Élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées

Youcef Bouchebaba

CRI, ENSMP
77305 Fontainebleau Cedex France
e-mail: {boucheba@cri.ensmp.fr }

Résumé
Les optimisations de code à base de transformations unimodulaires ne peuvent s'appliquer qu'à des nids de boucles parfaitement imbriquées. Les codes réels comportent souvent des nids de boucles mal imbriquées. Enfin les techniques de pipeline logiciel ne peuvent s'appliquer efficacement qu'à des corps de boucles sans tests. Pour permettre l'application de transformations unimodulaires et de pipeline logiciel à des codes réels, cet article propose une technique d'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées et non paramétriques à deux dimensions. On peut diviser ce travail en deux parties : dans la première, on s'intéresse à la transformation d'un nid de boucles mal imbriquées en un nid parfaitement imbriqué en ajoutant des gardes ad'hoc sur les instructions que l'on transfère dans la boucle la plus interne. Nous donnons la condition nécessaire et suffisante à l'application de cette transformation. Si cette condition n'est pas vérifiée, ou bien la deuxième boucle n'est jamais exécutée et dans ce cas on l'élimine entièrement, ou bien, on décompose le nid de boucles en une séquence de deux nids, tel que: dans un nid on fait entrer toutes les instructions dans la boucle la plus interne et dans l'autre on élimine la boucle interne. Dans la deuxième partie, on s'intéresse au problème inverse, l'élimination des gardes dans le code obtenu après l'application d'une transformation unimodulaire. On décompose ce dernier en une séquence de nids de boucles de manière à toujours avoir une seule contrainte inférieure et une seule contrainte supérieure dans la boucle la plus interne et pouvoir sortir les gardes et les instructions gardées de la boucle interne, voir à éliminer complètement les gardes.

Mots-clés : Transformation de programmes, Parallélisation automatique, Élimination des gardes.

---

1. Introduction

Plusieurs travaux [11, 2, 6, 9] ont utilisé les transformations unimodulaires pour augmenter la localité des données, augmenter le degré du parallélisme, ...etc. Cependant la plupart sont destinés aux nids de boucles parfaitement imbriquées. Dans la pratique, les programmes contiennent surtout des nids de boucles mal imbriquées, et leurs traitements pourraient permettre l'obtention de gain considérable. C'est pour ces raisons qu'on s'est intéressé au problème de l'élimination des gardes au sein de nids de boucles.

L'application d'une transformation unimodulaire directement sur un nid de boucles mal imbriquées n'est pas possible. On est donc ramené à rendre ce dernier parfaitement imbriqué en ajoutant des gardes sur les instructions qu'on fait rentrer dans la boucle la plus interne avant de pouvoir lui appliquer une transformation unimodulaire. Wolf[8] et Abu-Sufah [1] ont mentionné l'utilisation de cette technique mais n'ont pas envisagé tous les cas possibles et ils ont signalé que les gardes ajoutées consomment beaucoup de temps supplémentaires dans le code transformé. Wolfe [10] a par contre proposé d'appliquer d'abord une distribution de boucles, mais celle-ci n'est pas toujours possible et crée des problèmes de performance.

Dans cet article, on s'est intéressé à la condition nécessaire et suffisante pour que la transformation d'un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées soit correcte. On a aussi traité le cas où cette condition n'est pas vérifiée. En deuxième lieu, on donne un algorithme qui permet d'éliminer les gardes dans le code transformé en découpant ce dernier en une séquence de nids de boucles. Chamski [4] a aussi proposé d'utiliser le découpage de boucles pour éliminer les calculs de

```
Begin
    do ( i = 0, i = 4 )
        S = 0
        do (j = 0 , j = 6 )
            S = S + A(i,j) * X(j)
        enddo
        Y(i) = S
    enddo
End
```
(a)

```
Begin
    do ( i = 0, i = 4 )
        do (j = 0 , j = 6 )
            if (j=0) then S = 0
            S = S + A(i,j) * X(j)
            if (j=6) then Y(i) = S
        enddo
    enddo
End
```
(b)

FIG. 1 – *Produit matrice/vecteur: passage du nid mal imbriqué au nid parfaitement imbriqué.*

maxima et de minima dans les bornes de boucles. Par contre Lengauer [5] et Wetzel [7] se sont intéressés aux techniques de fusion et de distribution de boucles pour la génération de code.

Cet article est organisé de la manière suivante. On commence par décrire le problème avec un simple exemple, le produit matrice/vecteur. Puis on donne la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées. Dans la troisième partie on décrit l'algorithme qui permet d'éliminer les gardes dans le nid transformé. Enfin, on termine par une conclusion récapitulant les résultats et présentant les travaux futurs.

2. Exemple introductif

Afin de mieux comprendre le problème traité, on donne un simple exemple qui est le produit matrice/vecteur, dont l'algorithme est présenté en figure 1:(a).

Pour appliquer une transformation unimodulaire sur l'algorithme (a) on doit tout d'abord le rendre parfaitement imbriqué, c'est-à-dire transférer les instructions `S=0` et `Y(i)=S` à l'intérieur de la boucle `j`. Ceci est possible car la condition nécessaire et suffisante est vérifiée (`Section3`): pour toute valeur de i, il existe au moins une itération j . Le nid de boucles parfaitement imbriquées correspondant à l'algorithme (a) est donné en figure 1: b.

Une transformation unimodulaire est une transformation affine. Elle transforme donc une droite qui est à la frontière d'un polytope initial en une droite qui est à la frontière du polytope transformé.

Par exemple, la transformation unimodulaire de l'algorithme (b) par $T = \begin{pmatrix} 6 & 1 \\ 1 & 0 \end{pmatrix}$, nous donne l'algorithme de la figure 2 : c.

```
Begin
    do ( i' = 0, i' = 30 )
        do ( (-j' ≤ 0) ∧ (i' - 6 * j' ≤ 6), (j' ≤ 4) ∧ (-i' + 6 * j' ≤ 0))
            if (-i' + 6 * j' = 0) then S = 0
            S = S + A(j',i' - 6 * j') * X(i' - 6 * j')
            if (i' - 6 * j' = 6) then Y(j') = S
        enddo
    enddo
End
```
(c)

FIG. 2 – *Algorithme transformé*

On voit que l'image de la droite définie par $j = 0$ qui est à la frontière du polytope définissant le domaine d'itérations de l'algorithme (b) est la droite définie par $-i' + 6 * j' = 0$ qui est à la frontière du polytope du domaine de l'algorithme (c).

Lorsqu'on transforme un nid mal imbriqué en un nid parfaitement imbriqué, on ajoute des gardes sur les instructions qu'on fait rentrer dans la boucle la plus interne. Ces gardes sont définies par des droites qui sont aux frontières du polytope du domaine initial. Par conséquent, dans le nid transformé, les images de ces gardes corespondent à des droites qui sont toujours aux frontières du polytope correspondant.

La deuxième étape consiste à éliminer les gardes dans le code transformé. Pour pouvoir réaliser cela, on doit décomposer le nid de boucles (c) en une séquence de nids de boucles de manière à avoir dans chaque séquence une seule contrainte inférieure et une seule contrainte supérieure dans la boucle interne j'. Cette décomposition se fait en utilisant les sommets générateurs du polytope transformé et revient à transformer l'unique nid (c) en une séquence présenté en figure 3.

```
Begin
    do ( i' = 0, i' = 5 )
        do ( j' = 0 , j' = i'/6 - 1/6)
            S = S + A(j',i' - 6 * j') * X(i' - 6 * j')
        enddo
        if (entier i'/6) then { S = 0 ; S = S + A(j',i' - 6 * j') * X(i' - 6 * j') }
    enddo
    do ( i' = 6, i' = 24 )
        if (entier (i'-6)/6) then { S = S + A(j',i' - 6 * j') * X(i' - 6 * j') ; Y(j') = S }
        do ( j' = i/6 , j' = i'/6 - 1/6)
            S = S + A(j',i' - 6 * j') * X(i' - 6 * j')
        enddo
        if (entier i'/6) then { S = 0 ; S = S + A(j',i' - 6 * j') * X(i' - 6 * j') }
    enddo
    do ( i' = 25, i' = 30 )
        if (entier (i'-6)/6) then { S = S + A(j',i' - 6 * j') * X(i' - 6 * j') ; Y(j') = S }
        do ( j' = i/6 , j' = 4)
            S = S + A(j',i' - 6 * j') * X(i' - 6 * j')
        enddo
    enddo
End
```

FIG. 3 – *Code transformé et décomposé*

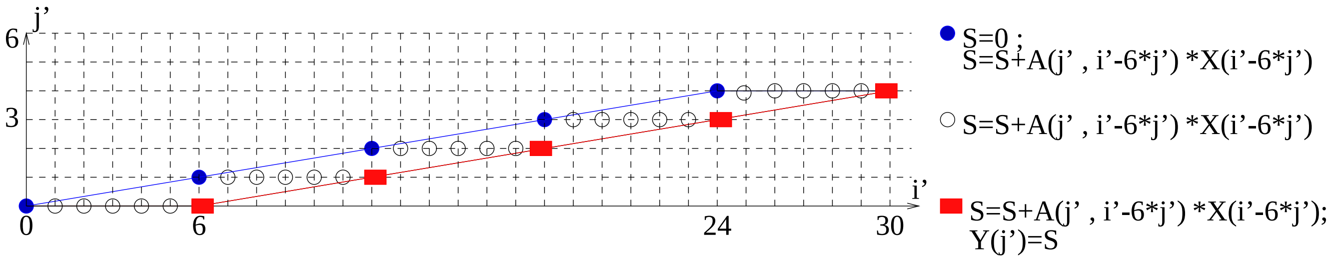

FIG. 4 – *Domaine d'itérations de l'algorithme (c)*

```
Begin
    do (i1 = inf, i1 = sup)
       S1
       do (i2 = a1 * i1 + b1 , i2 = a2 * i1 + b2 )
          S2
       enddo
       S'1
    enddo
End
       (d)
```

```
Begin
    do (i1 = inf, i1 = sup)
       do (i2 = a1 * i1 + b1 , i2 = a2 * i1 + b2 )
          If i2 = a1 * i1 + b1 then S1
          S2
          If i2 = a2 * i1 + b2 then S'1
       enddo
    enddo
End
       (e)
```

FIG. 5 – *Programmes traités*

L'étude du code transformé en figure 2 et du domaine d'itérations en figure 4 montre que l'on peut extraire l'affectation `S=0`, gardée par la condition $-i' + 6 * j' = 0$, après la boucle j', car elle s'exécute uniquement au cours de la dernière itération de j'. En effet, la droite définie par $-i' + 6 * j' = 0$ est la frontière haute du domaine d'itérations. L'affectation `S=0` s'exécutant avant l'affectation :

S=S+A(j',$i' - 6 * j'$) * X($i' - 6 * j'$)

on extrait ces deux instructions de la boucle j' et on les garde par la condition i' `mod 6 =0` , car elles ne s'exécutent que pour des valeurs de i' qui sont multiples de 6.

Par contre l'affectation $Y(j') = S$, gardée par $i' - 6 * j' = 6$, sera extraite avant la boucle j', car elle s'exécute uniquement au cours de la première itération de j'. En effet la droite définie par $i' - 6 * j' = 6$ est la frontière inférieure du domaine d'itération. l'affectation $Y(j') = S$ s'exécutant après l'affectation :

S=S+A(j',$i' - 6 * j'$) * X($i' - 6 * j'$))

on extrait ces deux instructions de la boucle j' et on les gardes par la condition $(i' - 6)$ `mod 6=0`, car elles ne s'exécutent que pour des valeurs de i' telles que $(i' - 6)$ est multiple de 6.

3. Transformation en nid de boucles parfaitement imbriquées

Les nids de boucles qu'on traite dans cet article sont de la forme (d), et on veut les mettre sous la forme (e). Ceci est possible si et seulement si ces deux conditions sont vérifiées:

- $(a_2 * sup + b_2 \geq a_1 * sup + b_1)$
- $(a_2 * inf + b_2 \geq a_1 * inf + b_1)$

Si ces conditions ne sont pas vérifiées, ou bien la boucle interne est vide, ou bien il est possible de les vérifier en décomposant l'intervalle de i_1 en deux sous-intervalles bornés par l'intersection des bornes inférieures et supérieures de la boucle sur i_2.

4. Elimination des gardes dans le nid de boucles transformé

Le nid de boucles (e) aura la forme (f) après la transformation unimodulaire (voir figure 6).

Les droites définies par $h_{m,1}*i'_1+h_{m,2}*i'_2 = b_m$ et $h'_{m,1}*i'_1+h'_{m,2}*i'_2 = b'_m$ sont toujours aux frontières du polytope du nid (f) mais elles peuvent être soit sur la frontière inférieure, soit sur la frontière supérieure, soit perpendiculaires à l'axe i'_1.

La borne inférieure de la boucle sur i'_2 peut être composée d'une ou deux contraintes, de même pour la contrainte supérieure. Le nombre de structures (nids de boucles) possibles que peut prendre le programme (f) après l'élimination des gardes est très important, car le programme résultant dépend de la position des droites définies par $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ et $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ et du nombre de contraintes inférieures et supérieures de la boucle i'_2.

```
Begin
    do (−i'_1 ≤ −b_0 , i'_1 ≤ b'_0)
       do (∧_{j=1}^{n}(a_j * i'_1 − d_j * i'_2 ≤ b_j), ∧_{j=1}^{n'}(a'_j * i'_1 + d'_j * i'_2 ≤ b'_j ))
          If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m alors S_1
          S_2                         (d)
          If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m). alors S'_1
       enddo
    enddo
End
```

(f)

FIG. 6 – *Programme après la transformation unimodulaire*

Nous donnons ici l'idée générale de notre algorithme, qui permet d'éliminer les gardes dans le nid transformé, en décomposant ce dernier en une séquence de nids de boucles. Cette décomposition se fait en utilisant les projections des sommets générateurs du polytope transformé sur l'axe i'_1. Cet algorithme est détaillé, justifié et prouvé correct et optimal dans un rapport interne [3].

Notre algorithme fonctionne suivant le schéma général suivant :

- 1) Découper l'intervalle $[b_0,b'_0]$ en sous intervalles $[ig_1,ig_1] \cup]ig_1,ig_2[\cup [ig_2,ig_2] \cup ......\cup [ig_{i-1},ig_{i-1}] \cup\ ]ig_{i-1},ig_{i+1}[\cup [ig_{i+1},ig_{i+1}] \cup ....\cup\ ]ig_{n-1},ig_n[\cup [ig_n,ig_n]$. ig_i est l'abscisse du sommet générateur i. Si deux sommets générateurs ont la même abscisse, on ne prend qu'une seule valeur. Chaque intervalle correspond à un nid de boucles.
- 2) Pour chaque intervalle:
  - a) Trouver la contrainte inférieure et la contrainte supérieure de la boucle i'_2.
  - b) Si la droite définie par $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$
    - i) Correspond à la contrainte inférieure de la boucle i'_2 alors:
      - Si $h_{m,2} = -1$, on transfère " S_1 " avant la boucle i'_2.
      - Si $h_{m,2} \neq -1$, on transfère "$If\ entier(\frac{b_m - h_{m,1} * i'_1}{h_{m,2}})\ then\ S_1$ " avant la boucle i'_2.
    - ii) Est perpendiculaire à l'axe i'_1 ($h_{m,2} = 0$, ceci est possible uniquement pour le premier ou le dernier intervalle), le code " S_1 " reste à l'intérieure de la boucle i'_2.
    - iii) Correspond à la contrainte supérieure de la boucle i'_2 alors:
      - Si $h_{m,2} = 1$, on transfère " $S_1;\ S_2$ " après la boucle i'_2.
      - Si $h_{m,2} \neq 1$, on transfère "$If\ entier(\frac{b_m - h_{m,1} * i'_1}{h_{m,2}})\ then\ S_1;\ S_2$ " après la boucle i'_2.
  - C) Si la droite définie par $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$
    - i) Correspond à la contrainte inférieure de la boucle i'_2 alors:
      - Si $h'_{m,2} = -1$, on transfère " S_2 ; S'_1 " avant la boucle i'_2.
      - Si $h'_{m,2} \neq -1$, on transfère " $If\ entier(\frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}})\ then\ S_2$; S'_1 " avant la boucle i'_2.
    - ii) Est perpendiculaire à l'axe i'_1 ($h'_{m,2} = 0$, ceci est possible uniquement pour le premier ou le dernier intervalle), le code " S'_1 " reste à l'intérieure de la boucle i'_2.
    - iii) Correspond à la contrainte supérieure de la boucle i'_2 alors:
      - Si $h'_{m,2} = 1$, on transfère " S'_1 " après la boucle i'_2.
      - Si $h'_{m,2} \neq 1$, on transfère "$If\ entier(\frac{b_m - h_{m,1} * i'_1}{h_{m,2}})\ then\ S'_1$ " après la boucle i'_2.
- 3) Regrouper le code de certains sommets générateurs avec le code de l'intervalle gauche ou droite qui leur sont adjacents quand ceci est possible.

Dans certains cas, on peut avoir un sommet générateur qu'on traite d'une manière spéciale. Cela revient à sortir avant ou après la boucle i'_2 le code " $S_1;\ S_2;\ S'_1$ ".

5. Conclusion

L'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées n'a été traitée par aucun auteur. Wolf [8] et Abu-Sufah [1] ont donné une idée générale de la première partie de ce problème, à savoir la transformation de nid de boucles mal imbriquées en un nid parfaitement imbriqué. Leurs approches consistent à faire rentrer dans la boucle la plus interne toutes les instructions mal placées en ajoutant des gardes. Mais ils n'ont pas traité tous les cas possibles, c'est-à-dire les cas où on peut faire rentrer les codes dans la boucle la plus interne uniquement pour certaines valeurs de l'indice de la première boucle. De plus, ils ne se sont pas intéressés à la deuxième partie de ce problème, à savoir l'élimination des gardes dans le nid de boucles transformé, ce qui est pourtant la partie la plus importante pour obtenir de bonnes performances.

De notre coté, nous avons traité les deux parties du problème en détail. Pour la première partie, nous avons donné la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid parfaitement imbriqué. Si cette condition n'est pas satisfaite, ou bien (1) la boucle interne est vide et nous éliminons entièrement le corps de la boucle interne, ou (2) nous décomposons le nid de boucles en une séquence de deux nids de boucles. Dans un nid, nous faisons rentrer toutes les instructions dans la boucle la plus interne et dans l'autre nous éliminons le corps de la boucle interne.

Pour la seconde partie du problème, nous avons proposé un algorithme qui permet d'éliminer les gardes, en décomposant le nid transformé en une séquence de nids de boucles. Notre algorithme minimise le nombre de nids de boucles nécessaires à l'élimination de toutes les gardes internes. Il se peut que des gardes subsistent dans le corps de la boucle externe. Elles pourraient aussi être éliminées par un déroulage de cette boucle, qui aurait l'inconvénient d'augmenter la taille du code.

Vu la difficulté de la deuxième partie de ce problème, nous avons limité notre étude aux nids de boucles à deux dimensions, non paramétriques, parce que le nombre de formes que peut prendre l'algorithme décomposé est déjà très élevé et l'expansion de code non négligeable dans les cas les plus défavorables. Dans de prochains travaux, nous espérons pouvoir généraliser cette étude au cas général, paramétrique et en dimension quelconque, mais la taille du code produit risque d'exploser. Il sera sans doute nécessaire de se restreindre aux boucles les plus internes pour lesquelles l'élimination de garde permet d'appliquer un algorithme de pipeline logiciel et produit de toute façon la partie essentielle de l'optimisation puisque ce sont les tests les plus fréquemment exécutés.

Bibliographie

1. W. Abu-Sufah. Improving the performance of virtual memory computers. *PhD thesis, University of Illinois at Urbana-Champaign*, Nov. 1978.
2. U. Banerjee. Loop transformations for restructuring compilers. *Kluwer academic publishers*, 93.
3. Y. Bouchebaba. Élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées. *Rapport interne du CRI : A/312*, 2000.
4. Z. Chamski. Environnement logiciel de programmation d'un accélérateur de calcul parallèle. *PhD thesis, University of Rennes I*, 93.
5. C. Lengauer et al. Code generation in the polytope model. *Pact98*, 98.
6. A. Choudhary M. Kandemir and J. Ramanujam. I/o-conscious tiling for disk-resident data sets. *Euro-par*, 1999.
7. S. Wetzel. Automatic code generation in the polytope model. *PhD thesis, University of Passau*, 93.
8. M. E. Wolf. Improving locality and parallelism in nested loops. *PhD thesis, University of stanford*, 1992.
9. M. E. Wolf and M. S. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE trans. on paral*, 2, 91.
10. M. Wolfe. Optimizing supercompilers for supercomputers. *MIT Press*, 1989.
11. M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley*, 1996.