Linéarisation du code pour processeur MIPS R3000

Etienne Bergeron

22 Janvier 2003

Résumé

Le but de ce document est d'introduire la génération de code vers un processeur MIPS aux étudiants du cours d'architecture des ordinateurs contemporains avancé (ift3380) de l'Université de Montréal. Il défini certains mécanismes de linéarisation de code pour un langage similaire au langage C.

1 Introduction

La génération de code assembleur à partir d'un langage source de plus haut niveau semble une tâche laborieuse pour les programmeurs n'ayant jamais écrit de code assembleur. De nos jours, cette transformation est automatisée et effectuée par des compilateurs. Les compilateurs les plus simples appliquent des règles de production sur le langage source pour obtenir le langage cible. Dans ce document, nous donnons des exemples de règles de production qu'un compilateur pourrait employer pour réaliser sa tâche. Ces règles ne sont pas exhaustives et ne permettent que de compiler un sous-ensemble de programme source.

La première section introduit la syntaxe des règles de production. La deuxième section défini les règles de génération pour les expressions et les structures de contrôle du langage. Par la suite, des mécanismes plus évolués permettant les appels de fonction sont détaillés. Le document contient des exemples de code permettant au lecteur d'observer du code MIPS généré.

2 Syntaxe des règles

Pour décrire les règles de production, nous employons une syntaxe définissant les expressions à compiler et définissant le code résultant de la transformation. L'emploi de règles de production permet d'automatiser le processus de transformation.

La notation employée pour les règles de production est décrite par la formule suivante :

$\mathcal{C}[\![\ C_{source}\ ]\!]_{\rho} \rightarrow \$tn$

$\Rightarrow\ C_{machine}$ où

C_{source}	est le code importé.
$C_{machine}$	est le code généré.
ρ	est l'environnement local. (ensemble associatif variable→mémoire)
$\$tn$	est le registre temporaire cible de l'évaluation,

2.1 Variables locales et globales

ρ est un ensemble associatif de variables vers l'emplacement de celle-ci. $\rho_{\texttt{i}}$ retourne l'emplacement de la variable `i`.

Les règles décrites ci-dessous ne supportent que les variables locales. Toute les variables locales sont conservées sur la pile d'exécution et un sous-ensemble des registres sont utilisés. Le code produit n'est pas un code optimisé. Ce document vise la génération simple de code et non l'optimisation de code.

Les mécanismes permettant d'associer les variables du langage source aux registres de l'architecture ne sont pas décrit dans ce document. Le lecteur doit allouer manuellement les espaces mémoires.

L'allocation des registres temporaires utilise une astuce simple. Le processeur mips possède des registres temporaire (**$t0-$t9**) que nous utilisons comme une pile d'évaluation pour évaluer les expressions. Ainsi, le registre $t0 est le premier registre à être utilisé pour évaluer une expression. Le registre $t1 devient le registre de référence pour la pile d'évaluation des expressions enfants. Le débordement de registre (le manque de registre) n'est pas traité dans ce document.

3 Règles

Une règle de production simplifie une expression du langage source. Pour obtenir du code assembleur à partir du langage source en employant les règles de production, il suffit d'appliquer les règles itérativement tant que des expressions du langages source sont encore simplifiables par les règles de production.

Certaines règles de production sont données ci-dessous et un exemple suit pour montrer le fonctionnement de l'algorithme.

3.1 Arithmétique

3.1.1 Addition

$\mathcal{C}[\![$ <*expr*>$_1$ + <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ $\mathcal{C}[\![$ <*expr*>$_1$ $]\!]_\rho \rightarrow \$tn$
$\mathcal{C}[\![$ <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn{+}1$
add $\$tn$ $\$tn$ $\$tn{+}1$

3.1.2 Soustraction

$\mathcal{C}[\![$ <*expr*>$_1$ − <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ $\mathcal{C}[\![$ <*expr*>$_1$ $]\!]_\rho \rightarrow \$tn$
$\mathcal{C}[\![$ <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn{+}1$
sub $\$tn$ $\$tn$ $\$tn{+}1$

3.1.3 Comparaison

$\mathcal{C}[\![$ <*expr*>$_1$ < <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ $\mathcal{C}[\![$ <*expr*>$_1$ $]\!]_\rho \rightarrow \$tn$
$\mathcal{C}[\![$ <*expr*>$_2$ $]\!]_\rho \rightarrow \$tn{+}1$
slt $\$tn$ $\$tn$ $\$tn{+}1$

3.1.4 Chargement d'une variable

$\mathcal{C}[\![$ **var** $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ **lw** $\$tn$ $\rho_{\mathbf{var}}(\$sp)$

3.1.5 Affectation d'une variable

$\mathcal{C}[\![$ **var** = <*expr*>$_{value}$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ $\mathcal{C}[\![$ <*expr*>$_{value}$ $]\!]_\rho \rightarrow \$tn$
sw $\$tn$ $\rho_{\mathbf{var}}(\$sp)$

3.1.6 Chargement d'un constante

$\mathcal{C}[\![$ **constant** $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$ **add** $\$tn$ $\$zero$ **constant**

3.1.7 Accès à un tableau

$\mathcal{C}[\![$ <*expr*>$_{var}$[<*expr*>$_i$] $]\!]_\rho \rightarrow$ \$tn

$\Rightarrow$ $\mathcal{C}[\![$ <*expr*>$_{var}$ $]\!]_\rho \rightarrow$ \$tn
$\mathcal{C}[\![$ <*expr*>$_i$ $]\!]_\rho \rightarrow$ \$tn+1
all \$tn+1 \$tn+1 **2**
add \$tn \$tn \$tn+1
lw \$tn (\$tn)

3.1.8 Exemple

L'expression **(x+y+z < y−x)** est transformée par les règles de production pour produire le code assembleur pour montrer le fonctionnement des règles de production. Voici les différentes itérations de l'algorithme :

$\Rightarrow$ $\mathcal{C}[\![\ \mathbf{x+y+z < y-x}\ ]\!]_{\rho \rightarrow \$t0}$ *// appliquer : 3.1.3*

$\Rightarrow$ $\mathcal{C}[\![\ \mathbf{x+y+z}\ ]\!]_{\rho \rightarrow \$t0}$ *// appliquer : 3.1.1*
$\mathcal{C}[\![\ \mathbf{y-x}\ ]\!]_{\rho \rightarrow \$t1}$ *// appliquer : 3.1.2*
slt $t0 $t0 $t1

$\Rightarrow$ $\mathcal{C}[\![\ \mathbf{x}\ ]\!]_{\rho \rightarrow \$t0}$
$\mathcal{C}[\![\ \mathbf{y+z}\ ]\!]_{\rho \rightarrow \$t1}$ *// appliquer : 3.1.1*
add $t0 $t0 $t1
$\mathcal{C}[\![\ \mathbf{y}\ ]\!]_{\rho \rightarrow \$t1}$
$\mathcal{C}[\![\ \mathbf{x}\ ]\!]_{\rho \rightarrow \$t2}$
add $t1 $t1 $t2
slt $t0 $t0 $t1

$\Rightarrow$ $\mathcal{C}[\![\ \mathbf{x}\ ]\!]_{\rho \rightarrow \$t0}$ *// appliquer : 3.1.4*
$\mathcal{C}[\![\ \mathbf{y}\ ]\!]_{\rho \rightarrow \$t1}$ *// appliquer : 3.1.4*
$\mathcal{C}[\![\ \mathbf{z}\ ]\!]_{\rho \rightarrow \$t2}$ *// appliquer : 3.1.4*
add $t1 $t1 $t2
add $t0 $t0 $t1
$\mathcal{C}[\![\ \mathbf{y}\ ]\!]_{\rho \rightarrow \$t1}$ *// appliquer : 3.1.4*
$\mathcal{C}[\![\ \mathbf{x}\ ]\!]_{\rho \rightarrow \$t2}$ *// appliquer : 3.1.4*
add $t1 $t1 $t2
slt $t0 $t0 $t1

$\Rightarrow$ **lw** $t0 4($sp)
lw $t1 8($sp)
lw $t2 12($sp)
add $t1 $t1 $t2
add $t0 $t0 $t1
lw $t1 8($sp)
lw $t2 4($sp)
add $t1 $t1 $t2
slt $t0 $t0 $t1

Le code produit par la dernière itération est le code assembleur résultant de la compilation de l'expression initiale. Après l'exécution de ce code assembleur, le registre **$t0** contient la valeur de l'évaluation de l'expression initiale.

3.2 Structure de contrôle

Les structures de contrôle ont des règles de production tout comme les expressions. Contrairement aux règles de production des expressions, elles ne retournent aucune valeur. Le paramètre spécifiant le registre de destination est tout de même nécessaire puisqu'il indique les registres libres pour évaluer les expressions contenues dans les structures de contrôle. Le lecteur doit produire des labels uniques pour pouvoir imbriquer des structures de contrôles.

3.2.1 Condition *if*

$\mathcal{C}[\![$ **if** (<*expr*>$_1$) <*stm*>$_{true}$ **else** <*stm*>$_{false}$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$

$\mathcal{C}[\![$ <*expr*>$_1$ $]\!]_\rho \rightarrow \$tn$
beq \$zero, \$tn , $\texttt{lb}_{\texttt{false}}$
$\texttt{lb}_{\texttt{true}}$:
$\mathcal{C}[\![$ <*stm*>$_{true}$ $]\!]_\rho \rightarrow \$tn$
j $\texttt{lb}_{\texttt{end}}$
$\texttt{lb}_{\texttt{false}}$:
$\mathcal{C}[\![$ <*stm*>$_{false}$ $]\!]_\rho \rightarrow \$tn$
$\texttt{lb}_{\texttt{end}}$:

3.2.2 Boucle *for*

$\mathcal{C}[\![$ **for** (<*stm*>$_{init}$; <*expr*>$_{test}$; <*stm*>$_{incr}$) <*stm*>$_{body}$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$

$\mathcal{C}[\![$ <*expr*>$_{init}$ $]\!]_\rho \rightarrow \$tn$
j $\texttt{lb}_{\texttt{test}}$
$\texttt{lb}_{\texttt{loop}}$:
$\mathcal{C}[\![$ <*stm*>$_{body}$ $]\!]_\rho \rightarrow \$tn$
$\mathcal{C}[\![$ <*stm*>$_{incr}$ $]\!]_\rho \rightarrow \$tn$
$\texttt{lb}_{\texttt{test}}$:
$\mathcal{C}[\![$ <*expr*>$_{test}$ $]\!]_\rho \rightarrow \$tn$
bne \$zero, \$tn , $\texttt{lb}_{\texttt{loop}}$

3.2.3 Boucle *while*

$\mathcal{C}[\![$ **while** (<*expr*>$_{test}$) <*stm*>$_{body}$ $]\!]_\rho \rightarrow \$tn$

$\Rightarrow$

j $\texttt{lb}_{\texttt{test}}$
$\texttt{lb}_{\texttt{loop}}$:
$\mathcal{C}[\![$ <*stm*>$_{body}$ $]\!]_\rho \rightarrow \$tn$
$\texttt{lb}_{\texttt{test}}$:
$\mathcal{C}[\![$ <*expr*>$_{test}$ $]\!]_\rho \rightarrow \$tn$
bne \$zero, \$tn , $\texttt{lb}_{\texttt{loop}}$

3.2.4 Exemple

$$\mathcal{C}[\![\ \textbf{for(i=0 ; i<n ; i=i+1)}\ ]\!]_{\rho} \rightarrow \$t0$$
if(i<5) s=1 ;
else s=0 ;

Voici les différentes itérations de l'algorithme :

$\Rightarrow$ $\mathcal{C}[\![\ \textbf{for(i=0 ; i<n ; i=i+1)}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.2.2*
if(i<5) s=1 ;
else s=0 ;

$\Rightarrow$ $\mathcal{C}[\![\ \textbf{i=0}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.5*
j $\texttt{lb}_{\texttt{test}}$
$\texttt{lb}_{\texttt{loop}}$:
$\mathcal{C}[\![\ \text{if(i<5) s=1 else s=0 ;}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.2.1*
$\mathcal{C}[\![\ \text{i=i+1}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.5*
$\texttt{lb}_{\texttt{test}}$:
$\mathcal{C}[\![\ \text{i<n}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.3*
bne \$zero, \$t*0* , $\texttt{lb}_{\texttt{loop}}$

$\Rightarrow$ $\mathcal{C}[\![\ 0\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.6*
sw \$t*0* $\rho_{\mathbf{i}}$**(\$sp)**
j $\texttt{lb}_{\texttt{test}}$
$\texttt{lb}_{\texttt{loop}}$:
$\mathcal{C}[\![\ \text{i<5}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.3*
beq \$zero, \$t*0* , $\texttt{lb}_{\texttt{false}}$
$\texttt{lb}_{\texttt{true}}$:
$\mathcal{C}[\![\ \text{s=1}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.5*
j $\texttt{lb}_{\texttt{end}}$
$\texttt{lb}_{\texttt{false}}$:
$\mathcal{C}[\![\ \text{s=0}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.5*
$\texttt{lb}_{\texttt{end}}$:
$\mathcal{C}[\![\ \text{i+1}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.1*
sw \$t*0* $\rho_{\mathbf{i}}$**(\$sp)**
$\texttt{lb}_{\texttt{test}}$:
$\mathcal{C}[\![\ \text{i}\ ]\!]_{\rho} \rightarrow \$t0$ *// appliquer : 3.1.4*
$\mathcal{C}[\![\ \text{n}\ ]\!]_{\rho} \rightarrow \$t1$ *// appliquer : 3.1.4*
slt \$t*0* \$t*0* \$t*1*
bne \$zero, \$t*0* , $\texttt{lb}_{\texttt{loop}}$

$\Rightarrow$

```
    add $t0  $zero 0
    sw $t0  ρ_i($sp)
    j lb_test
lb_loop :
    C⟦ i ⟧_ρ → $t0                    // appliquer : 3.1.4
    C⟦ 5 ⟧_ρ → $t1                    // appliquer : 3.1.6
    slt $tn  $tn  $tn+1
    beq $zero, $t0 , lb_false
lb_true :
    C⟦ 1 ⟧_ρ → $t0                    // appliquer : 3.1.6
    sw $t0  ρ_s($sp)
    j lb_end
lb_false :
    C⟦ 0 ⟧_ρ → $t0                    // appliquer : 3.1.6
    sw $t0  ρ_s($sp)
lb_end :
    C⟦ i ⟧_ρ → $t0                    // appliquer : 3.1.4
    C⟦ 1 ⟧_ρ → $t1                    // appliquer : 3.1.6
    add $t0  $t0  $t1
    sw $t0  ρ_i($sp)
lb_test :
    lw $t0  ρ_i($sp)
    lw $t1  ρ_n($sp)
    slt $t0  $t0  $t1
    bne $zero, $t0 , lb_loop
```

```
⇒       add $t0  $zero 0
        sw $t0  0($sp)
        j lb_test
    lb_loop :
        lw $t0  0($sp)
        add $t1  $zero 5
        slt $tn  $tn  $tn+1
        beq $zero, $t0 , lb_false
    lb_true :
        add $t0  $zero 1
        sw $t0  4($sp)
        j lb_end
    lb_false :
        add $t0  $zero 0
        sw $t0  4($sp)
    lb_end :
        lw $t0  0($sp)
        add $t1  $zero 1
        add $t0  $t0  $t1
        sw $t0  0($sp)
    lb_test :
        lw $t0  0($sp)
        lw $t1  8($sp)
        slt $t0  $t0  $t1
        bne $zero, $t0 , lb_loop
```

La dernière itération montre le code assembleur produit pour l'énoncé initiale contenant des structures de contrôle.

Les variables ont été assignées aux cases mémoires comme suit :
- **i** → **$sp+0**
- **s** → **$sp+4**
- **n** → **$sp+8**

3.3 Optimisation

Le code produit par l'algorithme est très naïf. Par exemple, l'expression **i=i+1** n'utilise pas l'instruction assembleur **addi**. L'ajout de règles de production permet d'optimiser certains cas. Par contre, un nouveau problème se pose : Quelles règles choisir pour produire le code.

4 Appel de fonction

Cette section introduit les mécanismes de gestion de la pile d'exécution et la convention d'appel de fonction. Ces mécanismes sont requis pour le bon fonctionnement des structures de contrôle plus évoluées.

4.1 Pile d'exécution

Le processeur possède une pile d'exécution pour y entreposer de l'information temporaire. Comme les éléments de la pile d'exécution sont fréquemment employés, le processeur MIPS dédie un de ses registres à la gestion de la pile. Le registre `$sp` pointe toujours vers le dessus de la pile d'exécution.

La pile est renversé (ie : elle décroît). Ainsi, les données sont rajoutées en dessous des données existantes.

4.1.1 Ajout

Pour ajouter une donnée sur la pile d'exécution, le programmeur doit employer deux instructions. L'exemple ci-dessous ajoute la donnée contenue dans le registre $t0 sur la pile d'exécution.

```
sub $sp, $sp, 4
sw $t0, ($sp)
```

4.1.2 Retrait

Pour retirer une donnée de la pile d'exécution, le programmeur doit employer deux instructions. L'exemple ci-dessous retire une donnée de la pile d'exécution.

```
lw $t0, ($sp)
add $sp, $sp, 4
```

4.2 Convention d'appel de fonction

La notion de fonction permet d'obtenir des programmes bien structurés. Cette structure de contrôle est très importante. Le processeur ne fournit pas d'instructions spécialisées pour les appels de fonction.

Des questions que le programmeur assembleur doit se poser lors d'un appel de fonction sont :

1. Est-ce que la fonction va écraser les données de mes registres ?
2. Comment vais-je passer les paramètres à la fonction appelée ?
3. Comment vais-je recevoir le résultat de l'appel de fonction ?

Le programmeur peut définir sa propre convention d'appel de fonction. Si plusieurs conventions d'appel co-existent, elles doivent être compatibles. La

convention d'appel défini les registres qui doivent être sauvegardés respectivement par l'appelé et appelant. Elle défini les mécanismes de passage de paramètres et de retour de valeur.

Une convention d'appel de fonction standard existe pour faciliter l'interopérabilité des langages.

4.2.1 Appelant

Lors de l'appel de fonction, l'appelant doit effectuer certains traitements pour respecter cette convention. L'appelant doit :

1. sauvegarder les registres *caller-save*,
2. passer les paramètres selon la convention d'appel,
3. brancher à la fonction,
4. charger la valeur de retour,
5. détruire les paramètres,
6. restaurer les registres *caller-save*.

L'instruction **jal** (*jump and link*) branche à une l'adresse spécifique et met l'adresse de retour dans le registrea **$ra** Le mécanisme d'appel de fonction consiste à passer tous les paramètres et à appeler l'instruction **jal**.

```
; Empiler les parametres
sub $sp, $sp, 4
sw $t1, ($sp)
...

; Sauvergarder les caller-save
sub $sp, $sp, 4
sw $t0, ($sp)
...
sub $sp, $sp, 4
sw $t9, ($sp)

; Appel de fonction
jal func

; Restauration des caller-save
lw $t9, ($sp)
add $sp, $sp, 4
...
lw $t0, ($sp)
add $sp, $sp, 4

; Enlever les parametres
add $sp, $sp, N*4
```

FIG. 1 – Exemple d'appel de fonction

4.2.2 Appelé

Lorsqu'une fonction est appelée, elle exécute un bout de code appelé *prologue* qui ajuste les paramètres requis pour respecter la convention d'appel. L'*épilogue* est le code exécuté lorsque la fonction retourne une valeur.

Au moment où la fonction est appellée, l'appellé doit :

1. allouer l'espace sur la pile pour les variables locales,
2. sauvegarder les registres *callee-save*,
3. sauvergarder l'adresse de retour.

Lorsque la fonction termine, l'appellé doit :

1. retourner la valeur de retour selon la convention,
2. libérer le bloc d'activation,
3. brancher à l'appellant.

```
; PROLOGUE
; Allouer l'espace nécessaire sur la pile d'exécution
sub $sp, $sp, 4*N ; ; N = ( N-Callee-save + N-Local-var + N-temporaire )

; Sauvegarder les registres callee-save
sw $s0, 0($sp)
sw $s1, 4($sp)
...
; Sauvegarder l'adresse de retour
sw $ra, 32($sp)

; CORPS DE LA METHODE
lw $t0, 36($sp) ; ; N
lw $t1, 40($sp) ; ; N+4
add $t0, $t0, $t1
...
; EPILOGUE
; Valeur de retour dans $v0
add $v0, $zero, $t0

; Charger l'adresse de retour dans $ra
lw $ra, 32($sp)

; Liberer le bloc d'activation
add $sp, $sp, 4*N

; Branchement au parent
jr $ra
```

FIG. 2 – Exemple de définition de fonction

4.3 Exemple

Voici le code source de fib.c :

```
int fib(int n) {
  if(n<2) return n;
  return fib(n-1)+fib(n-2);
}

int main() {
  fib(2);
  return 0;
}
```

Le code assembleur produit :

```
 .text
 .globl fib
fib:
     sub  $sp, $sp, 12      # Allouer le bloc d'activation
     sw   $ra, 8($sp)       # Sauvergarder l'adresse de retour

     lw   $t0, 12($sp)      # Chargement de N
     slt  $t0, $t0, 2       # n < 2
     beq  $t0, $zero, fibgt

fiblt:
     lw   $t0, 12($sp)      # return N
     j fibend

fibgt:
     lw   $t0, 12($sp)      # Chargement de N
     sub  $t0, $t0, 1       # N-1
     sub  $sp, $sp, 4       # Push N-1
     sw   $t0, ($sp)
     jal  fib               # Appel de fib
     add  $sp, $sp, 4
     sw   $v0, 4($sp)       # temp1 <-- fib(n-1)

     lw   $t0, 12($sp)      # Chargement de N
     sub  $t0, $t0, 2       # N-2
     sub  $sp, $sp, 4       # Push N-2
     sw   $t0, ($sp)
     jal  fib               # Appel de fib
     add  $sp, $sp, 4
     sw   $v0, 0($sp)       # temp2 <-- fib(n+2)

     lw   $t0, 0($sp)       # Chargement de temp2
     lw   $t0, 4($sp)       # Chargement de temp1
     add  $t0, $t0, $t1     # return temp1 + temp2

fibend:
     add  $v0, $t0, $zero   # Deplacer le resultat dans v0
     lw   $ra, 8($sp)       # Chargement de l'adresse de retour
     add  $sp, $sp, 12      # Liberer le bloc d'activation

     jr   $ra               # Retour a l'appelant
```

```
   .text
   .globl main
main:
     sub  $sp, $sp, 4       # Allouer le bloc d'activation
     sw   $ra, ($sp)        # Sauvergarder l'adresse de retour

     add  $t0, $zero, 2
     sub  $sp, $sp, 4       # Push 2
     sw   $t0, ($sp)
     jal  fib               # Appel de fib
     add  $sp, $sp, 4       # Liberer les parametres

     add  $v0, $zero, $zero # Valeur de retour
     lw   $ra, ($sp)        # Chargement de l'adresse de retour
     add  $sp, $sp, 4       # Liberer le bloc d'activation
     jr   $ra               # Retour a l'appelant
```

5 Conclusion

La technique présentée permet de produire du code assembleur pour MIPS pour la majorité des codes. Par contre, le code produit n'est pas optimisé. De simples modifications au code résultant peuvent donner un code ayant de meilleures performances. De plus, la technique présentée ne traite pas le cas où la génération de code demande un plus grand nombre de registres que le nombre de registres du processeur.