RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

Réutilisation de la mémoire pour le pavage

Youcef Bouchebaba

Centre de Recherche en Informatique
Ecole des Mines de Paris
35, Rue Saint-honoré
77305 Fontainebleau- France
boucheba@cri.ensmp.fr

Résumé
Dans nos précédents travaux nous avons traité le problème de la fusion avec pavage sur une séquence de nids de boucles de *profondeur 2* et nous avons développé une technique d'allocation de la mémoire qui consiste à remplacer chaque tableau par un tampon circulaire contenant les données vivantes. Dans le cas de la fusion sans pavage, cette technique est efficace. Par contre, dans le cas de la fusion avec pavage, ces différents tampons contiennent des données qui ne sont pas utilisées dans le futur et les fonctions d'accès qui leurs sont associées sont compliquées. Pour éviter ces deux problèmes nous proposons de diviser chaque tampon en plusieurs sous-tampons. Les codes en entrée sont des séquences de nids de boucles de profondeur quelconque. Chacun de ces nids accède à des données suivant un *patron*, c'est à dire des boucles dans lesquelles les références à un même tableau sont égales à une translation près. Les dépendances sont donc uniformes.

Mots-clés : Pavage, Fusion, Réutilisation de la mémoire, Alignement de boucles.

1. Introduction

Dans cet article nous nous intéressons au problème de la réutilisation de la mémoire [7, 8, 11]. Nous exploitons la notion de données vivantes introduite initialement par Gannon et al [8] et reprise ensuite par Eisenbeis et al [6] afin de l'utiliser dans la fusion avec pavage d'une séquence de nids de boucles. Notre méthode consiste à remplacer le tableau associé à chaque nid par un ensemble de tampons qui contiennent les données vivantes.
Les codes considérés en entrée sont des séquences de nids de boucles de profondeur unique quelconque. Chacun de ces nids accède à des données suivant un *patron* : les références à un même tableau sont égales à une translation près. Par conséquent, les dépendances sont uniformes. Les bornes de ces différents nids sont des constantes numériques. Les différents tableaux ont tous la même dimension.
Dans nos précédents travaux [1, 2, 3], nous avons étudié le problème de l'application de la fusion avec pavage sur une séquence de nids de boucles et nous avons développé une technique d'allocation de tampons qui consiste à remplacer le tableau associé à chaque nid par un tampon circulaire. Dans le cas de la fusion sans pavage, les fonctions d'adressages associées aux différents tampons sont simples et le nombre de données mortes stockées dans ces différents tampons est négligeable devant le nombre de données vivantes. Par contre, dans le cas de la fusion avec pavage, les fonctions d'adressages sont compliquées et le pourcentage de données mortes stockées dans ces tampons est considérable. Pour palier ces deux problèmes, nous proposons de remplacer la tableau associé à chaque nid par plusieurs tampons.
Dans la *section 2*, nous donnons un exemple introductif. La *section 3* décrit les codes pris en entrée dans cet article. Dans la *section 4*, nous décrivons une technique d'allocation de tampons pour la fusion avec pavage d'une séquence de nids de boucles. La *section 5* présente les transformations que nous avons implantées dans *Pips* [13]. Notre conclusion est donnée dans la dernière section.

2. Exemple

Afin de mieux comprendre le problème traité, voici un exemple de code constitué de deux nids de boucles (*figure 1*), auquel nous appliquons le pavage qui est défini par la matrice de pavage $\mathcal{A} = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & 0 & 4 & 1 \end{pmatrix}$ et la matrice de permutation $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

```
do  (i = 1,  Ni)
      do  (j = 1,  Nj)
          A1(i,  j) = ....
      enddo
enddo
do  (i = 2,  Ni − 1)
      do  (j = 2,  Nj − 1)
          A2(i,  j) = A1(i − 1,  j) + A1(i,  j − 1)
                     +A1(i,  j) + A1(i,  j + 1) + A1(i + 1,  j)
      enddo
enddo
```

FIG. 1 – *Exemple de code*

Dans nos précédents travaux [1, 2, 3], nous avons détaillé les fonctions des matrices $\mathcal{A}$ et $\mathcal{P}$ et nous avons montré que nous devons translater le domaine d'itération du premier nid avec un délai $\vec{h} = (h_1, h_2)^t$ pour pouvoir appliquer ce pavage. Nous rappelons que la matrice $\mathcal{A}$ permet de spécifier les tailles des tuiles et que la matrice $\mathcal{P}$ permet de spécifier leur ordonnancement. Après l'application du pavage défini par les matrices $\mathcal{A}$ et $\mathcal{P}$ et la translation du domaine d'itération du premier nid avec un délai $\vec{h} = (h_1, h_2)^t = -(1, 1)^t$ permettant de respecter les dépendances et d'obtenir un nid de boucles complètement permutables, nous obtenons le code suivant :

```
do  (i1 = 0,  Ni/4 − 1)
 do  (j1 = 0,  Nj/4 − 1)
  do  (i2 = 0,  3)
   do  (j2 = 0,  3)
       A1(4i1 + i2 + 1,  4j1 + j2 + 1) = ....
       if  ((4i1 + i2 ≥ 2)  and  (4j1 + j2+ ≥ 2))
         A2(4i1 + i2,  4j1 + j2) =  A1(4i1 + i2 − 1,  4j1 + j2)
                                 +  A1(4i1 + i2,  4j1 + j2 − 1)
                                 +  A1(4i1 + i2,  4j1 + j2)
                                 +  A1(4i1 + i2,  4j1 + j2 + 1)
                                 +  A1(4i1 + i2 + 1,  4j1 + j2)
   enddo
  enddo
 enddo
enddo
```

FIG. 2 – *Code après application du pavage défini par $\mathcal{A}$ et $\mathcal{P}$ et la translation du domaine d'itération du premier nid avec* $\vec{h} = -(1, 1)^t$

Pour maintenir en mémoire les données vivantes et un minimum des données mortes, nous remplaçons le tableau A_1 par trois tampons :

1. Un tampon $B_{1,1}$ associé à la boucle i_1 qui contiendra les données vivantes produites dans une série de tuiles tel que $i_1 = a$ et qui seront utilisées dans la prochaine série tel que $i_1 = a + 1$. Ce tampon sera géré comme un tableau à 2 dimensions ($B_{1,1}[2, N_j]$) : 2 représente la plus grande projection de toutes les dépendances sur l'axe i et N_j est le nombre d'itérations sur l'axe j. Pour ranger les éléments du tableau A_1 dans ce tampon, nous lui associons une fonction d'accès $F_{1,1}(i_2, j_1, j_2)$= $(i_2 - 2, 4 * j_1 + j_2)$.
2. Un tampon $B_{1,2}$ associé à la boucle j_1 qui contiendra les données vivantes qui sont produites dans une tuile ($j_1 = b$ et i_1 est fixe) et qui seront utilisées dans la prochaine ($j_1 = b + 1$ et i_1 est fixe). De la même manière que la cas précédent, ce tampon sera géré comme un tableau à 2 dimensions ($B_{1,2}[4, 2]$) : 4 est la taille des tuiles sur l'axe i et 2 est la plus grande projection de toutes les dépendances sur l'axe j. La fonction d'accès de ce tampon est définie par $F_{1,2}(i_2, j_2)$= $(i_2, j_2 - 2)$.
3. Un tampon $B_{1,3}$ associé aux boucles internes i_2 et j_2 qui contiendra les données vivantes à l'intérieur d'une même tuile. Ce tampon sera considéré comme un tableau à 2 dimensions ($B_{1,3}[3, 4]$) : 3 est la plus grande projection de toutes les dépendances sur l'axe i plus 1 et 4 est la taille des tuiles sur l'axe j. La fonction d'accès de ce tampon est définie par $F_{1,3}(i_2, j_2) = (i_2 \bmod 3, j_2)$.

Par manque de place, nous ne donnons pas le code généré avec les trois tampons précédents. Dans la *section 4.4*, nous détaillerons l'étude des volumes mémoires des différents tampons et leurs fonctions d'accès pour des codes dont les nids sont de profondeur quelconque et pour n'importe quel pavage et ordonnancement.

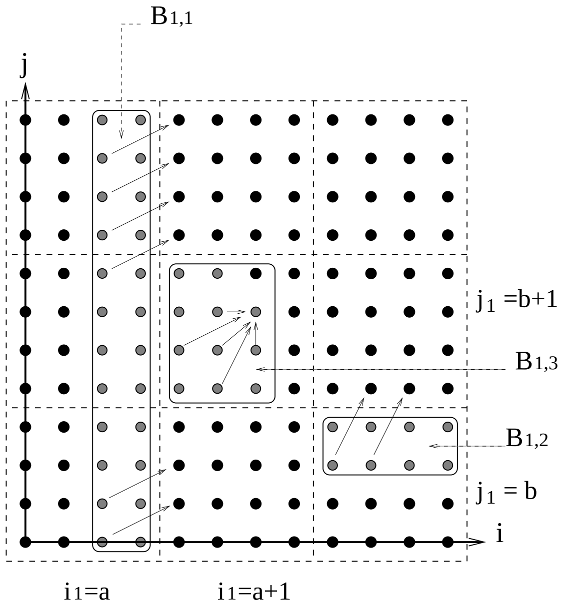

FIG. 3 – *Allocation de 3 tampons pour le code de la figure 2*

3. Code en entrée

Dans cet article, les codes en entrée sont des séquences de nids de boucles (chaîne de tâches) de profondeur unique quelconque (voir *figure 4* où $\otimes$ représente une opération quelconque). Chacun de ces nids accède à des données suivant un *patron*, il s'agit de boucles dans lesquelles les références à un même tableau sont égales à une translation près. Par conséquent, les dépendances sont uniformes. Les bornes de ces différents nids sont des constantes numériques. Les différents tableaux ont la même dimension. Le *patron* associé à chaque nid est représenté implicitement dans ce nid et donné explicitement par l'utilisateur sous forme d'un ensemble de vecteurs $R^k = \{\vec{v}_1^k, \vec{v}_2^k, \ldots, \vec{v}_{m_k}^k\}$.

$$
\begin{array}{l}
\texttt{do } \vec{i} \in D_1 \\
\quad A_1(\vec{i}) = A_0(\vec{i}+\vec{v}_1^1) \otimes \ldots \otimes A_0(\vec{i}+\vec{v}_{m_1}^1) \\
\texttt{enddo} \\
. \\
. \\
\texttt{do } \vec{i} \in D_k \\
\quad A_k(\vec{i}) = A_{k-1}(\vec{i}+\vec{v}_1^k) \otimes \ldots \otimes A_{k-1}(\vec{i}+\vec{v}_{m_k}^k) \\
\texttt{enddo} \\
. \\
. \\
\texttt{do } \vec{i} \in D_n \\
\quad A_n(\vec{i}) = A_{n-1}(\vec{i}+\vec{v}_1^n) \otimes \ldots \otimes A_{n-1}(\vec{i}+\vec{v}_{m_n}^n) \\
\texttt{enddo}
\end{array}
$$

FIG. 4 – *Code en entrée (séquence de nids de boucles)*

Le domaine D_0 associé au tableau A_0 est défini par l'utilisateur. Pour éviter des accès illégaux au niveau des différents tableaux, les domaines $D_{k=1\ldots n}$, sont dérivés de la manière suivante :
$D_k = \{\vec{i} \mid \forall \vec{v} \in R^k : \vec{i} + \vec{v} \in D_{k-1}\}$.
Nous supposons que les vecteurs des différents *patrons* sont ordonnés lexicographiquement de la manière suivante : $\vec{v}_1^k \preceq \vec{v}_2^k \preceq \ldots \preceq \vec{v}_{m_k}^k$.

4. Fusion avec pavage et allocation de plusieurs tampons

Dans nos précédents travaux [1, 2, 3], les codes que nous avons traités sont des séquences de nids de boucles de *profondeur 2*. Dans cet article, nous généralisons notre méthode à des séquences de nids de boucles de profondeur quelconque. Notre pavage est représenté par deux matrices; $\mathcal{A}$ qui définit les tuiles et $\mathcal{P}$ qui définit l'ordonnancement.

4.1. Matrice du pavage

La matrice du pavage $\mathcal{A}(n, 2*n)$ permet de spécifier les tailles des tuiles et de transformer chaque point $\vec{i} = (i_1, \cdots, i_n)^t \in Z^n$ en un autre point $\vec{i'} = (i'_1, \cdots, i'_{2n})^t \in Z^{2n}$ (*figure 5*). Cette matrice a la forme suivante :

$$
\begin{pmatrix}
a_{1,1} & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & a_{i,2i-1} & 1 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & a_{n,2n-1} & 1
\end{pmatrix}
$$

Tous les éléments de la $i^{ème}$ ligne de cette matrice sont nuls sauf :

1. $a_{i,2i-1}$ qui représente la taille des tuiles sur le $i^{ème}$ axe.
2. $a_{i,2i}$ qui est égal à 1.

Les relations entre $\vec{i}$ et $\vec{i'}$ sont données par :

1. $\vec{i}=\mathcal{A}\,\vec{i'}$.
2. $\vec{i'}=(\lfloor\frac{i_1}{a_{1,1}}\rfloor, i_1 \bmod a_{1,1}, \cdots, \lfloor\frac{i_m}{a_{m,2m-1}}\rfloor, i_m \bmod a_{m,2*m-1}, \cdots, \lfloor\frac{i_n}{a_{n,2n-1}}\rfloor, i_n \bmod a_{n,2*n-1})^t$.

```
do  (i' = ...)
   S_1 : C_1(i')  A_1(𝒜i' − h_1) = A_0(𝒜i' − h_1 + v_1^1) ⊗ ··· ⊗ A_0(𝒜i' − h_1 + v_{m_1}^1)
   .
   .
   S_k : C_k(i')  A_k(𝒜i' − h_k) = A_{k−1}(𝒜i' − h_k + v_1^k) ⊗ ··· ⊗ A_{k−1}(𝒜i' − h_k + v_{m_k}^k)
   .
   .
   S_n : C_n(i')  A_n(𝒜i' − h_n) = A_{n−1}(𝒜i' − h_n + v_1^n) ⊗ ··· ⊗ A_{n−1}(𝒜i' − h_n + v_{m_n}^n)
enddo
```

FIG. 5 – *Code après application de $\mathcal{A}$ et translation du domaine d'itération de chaque nid avec un délai $\vec{h}_k$*

$C_k(\vec{i}')$ représente une garde sur l'instruction S_k [1, 2, 3].

4.2. Matrice de permutation

La matrice de permutation $\mathcal{P}(2*n,\ 2*n)$ permet d'échanger plusieurs boucles du nid après l'application de la matrice $\mathcal{A}$ et de transformer chaque point $\vec{i}' = (i'_1, \cdots, i'_{2n})^t \in Z^{2n}$ en un autre point $\vec{l} = (l_1, \cdots, l_{2n})^t \in Z^{2n}$ (*figure 6*) avec $\vec{l}=\mathcal{P}\ \vec{i}'$.

```
do  (l = ...)
  S_1 : C_1(l)  A_1(𝒜𝒫^{−1}l − h_1) = A_0(𝒜𝒫^{−1}l − h_1 + v_1^1) ⊗ .... ⊗ A_0(𝒜𝒫^{−1}l − h_1 + v_{m_1}^1)
   .
   .
   S_k : C_k(l)  A_1(𝒜𝒫^{−1}l − h_k) = A_0(𝒜𝒫^{−1}l − h_k + v_1^k) ⊗ .... ⊗ A_0(𝒜𝒫^{−1}l − h_k + v_{m_k}^k)
   .
   .
   S_n : C_n(l)  A_1(𝒜𝒫^{−1}l − h_n) = A_0(𝒜𝒫^{−1}l − h_n + v_1^n) ⊗ .... ⊗ A_0(AP^{−1}l − h_n + v_{m_n}^n)
enddo
```

FIG. 6 – *Code après application de la matrice $\mathcal{P}$ sur le code de la figure 5*

4.3. Modélisation du pavage

Notre pavage sera représenté par une transformation ω :
$\omega : Z^n \rightarrow Z^{2*n}$
$\vec{i} \longmapsto \vec{l}= \mathcal{P}.(\lfloor \frac{i_1}{a_{1,1}} \rfloor, \imath_1 \bmod\ a_{1,1}, \cdots, \lfloor \frac{i_m}{a_{m,2m-1}} \rfloor, \imath_m \bmod\ a_{m,2m-1}, \cdots, \lfloor \frac{i_n}{a_{n,2n-1}} \rfloor, \imath_n \bmod\ a_{n,2n-1})^t$.
Pour pouvoir appliquer notre pavage sur le code de la *figure 4*, nous devons translater chaque domaine D_k avec un délai $\vec{h}_k$ [1, 2, 3]. Les différents délais doivent satisfaire la condition suivante :

$$\forall\ k, \forall \vec{i}, \forall q : \omega(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \preceq \omega(\vec{i}) \qquad (1)$$

4.4. Allocation de tampons

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les mêmes données, nous avons proposé dans nos précédents travaux [1, 2] de remplacer les tableaux A_1, A_2,...et A_{n-1} par des tampons circulaires B_1, B_2,...et B_{n-1}. Un tampon B_i est un tableau à une dimension qui contiendra les données vivantes du tableau A_i. Cette technique est efficace pour la fusion sans pavage. Par contre, dans la cas de la fusion avec pavage, cette technique possède deux inconvénients :

– Des données mortes sont stockées dans ces tampons afin de simplifier les fonctions d'accès et la génération du code.

– La taille de ces tampons explose lorsque les tailles des tuiles deviennent très grandes.

Pour palier ces deux problèmes, nous proposons de remplacer chaque tableau A_k par $n+1$ tampons :

4.4.1. Tampons associés aux boucles externes

Le pavage permet de transformer un nid de profondeur n en un autre nid de profondeur $2n$. Les n boucles externes vont parcourir les tuiles et les n boucles internes vont parcourir les itérations à l'intérieur de ces tuiles. Pour chaque boucle externe l_m, nous faisons correspondre un tampon $B_{k,m}$ (k correspond au tableau A_K et m correspond à la boucle l_m) qui contiendra les données vivantes du tableau A_k produites dans des tuiles tel que ($l_m = b$) et qui seront utilisées dans les prochaines tuiles tel que ($l_m = b+1$). Pour calculer les tailles de ces tampons, nous utilisons les notations suivantes :

1. $E(n,n)$ la matrice de permutation des boucles externes : $E_{i,j} = \begin{cases} 1 \text{ si } \mathcal{P}_{i,2j-1} = 1 \\ 0 \quad\quad \text{sinon} \end{cases}$
2. $I(n,n)$ la matrice de permutation des boucles internes : $I_{i,j} = \begin{cases} 1 \text{ si } \mathcal{P}_{i+n,2j} = 1 \\ 0 \quad\quad \text{sinon} \end{cases}$
3. $\vec{T} = (T_1, \cdots, T_n)^t$le vecteur des tailles d'une tuile : $T_i = a_{i,2i-1}$.
4. $\vec{N}_k = (N_{k,1}, \cdots, N_{k,n})^t$ avec $N_{k,m}$ est le nombre d'itérations de la boucle i_m du *nid k* du code de la *figure 4*.
5. $\vec{d}_k = (d_{k,1}, \cdots, d_{k,n})^t$ avec $d_{k,m}$ est la plus grande projection de toutes les dépendances sur le $m^{ème}$ axe (dépendance entre l'instruction qui écrit dans le tableau A_k et celle qui lit ce même tableau).
6. $\vec{T}' = E\,\vec{T}$, $\vec{N}' = E\,\vec{N}$ et $\vec{d}' = E\,\vec{d}$.

Le volume mémoire du tampon $B_{k,m}$ associé au tableau A_k et à la $m^{ème}$ boucle externe est inférieur à $V_{k,m} = \prod_{i=1}^{m-1} T'_i * d'_{k,m} * \prod_{i=m+1}^{n} N'_{k,i}$.
Chaque coefficient dans cette formule correspond à une dimension dans le tampon $B_{k,m}$. Il y a $n!$ façons d'organiser les dimensions de ce tampon. Dans cet article, nous considérons cette organisation $[T'_1, \cdots, T'_{m-1}, d'_{k,m}, N'_{K,m+1}, \cdots, N'_{K,n}]$.

Il est très important d'étudier la meilleure organisation de ces dimensions en fonction des matrices E et I afin d'augmenter la localité des données. Dans nos prochains travaux nous nous intéresserons à cette partie du problème qui va jouer un très grand rôle dans la minimisation des transferts des données entre les différents niveaux de la hiérarchie mémoire.

Fonctions d'accès

Afin de ranger les éléments du tableau A_K dans les différents tampons qui lui sont associés, nous définissons pour chaque tampon $B_{k,m}$ une fonction d'accès $F_{k,m}$:
$F_{k,m}(\vec{i}') = (E_1\vec{i}'_{in}, \cdots, E_{m-1}\vec{i}'_{in}, E_m(\vec{i}'_{in} - (\vec{T} - \vec{d}_k)), (E_{m+1}\vec{T})\,(E_{m+1}\vec{i}'_E) + E_{m+1}\vec{i}'_{in}, \cdots, (E_n\,\vec{T})(E_n\vec{i}'_E) + E_n\vec{i}'_{in})$.

1. E_m représente la $m^{ème}$ ligne de la matrice E.
2. $\vec{i}'_E$ est le sous vecteur de $\vec{i}'$ qui permet de parcourir les tuiles.
3. $\vec{i}'_{in}$ est le sous vecteur de $\vec{i}'$ qui permet de parcourir les itérations à l'intérieur des tuiles.

4.4.2. Tampon associé aux boucles internes

Pour toutes les boucles internes, nous définissons un seul tampon $B_{k,n+1}$ qui contiendra les données vivantes à l'intérieur d'une même tuile. Le volume mémoire de ce tampon est borné par :
$V_{k,n+1} = (I_1\,\vec{d}+1) * \prod_{k=2}^{n}(I_k\,\vec{T})$.
De la même manière que le cas précédent, chaque coefficient dans cette formule, correspondra à une dimension dans le tampon $B_{k,n+1}$. Il y a $n!$ façons d'organiser ces dimensions. Pour avoir la meilleure

localité dans ce cas, il suffit d'opter pour cette organisation $[I_1.\vec{d}+1, I_2.\vec{T}, \cdots, I_n.\vec{T}]$.

Fonction d'accès

La fonction d'accès associée au tampon $B_{k,n+1}$ est définie par :
$F_{k,n+1} = ((I_1\ \vec{i}'_{in}) \bmod\ (I_1\ \vec{d}+1), I_2.\vec{i}'_{in}, \cdots, I_n\ \vec{i}'_{in})$.

5. Implantation

Dans le cadre du projet *Conso* [12], nous avons implanté dans *Pips* [13] :

1. Fusion d'une séquence de nids de boucles.
2. Fusion d'une séquence de nids de boucles avec le remplacement de chaque tableau par un tampon circulaire.
3. Fusion avec pavage d'une séquence de nids de boucles.
4. Fusion avec pavage d'une séquence de nids de boucles avec le remplacement de chaque tableau par $n+1$ tampons (Transformation décrite dans cet article).

Dans notre implantation, les codes en entrée sont des séquences de nids de boucles de profondeur quelconque et tous les ordonnancements possibles sont pris en considération (la matrice $\mathcal{P}$ est un paramètre).
Dans la version finale de ce papier, nous comparerons les défauts de cache causés par ces différentes transformations.

6. Conclusion

Ces travaux sont la suite de nos précédents travaux sur l'application simultanée de la fusion et du pavage sur une séquence de nids de boucles [1, 2, 3]. Nous avons présenté une nouvelle méthode d'allocation de la mémoire qui consiste à remplacer chaque tableau par $n+1$ tampons avec n est la profondeur des différents nids. La profondeur des nids du code en entrée et la matrice de permutation $\mathcal{P}$ sont quelconques. Cette transformation est implantée d'une manière efficace dans *Pips* [13].
Dans cet article, nous avons montré comment nous pouvons combiner les techniques de transformations de programmes comme la fusion de boucles avec alignement[4, 5, 10], le pavage [9, 14] et la réutilisation de la mémoire [7, 8, 11].
Dans nos prochains travaux, nous nous intéresserons à l'étude de la combinaison de l'allocation des registres avec l'allocation des tampons et nous étudierons l'organisation des dimensions des différents tampons afin d'augmenter la localité des données.

Bibliographie

1. Youcef Bouchebaba and Fabien Coelho. Buffered tiling for sequences of loops nests. In *Compilers and Operating Systems for Low Power 2001*, pages 31–36.
2. Youcef Bouchebaba and Fabien Coelho. Pavage pour une séquence de nids de boucles. *Soumis à la revue Technique et sciences informatiques*, 2000.
3. Youcef Bouchebaba and Fabien Coelho. Tiling pour une séquence de nids de boucles. In *Renpar2001*, pages 31–36, Paris, 2001.
4. Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.
5. Alain Darte and Guillaume Huard. Loop shifting for loop compaction. *International Journal of Parallel Programming*, 28(5):499–534, 2000.
6. C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. rapport de recherche 1262, INRIA, 1990.
7. P. Feautrier and V. Lefebvre. Storage management in parallel programs. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, London, 1997.
8. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distibuted Computing*, 5(10):587–616, 1988.

9. F. Irigoin and R. Triolet. Supernode partitioning. In *In Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.
10. W. Pugh and E. Rosser. Iteration space slicing for locality. In *LCPC99*, pages 165–184, San Diego, CA, 1999.
11. Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.
12. Équipe ATIP. Projet conso : Energie, temps réel et systèmes embarqués (http://www.cri.ensmp.fr/projet_conso.html).
13. Équipe PIPS. Pips (interprocedural parallelizer for scientific programs) http://www.cri.ensmp.fr/pips.
14. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.