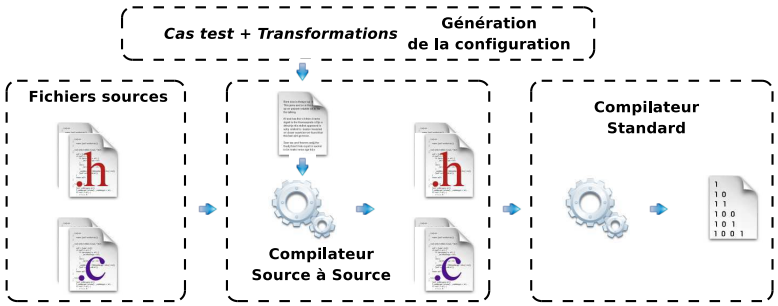


FIG. 3: Chaîne de compilation étendue.

de nouveaux sources suite à des séries de transformation idempotentes. Le compilateur source à source `PIPS` [7], développé aux mines de Paris, permet de créer plusieurs générations de sources différents, en modifiant simplement un fichier de configuration détaillant la nature, l'ordre et la localisation de ces transformations. La figure 5 détaille le mode opératoire choisi : l'algorithme génétique s'appuie sur les données (transformations et cas tests) fournies par l'utilisateur pour créer différentes configurations et finalement en choisir une qui peut alors alimenter le compilateur source à source.

Chaque individu est représenté par un ensemble de fichiers sources et par un ensemble de transformations qui lui sont appliquées. À chaque génération, on fait évoluer les individus et on en crée de nouveaux en manipulant leur ensemble de transformations, et les appels au compilateur `PIPS` se chargent des transformations de code permettant ensuite une évaluation des individus pour l'étape de sélection.

Une originalité de l'approche source à source est que le résultat de l'exploration n'est pas un binaire, mais un ensemble de transformations sous forme de fichier de configuration. L'investissement en temps d'exploration est donc vite rentabilisé puisque l'on peut réutiliser ce jeu de transformation pour les compilations futures, ou comme base pour un nouveau processus d'exploration. La granularité des transformations appliquées – le bloc d'instruction – permet de pousser la finesse de l'analyse plus loin que les approches classiques. Un port `python` de `PIPS` a été réalisé, ce qui a permis un prototypage rapide de l'algorithme et plusieurs expériences, détaillées dans la section suivante, ont été conduites.

6. Expériences & Validation

Pour notre implémentation, nous avons repris l'exemple de la section 3.2. La métrique utilisée est le nombres de cycles CPU moyen pour une exécution sur un jeu de données aléatoires. On notera les limites de cette métrique : une optimisation bonne sur des données irrégulières ne le sera pas toujours sur des données régulières.

Pour valider le prototype, on utilise les transformations caractéristiques suivantes : déroulement de boucle, expansion de procédure, découpage d'indice et échange de boucle. Cette liste gagnerait à être enrichie par la scalarisation et la distribution de boucles, opérations non disponibles lors de la réalisation des expériences. Les expériences sont conduites sur processeur `Intel Pentium 4` cadencé à 3 GHz. Le score de fitness est obtenu par une moyenne sur 100 exécutions et le nombre de cycle est observé sur le *Time Stamp Counter*. §6.1, §6.2 et §6.3 détaillent chaque série d'expérience et un résumé des résultats est fournit au §6.4

6.1. Parcours complet

Cette première expérience nécessite la génération de l'ensemble des combinaisons de transformations possibles à partir d'un ensemble paramétré fourni par l'utilisateur. Chaque série de transformation est appliquée au code source original, qui est ensuite

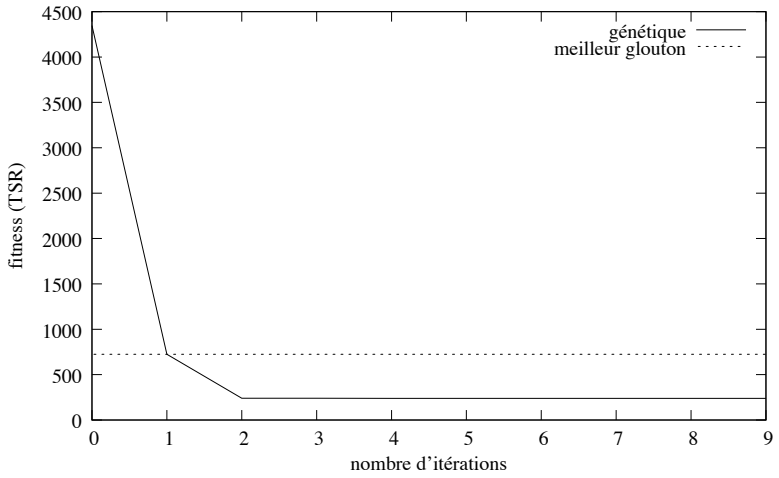


FIG. 4: Évolutions de l'approche génétique.

compilé et évalué. Au terme de l'exécution, on trie les différents individus pour exhiber le meilleur élément, et la série de transformations associées.

6.2. Parcours glouton

Cette expérience reprend le même jeu de données que l'expérience précédente, effectue le même nombre d'itérations mais ne conserve à chaque itération que les K meilleurs éléments de l'itération précédente. Cette méthode ne conduit pas à un résultat optimal : le déroulement de boucle s'avère bénéfique dès le début et est donc sélectionné au détriment des autres transformations de boucle qui ne sont plus applicables par la suite. On reste donc sur un résultat optimal localement dont on ne s'échappe pas, ce qui justifie l'approche génétique.

6.3. Parcours par algorithme génétique

Dans cette expérience, on reproduit l'expérience précédente en se basant sur le mécanisme décrit au §4. Cette approche permet d'arriver au même résultat que 6.1 en un temps moindre, modulo le paramétrage d'une borne sur le nombre de générations

6.4. Résultats

Le tableau suivant résume les résultats obtenus :

Parcours	**Complet**	**Glouton**	**Génétique**
Temps de parcours (s)	160	55	$14 \leq \ldots \leq 59$
Accélération / `-O3`	6.9	2.2	$6.8 \geq \ldots \geq 2.3$

L'algorithme glouton sélectionne les 3 meilleurs éléments à chaque passe et effectue autant de passes que de transformations. Ses performances sont honorables et illustrent bien qu'en limitant naïvement l'explosion combinatoire, on obtient des résultats en un temps raisonnable, mais potentiellement éloigné de l'optimum. L'algorithme génétique effectue 10 passes. C'est un choix délibéré afin de limiter le temps de calcul néanmoins cette décision est entièrement paramétrable par l'utilisateur. Un intervalle est donné à cause de la nature aléatoire de l'algorithme. La figure 4 montre l'évolution du meilleur individu trouvé génétiquement à chaque passe par rapport au meilleur individu final de l'algorithme glouton.

7. Conclusion & Perspectives

Dans cet article, nous avons montré comment la combinaison d'un algorithme génétique (AG) et d'un compilateur source à source permettait d'explorer avantageusement et automatiquement l'espace des transformations de code possibles : l'approche

source à source offre une bonne flexibilité tandis que l'algorithme génétique limite l'explosion combinatoire. En ce sens, ce travail étend plusieurs approches antérieures proposant l'exploitation d'AG pour l'optimisation de codes en offrant notamment un contrôle plus fin des transformations applicables.

Un premier prototype a permis de valider cette approche sur un cas d'école. Ce test montre déjà des facteurs d'accélérations intéressants par rapport à une compilation classique. Il s'agit maintenant de valider le prototype sur des cas réels plus conséquents, ce qui nous amènera à considérer l'augmentation du nombre de transformations applicables et la parallélisation de l'exploration.

Ces travaux préliminaires peuvent s'étendre à d'autres domaines connexes. On peut notamment espérer obtenir des résultats très intéressants en appliquant les idées développées dans l'article aux cas plus spécifiques que sont les noyaux de calculs des GPGPU, les codes destinés à être traduit en `VHDL` ou vers des assembleurs spécifiques comme celui du `Terapix` de Thalès.

Remerciements

Ces travaux sont financés par le projet ANR AF FREIA et l'Université du Luxembourg. Nous remercions tout particulièrement Pascal Bouvry, Ronan Keryell et François Irigoin pour leur relecture et leurs conseils.

Bibliographie

1. E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*, volume 42 of *Operations Research/Computer Science Interfaces*. Springer-Verlag Heidelberg, 2008.
2. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 1994.
3. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.
4. D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, 1989.
5. Kenneth Hoste and Lieven Ee khout. Cole : compiler optimization level exploration. In *CGO '08 : Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, New York, NY, USA, 2008. ACM.
6. Kenneth Hoste and Lieven Eeckhout. Cole : compiler optimization level exploration. In *CGO*, pages 165–174, 2008.
7. François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : an overview of the pips project. In *ICS*, pages 244–251, 1991.
8. Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. A feasibility study in iterative compilation. In *ISHPC*, pages 121–132, 1999.
9. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. B. Whalley, J. W. Davidson, M. W. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *LCTES*, pages 12–23, 2003.
10. S.R. Ladd. Acovea : Using Natural Selection to Investigate Software Complexities. [Online] see `www.coyotegulch.com/products/acovea/`, 2007.
11. Andy Nisbet. GAPS : A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In *HPCN Europe*, pages 987–989, 1998.
12. R.M. Stallman et al. *Using GCC : The GNU Compiler Collection Reference Manual*. FSF, 2005.
13. Wikibooks, editor. *GNU C Compiler Internals*. `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`, 2006-2009.