Tiling and Memory Reuse for Sequences of Nested Loops

Youcef Bouchebaba and Fabien Coelho

CRI, ENSMP, 35, rue Saint Honoré, 77305 Fontainebleau, France
{boucheba, coelho}@cri.ensmp.fr

Abstract. Our aim is to minimize the electrical energy used during the execution of signal processing applications that are a sequence of loop nests. This energy is mostly used to transfer data among various levels of memory hierarchy. To minimize these transfers, we transform these programs by using simultaneously loop permutation, tiling, loop fusion with shifting and memory reuse. Each input nest uses a stencil of data produced in the previous nest and the references to the same array are equal, up to a shift. All transformations described in this paper have been implemented in *pips*, our optimizing compiler and cache misses reductions have been measured.

1 Introduction

In this paper we are interested in the application of fusion with tiling to a sequence of loop nests and in memory reuse in the merged and tiled nest. Our transformations aim at improving data locality so as to replace costly transfers from main memory to cheaper cache or register memory accesses. Many authors have worked in tiling [9,16,14], fusion [5,4,11,17], loop shifting [5,4] and memory reuse [6,13,8]. Here, we combine these techniques to apply them to sequence of loop nests.

We assume that input programs are sequences of loop nests. Each of these nests uses a stencil of data produced in the previous nest and the references to the same array are equal, up to a shift. Consequently, the dependences are uniform. We limit our method to this class of code (chains of jobs), because the problem of loop fusion with shifting in general (graphs of jobs) is *NP_hard* [4].

Our tiling is used as a loop transformation [16] and is represented by two matrices: (1) a matrix $\mathcal{A}$ of hierarchical tiling that gives the various tile coefficients and (2) a permutation matrix $\mathcal{P}$ that allows to exchange several loops and so to specify the organization of tiles and to consider all possible schedules.

After application of fusion with tiling, we have to guarantee that all necessary data for the computation of a given iteration has already been computed by the previous iterations. For this purpose, we shift the computation of each nest by a delay $\boldsymbol{h_k}$. Contrary to the other works, it is always possible to apply our fusion with tiling. To avoid loading several times the same data, we use the notion of live data introduced initially by Gannon et al [8] and applied by Einsenbeis et al [6], to fusion with tiling. Our method replaces the array associated to each nest by a set of buffers that will contain the live data of the corresponding array.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 255–264.

2 Input Code

The input codes are signal processing applications [10], that are sequences of loop nests of equal but arbitrary depth (see *Figure 1 (a)*). Each of these nests uses a stencil of data produced in the previous nest and represented by a set $V^k = \{\boldsymbol{v}_1^k,\ \boldsymbol{v}_2^k, \cdots, \boldsymbol{v}_{m_k}^k\}$. The references to the same array are equal, up to a shift. The bounds of these various nests are numerical constants and the various arrays have the same dimension.

```
do i_1 ∈ D_1
 A_1(i_1) = A_0(i_1 + v_1^1) ⊗ .... ⊗ A_0(i_1 + v_{m_1}^1)
enddo
.
do i_k ∈ D_k
 A_k(i_k) = A_{k-1}(i_k + v_1^k) ⊗ .... ⊗ A_{k-1}(i_k + v_{m_k}^k)
enddo
.
do i_n ∈ D_n
 A_n(i_n) = A_{n-1}(i_n + v_1^n) ⊗ .... ⊗ A_{n-1}(i_n + v_{m_n}^n)
enddo
```
(a) Input code in general form

```
do (i = 4, N − 5)
 do (j = 4, N − 5)
  A_1(i, j) = A_0(i − 4, j) + A_0(i, j − 4)
  +A_0(i, j) + A_0(i, j + 4) + A_0(i + 4, j)
 enddo
enddo
do (i = 8, N − 9)
 do (j = 8, N − 9)
  A_2(i, j) = A_1(i − 4, j) + A_1(i, j − 4)
  +A_1(i, j) + A_1(i, j + 4) + A_1(i + 4, j)
 enddo
enddo
```
(b) Specific example

Fig. 1. General input code and a specific example. Where ⊗ represents any operation.

Domain D_0 associated with the array A_0 is defined by the user. To avoid illegal accesses to the various arrays, the domains D_k $(1 \leq k \leq n)$ are derived in the following way: $D_k = \{\boldsymbol{i} \mid \forall \boldsymbol{v} \in V^k :\ \boldsymbol{i} + \boldsymbol{v} \in D_{k-1}\}$. We suppose that vectors of the various stencils are lexicographically ordered: $\forall k :\ \boldsymbol{v}_1^k \preceq \boldsymbol{v}_2^k \preceq ..... \preceq \boldsymbol{v}_{m_k}^k$. In this paper, we limited our study to codes given in *Figure 1 (a)*, $A_k(\boldsymbol{i})$ is computed using elements of array A_{k-1}. Our method is generalizable easily to a code, such as the computation of the element $A_k(\boldsymbol{i})$ in the nest k, it will be according to the arrays $A_0, \cdots A_{k-1}$.

3 Loop Fusion

To merge all the nests into one, we should make sure that all the elements of array A_{k-1} that are necessary for the computation of an element $A_k(\boldsymbol{i}_k)$ at iteration $\boldsymbol{i}_k$ in the merged nest have already been computed by previous iterations. To satisfy this condition, we shift the iteration domain of every nest by a delay $\boldsymbol{h}_k$. Let Time_k be the shifting function associated to nest k and defined in the following way: $\text{Time}_k : D_k \rightarrow Z^n$ so that $\boldsymbol{i}_k \longmapsto \boldsymbol{i} = \boldsymbol{i}_k + \boldsymbol{h}_k$.

The fusion of all nests is legal if and only if each shifting function Time_k meets the following condition:

$$\forall \boldsymbol{i}_k,\ \forall \boldsymbol{i}_{k+1}\ , \forall \boldsymbol{v} \in V^{k+1}\ : \boldsymbol{i}_k = \boldsymbol{i}_{k+1} + \boldsymbol{v} \Rightarrow \text{Time}_k(\boldsymbol{i}_k) \preceq \text{Time}_{k+1}(\boldsymbol{i}_{k+1}) \quad (1)$$

The condition (1) means that if an iteration $\boldsymbol{i}_k$ produces an element that will be consumed by iteration $\boldsymbol{i}_{k+1}$, then the shift of the iteration $\boldsymbol{i}_k$ by Time_k should be lexicographically lower than the shift of the iteration $\boldsymbol{i}_{k+1}$ by Time_{k+1}.

The merged code after shifting of the various iteration domains is given in *Figure 2.* S_k is the instruction label and $D_{iter} = \cup_{k=1}^{n}(D'_k)$, with $D'_k = \{\boldsymbol{i} = \boldsymbol{i}_k + \boldsymbol{h}_k \mid \boldsymbol{i}_k \in D_k\}$ the shift of domain D_k by vector $\boldsymbol{h}_k$. This domain is not necessarily convex. If not, we use its convex hull to generate the code. As instruction S_k might not be executed at each iteration of domain D_{iter}, we guard it by condition $C_k(\boldsymbol{i}) =$ if $(\boldsymbol{i} \in D'_k)$, which can be later eliminated [12].

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

Fig. 2. Merged nest.

As $\boldsymbol{v}_1^k \preceq \boldsymbol{v}_2^k \preceq ..... \preceq \boldsymbol{v}_{m_k}^k$, the validity condition of fusion given in (1) will be equivalent to $-\boldsymbol{h}_k \succeq -\boldsymbol{h}_{k+1} + \boldsymbol{v}_{m_{k+1}}^{k+1}$ $(1 \leq k \leq n-1)$.

3.1 Fusion with Buffer Allocation

To save memory space and to avoid loading several times the same element, we replace the arrays A_1, A_2,..and A_{n-1} by circular buffers B_1, B_2,..and B_{n-1}. Buffer B_i is a one-dimensional array that will contain the live data of array A_i. Each of these buffers will be managed in a circular way and an access function will be associated with it to load and store its elements.

Live data. Let $\boldsymbol{O}_k$ and $\boldsymbol{N}_k + \boldsymbol{O}_k - \boldsymbol{1}$ respectively the lower and upper bound of domain $D'_k : D'_k = \{\boldsymbol{i} \mid \boldsymbol{O}_k \leq \boldsymbol{i} \leq \boldsymbol{N}_k + \boldsymbol{O}_k - \boldsymbol{1}\}$. The memory volume $M_k(\boldsymbol{i})$ corresponding to an iteration $\boldsymbol{i} \in D'_k$ $(2 \leq k \leq n)$ is the number of elements of the array A_{k-1} that were defined before $\boldsymbol{i}$ and that are not yet fully used: $M_k(\boldsymbol{i}) = |E_k(\boldsymbol{i})|$ with $E_k(\boldsymbol{i}) = \{\boldsymbol{i}_1 \in D'_{k-1} | \ \exists \boldsymbol{v} \in V^k, \ \exists \ \boldsymbol{i}_2 \in D'_k \ : \boldsymbol{i}_1 - \boldsymbol{h}_{k-1} = \boldsymbol{i}_2 - \boldsymbol{h}_k + \boldsymbol{v}$ and $\boldsymbol{i}_1 \preceq \boldsymbol{i} \preceq \boldsymbol{i}_2\}$.

At iteration $\boldsymbol{i}$, to compute $A_k(\boldsymbol{i} - \boldsymbol{h}_k)$, we use m_k elements of array A_{k-1} produced respectively by $\boldsymbol{i}_1, \cdots, \boldsymbol{i}_{m_k}$ such that $\boldsymbol{i}_q = \boldsymbol{i} - (\boldsymbol{h}_k - \boldsymbol{h}_{k-1} - \boldsymbol{v}_q^k)$.

The oldest of these productions is $\boldsymbol{i}_1$. Consequently the volume $M_k(\boldsymbol{i})$ is bounded by the number of iterations in D'_{k-1} between $\boldsymbol{i}_1$ and $\boldsymbol{i}$. This upper boundary is given by $\mathrm{Sup}_k = \boldsymbol{C}_k.\ (\boldsymbol{h}_k - \boldsymbol{h}_{k-1} - \boldsymbol{v}_1^k) + 1$ with

$\boldsymbol{C}_k = (\prod_{i=2}^{n} N_{k-1,i}, \prod_{i=3}^{n} N_{k-1,i}, \cdots, \prod_{i=n-1}^{n} N_{k-1,i},\ N_{k-1,n},\ 1)^t$ and $N_{k,i}$ is the i^{th} component of $\boldsymbol{N}_k$.

Code generation. Let B_k $(1 \leq k \leq n-1)$ be the buffers associated with arrays A_k $(1 \leq k \leq n-1)$ and $succ(\boldsymbol{i})$ the successor of $\boldsymbol{i}$ in the domain D'_k. Sup_{k+1}, given previously, represents an upper bound for the number of live data

of array A_k. Consequently the size of buffer B_k can safely be set to Sup_{k+1} and we associate with it the access function $F_k : D'_k \rightarrow N$ such that:

1. $F_k(\boldsymbol{O}_k) = 0$
2. $F_k(succ(\boldsymbol{i})) = \begin{cases} F_k(\boldsymbol{i}) + 1 \text{ if } (F_k(\boldsymbol{i}) \neq \mathrm{Sup}_{k+1} - 1) \\ 0 \qquad\qquad \text{otherwise} \end{cases}$

To satisfy these two conditions, it is sufficient to choose
$F_k(\boldsymbol{i}) = (\boldsymbol{C}_k \ . \ (\boldsymbol{i} - \boldsymbol{O}_k)) \mod \mathrm{Sup}_{k+1}$.

Let's consider statement S_k of the merged code in *Figure* 2. At iteration $\boldsymbol{i}$, we compute the element $A_k(\boldsymbol{i} - \boldsymbol{h}_k)$ as a function of the m_k elements of array A_{k-1} produced respectively by $\boldsymbol{i}_1$, $\boldsymbol{i}_2$,$\cdots$and $\boldsymbol{i}_{m_k}$. The element $A_k(\boldsymbol{i} - \boldsymbol{h}_k)$ is stored in the buffer B_k at position $F_k(\boldsymbol{i})$. The elements of array A_{k-1} are already stored in the buffer B_{k-1} at positions $F_{k-1}(\boldsymbol{i}_1)$, $F_{k-1}(\boldsymbol{i}_2)$,..,$F_{k-1}(\boldsymbol{i}_{m_k})$ ($\boldsymbol{i}_q = \boldsymbol{i} - (\boldsymbol{h}_k - \boldsymbol{h}_{k-1} - \boldsymbol{v}_q^k)$). Thus the statement S_k will be replaced by
$C_k(\boldsymbol{i})$ $B_k(F_k(\boldsymbol{i})) = B_{k-1}(F_{k-1}(\boldsymbol{i}_1)) \otimes \cdots \otimes B_{k-1}(F_{k-1}(\boldsymbol{i}_{m_k}))$.

4 Tiling with Fusion

A lot of work on tiling has been done but most of it is only dedicated to a single loop nest. In this paper, we present a simple and effective method that simultaneously applies tiling with fusion to a sequence of loop nests. Our tiling is used as a loop transformation [16] and is represented by two matrices: (1) a matrix $\mathcal{A}$ of hierarchical tiling that gives the various coefficients of tiles and (2) a permutation matrix $\mathcal{P}$ that allows to exchange several loops and so to specify the organization of tiles and to consider all possible tilings. As for fusion, the first step before applying tiling with fusion to a code similar to the one in *Figure 1 (a)* is to shift the iteration domain of every nest by a delay $\boldsymbol{h}_k$. We note by $D'_k = \{\boldsymbol{i} = \boldsymbol{i}_k + \boldsymbol{h}_k | \boldsymbol{i}_k \in D_k\}$ the shift of domain D_k by vector $\boldsymbol{h}_k$.

4.1 One-Level Tiling

In this case, we are interested only in data that lives in the cache memory. Thus our tiling is at one level.

Matrix $\mathcal{A}$. Matrix $\mathcal{A}(n,\ 2n)$ defines the various coefficients of tiles and allows us to transform every point $\boldsymbol{i} = (i_1, \cdots,\ i_n)^t \in \cup_{i=1}^{n}(D'_k)$ into a point $\boldsymbol{i'} = (i'_1, \cdots,\ i'_{2n})^t \in Z^{2n}$ (*figure 3*). This matrix has the following shape:

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

All the elements of the i^{th} line of this matrix are equal to zero except: 1) $a_{i,2i-1}$, which represents the size of tiles on the i^{th} axis and 2) $a_{i,2i}$, which is equal to 1.

```
do (i' = ...)
  S1 : C1(i')  A1(Ai' − h1) = A0(Ai' − h1 + v1^1) ⊗ .... ⊗ A0(Ai' − h1 + v_{m1}^1)
  .
  Sk : Ck(i')  Ak(Ai' − hk) = Ak−1(Ai' − hk + v1^k) ⊗ .... ⊗ Ak−1(Ai' − hk + v_{mk}^k)
  .
  Sn : Cn(i')  An(Ai' − hn) = An−1(Ai' − hn + v1^n) ⊗ .... ⊗ An−1(Ai' − hn + v_{mn}^n)
enddo
```

Fig. 3. Code after application of $\mathcal{A}$.

The relationship between $\boldsymbol{i}$ and $\boldsymbol{i'}$ is given by:

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

Matrix $\mathcal{P}$. The matrix $\mathcal{A}$ has no impact on the execution order of the initial code. Permutation matrix $\mathcal{P}(2n,\ 2n)$ allows to exchange several loops of code in *Figure 3* and is used to specify the order in which the iterations are executed. This matrix transforms every point $\boldsymbol{i'} = (i'_1,\ i'_2, \cdots,\ i'_{2n})^t \in Z^{2n}$ (*Figure 3*) into a point $\boldsymbol{l} = (l_1,\ l_2, \cdots,\ l_{2n})^t \in Z^{2n}$ such as $\boldsymbol{l} = \mathcal{P}\ \boldsymbol{i'}$. Every line and column of this matrix has one and only one element that is equal to 1.

Tiling modeling. Our tiling is represented by a transformation ω_1:
$\omega_1 : Z^n \rightarrow Z^{2n}$

$$\boldsymbol{i} \longmapsto \boldsymbol{l} = \mathcal{P}\ .\ (\lfloor \frac{i_1}{a_{1,1}} \rfloor,\ \ i_1 \bmod a_{1,1}, \cdots,\ \lfloor \frac{i_m}{a_{m,2m-1}} \rfloor,\ \ i_m \bmod a_{m,2m-1}, \cdots,$$
$$\lfloor \frac{i_n}{a_{n,2n-1}} \rfloor,\ \ i_n \bmod a_{n,2n-1})^t.$$

As mentioned in our previous work [1,2], the simultaneous application of tiling with fusion to the code in *Figure 1(a)* is valid if and only if:

$$\forall\ k, \forall \boldsymbol{i} \in D'_k, \forall q : \omega_1(\boldsymbol{i} + \boldsymbol{v}_q^{k+1} - \boldsymbol{h}_{k+1} + \boldsymbol{h}_k) \preceq \omega_1(\boldsymbol{i}) \tag{2}$$

One legal delay of *formula (2)*, is $-\boldsymbol{h}_k = -\boldsymbol{h}_{k+1} + (\max_l\ v_{l,1}^{k+1}, \cdots,\ \max_l\ v_{l,n}^{k+1})^t$ and $\boldsymbol{h}_n = \boldsymbol{0}$. Where $v_{l,i}^{k+1}$ is the i^{th} component of vector $\boldsymbol{v}_l^{k+1}$. The choice of this delay makes the merged nest fully permutable. We know that if a loop nest is fully permutable, we can apply to it any tiling parallel to its axis [15].

Buffer allocation. To maintain in memory the live data and to avoid loading several times the same data, we suggested in our previous work [1,2] to replace arrays A_1, A_2,...and A_{n-1} by circular buffers B_1, B_2,...and B_{n-1}. A buffer B_i is a one-dimensional array that contains the live data of array A_i. This technique is effective for the fusion without tiling. On the other hand, in the case of fusion with tiling, this technique has two drawbacks: 1) dead data are stored in these buffers to simplify access functions and 2) the size of these buffers increases when the tile size becomes large. For the purpose of eliminating these two problems, we replace every array A_k by $n+1$ buffers.

a) Buffers associated with external loops: One-level tiling allows to transform a nest of depth n into another nest of depth $2n$. The n external loops iterate over tiles, while the n internal loops iterate over iterations inside these tiles. For every external loop m, we associate a buffer $B_{k,m}$ (k corresponds to array A_k) that will contain the live data of array A_k produced in tiles such that ($l_m = b$) and used in the next tiles such that ($l'_m = b+1$). To specify the size of these buffers, we use the following notations:

- $E(n,\ n)$, the permutation matrix of external loops: $E_{i,j} = \begin{cases} 1 \text{ if } \mathcal{P}_{i,2j-1} = 1 \\ 0 \quad \text{otherwise} \end{cases}$
- $I(n,\ n)$, the permutation matrix of internal loops: $I_{i,j} = \begin{cases} 1 \text{ if } \mathcal{P}_{i+n,2j} = 1 \\ 0 \quad \text{otherwise} \end{cases}$
- $\boldsymbol{T} = (T_1, \cdots,\ T_n)^t$, the tile size vector: $T_i = a_{i,2i-1}$;
- $\boldsymbol{N}_k = (N_{k,1}, \cdots,\ N_{k,n})^t$ where $N_{k,m}$ is the number of iterations of loop i_m in *nest* k of code in *Figure 1(a)*.
- $\boldsymbol{d}_k = (d_{k,1}, \cdots,\ d_{k,n})^t$, where $d_{k,m}$ is the maximum of the projections of all dependences on the m^{th} axis (dependences connected to array A_k);
- $\boldsymbol{T'} = E\ \boldsymbol{T}$, $\boldsymbol{N'}_k = E\ \boldsymbol{N}_k$ and $\boldsymbol{d'}_k = E\ \boldsymbol{d}_k$.

The memory volume required for buffer $B_{k,m}$ associated with array A_k and the m^{th} external loop is less than $V_{k,m} = \prod_{i=1}^{m-1} T'_i * d'_{k,m} * \prod_{i=m+1}^{n} N'_{k,i}$. Every coefficient in this formula corresponds to a dimension in the buffer $B_{k,m}$. There are $n!$ ways to organize the dimensions of this buffer. In this paper, we will consider the following organization:$[T'_1, .., T'_{m-1},\ d'_{k,m}, N'_{k,m+1}, .., N'_{k,n}]$.

To locate the elements of array A_k in the various buffers associated with it, we define for every buffer $B_{k,m}$ an access function $F_{k,m}$:

$$F_{k,m}(\boldsymbol{i'}) = (E_1\boldsymbol{i'}_{in}, \cdots,\ E_{m-1}\boldsymbol{i'}_{in},\ E_m(\boldsymbol{i'}_{in} - (\boldsymbol{T} - \boldsymbol{d}_k)),\ (E_{m+1}\boldsymbol{T})\ (E_{m+1}\boldsymbol{i'}_E) \\ +E_{m+1}\boldsymbol{i'}_{in}, \cdots,\ (E_n\ \boldsymbol{T})(E_n\boldsymbol{i'}_E) + E_n\boldsymbol{i'}_{in}), where:$$

- E_m represents the m^{th} line of matrix E;
- $\boldsymbol{i'}_E$ is sub vector of $\boldsymbol{i'}$ which iterate over tiles;
- $\boldsymbol{i'}_{in}$ is sub vector of $\boldsymbol{i'}$ which iterate over iterations inside tiles.

b) Buffers associated with internal loops: For all the internal loops, we define a single buffer $B_{k,n+1}$ which contains the live data inside the same tile. The memory volume of this buffer is bounded by $V_{k,n+1} = (I_1\ \boldsymbol{d}_k + 1) * \prod_{k=2}^{n}(I_k\ \boldsymbol{T})$.

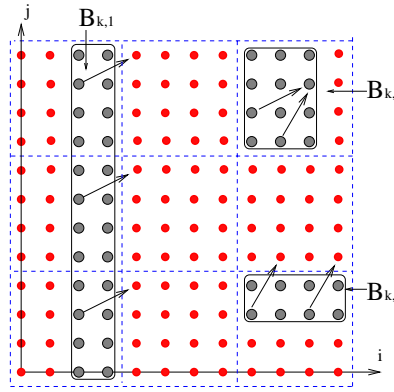

Fig. 4. Example with allocation of three buffers.

As in the previous case, every coefficient in this formula corresponds to a dimension in buffer $B_{k,n+1}$. There are $n!$ ways to organize these dimensions. To obtain the best locality in that case, we choose the following organization: $[I_1\ \boldsymbol{d}_k + 1,\ I_2\ \boldsymbol{T}, \cdots,\ I_n\ \boldsymbol{T}]$. The access function associated with buffer $B_{k,n+1}$ is defined by: $F_{k,n+1}(\boldsymbol{i'}) = ((I_1\ \boldsymbol{i'}_{in}) \bmod\ (I_1\ \boldsymbol{d}_k + 1),\ I_2\ \boldsymbol{i'}_{in}, \cdots,\ I_n\ \boldsymbol{i'}_{in})$.

As shown in *figure 4*, if the nest depth is 2 ($n = 2$), every array A_k will be replaced by three buffers : $B_{k,1},\ B_{k,2}$ and $B_{k,3}$.

4.2 Two-Level Tiling

In this case we are interested in data that lives in the cache and registers. Thus our tiling is at two levels.

Matrix $\mathcal{A}$. Matrix $\mathcal{A}(n,\ 3n)$ allows to transform every point $\boldsymbol{i} = (i_1, \cdots,\ i_n)^t \in Z^n$ into a point $\boldsymbol{i'} = (i'_1, \cdots,\ i'_{3n})^t \in Z^{3n}$ and has the following shape:

$$\mathcal{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{i,3i-2} & a_{i,3i-1} & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & a_{n,3n-2} & a_{n,3n-1} & 1 \end{pmatrix}$$

All elements of the i^{th} line of this matrix are equal to zero except:

- $a_{i,3i-2}$, which represents the external tile size on the i^{th} axis.
- $a_{i,3i-1}$, which represents the internal tile size on the i^{th} axis.
- $a_{i,3i}$, which is equal at 1.

The relationship between $\boldsymbol{i}$ and $\boldsymbol{i'}$ is given by:

$$1.\ \boldsymbol{i} = \mathcal{A}\boldsymbol{i'}$$
$$2.\ \boldsymbol{i'} = (\lfloor\frac{i_1}{a_{1,1}}\rfloor,\ \lfloor\frac{i_1 \bmod a_{1,1}}{a_{1,2}}\rfloor,\ i_1 \bmod a_{1,2}, \cdots,$$
$$\lfloor\frac{i_n}{a_{n,3n-2}}\rfloor,\ \lfloor\frac{i_n \bmod a_{n,3n-2}}{a_{n,3n-1}}\rfloor,\ i_n \bmod a_{n,3n-2})^t.$$

Matrix $\mathcal{P}$. Matrix $\mathcal{P}(3n,\ 3n)$ is a permutation matrix used to transform every point $\boldsymbol{i'} = (i'_1,\ i'_2, \cdots,\ i'_{3n})^t \in Z^{3n}$ into a point $\boldsymbol{l} = (l_1,\ l_2, \cdots,\ l_{3n})^t \in Z^{3n}$, with $\boldsymbol{l} = \mathcal{P}.\boldsymbol{i'}$.

Tiling modeling. Our tiling is represented by a transformation ω_2 :
$\omega_2 : Z^n \rightarrow Z^{3n}$

$$\boldsymbol{i} \longmapsto \boldsymbol{l} = \mathcal{P}\ .\ (\lfloor\frac{i_1}{a_{1,1}}\rfloor,\ \lfloor\frac{i_1 \bmod a_{1,1}}{a_{1,2}}\rfloor,\ i_1 \bmod a_{1,2}, \cdots,$$
$$\lfloor\frac{i_n}{a_{n,3n-2}}\rfloor,\ \lfloor\frac{i_n \bmod a_{n,3n-2}}{a_{n,3n-1}}\rfloor,\ i_n \bmod a_{n,3n-2})^t$$

As for one-level tiling, to apply tiling at two levels with fusion to the code in *figure 1(a)*, we have to shift every domain D_k by a delay $\boldsymbol{h}_k$ and these various delays should satisfy the following condition:

$$\forall\ k, \forall \boldsymbol{i} \in D'_k, \forall q : \omega_2(\boldsymbol{i} + \boldsymbol{v}_q^{k+1} - \boldsymbol{h}_{k+1} + \boldsymbol{h}_k) \preceq \omega_2(\boldsymbol{i}) \tag{3}$$

As before, one possible solution is $-\boldsymbol{h}_k = -\boldsymbol{h}_{k+1} + (\max_l\ v_{l,1}^{k+1}, \cdots,\ \max_l\ v_{l,n}^{k+1})^t$.

5 Implementation and Tests

All transformations described in this paper have been implemented in *Pips* [7]. To measure the external cache misses caused by the various transformations of the example in *Figure 1 (b)*, we used an *UltraSparc10* machine with 512 MB main memory, 2 MB external cache ($L2$) and 16 KB internal cache ($L1$). *Figure 5* gives the experimental results for the external cache misses caused by these various transformations. As one can see from this figure, all the transformations considerably decrease the number of external cache misses when compared to the initial code. Our new method of buffer allocation for tiling with fusion gives the best result and reduces the cache misses by almost a factor of 2 when compared to the initial code.

As often with cache we obtained a few points incompatible with the average behavior. We haven't explained them yet but they have not occurred with the tiled versions. The line of rate $16/L$ (L is size of external cache line) represents the theoretical values for cache misses of the initial code. We do not give the execution times, because we are interested in the energy consumption, which is strongly dependent on cache misses [3].

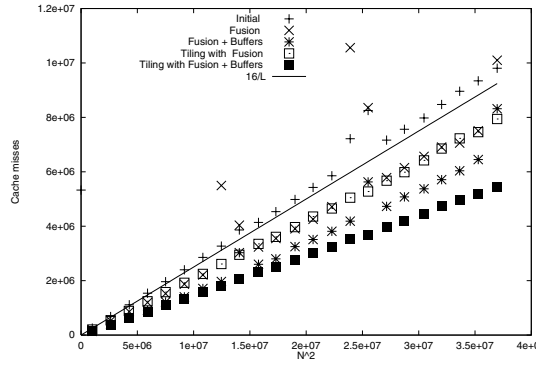

Fig. 5. External cache misses caused by transformations of code in *Figure 1(b)*.

6 Conclusion

There is a lot of work on the application of tiling[9,16,14], fusion [5,4,11,17], loop shifting [5,4] and memory allocations[6,13,8]. To our knowledge, the simultaneous application of all these transformations has not been treated. In this paper, we combined all these transformations to apply them to a sequence of loop nests.

We gave a system of inequalities that takes into account the relationships between the added delays, the various stencils, and the two matrices $\mathcal{A}$ and $\mathcal{P}$ defining the tiling. For this system of inequalities, we give a solution for a class of tiling. We have proposed a new method to increase data locality that replaces the array associated with each nest by a set of buffers that contain the live data of the corresponding array. Our tests show that the replacement of the various arrays by buffers considerably decreases the number of external cache misses.

All the transformations described in this paper have been implemented in *pips* [7]. In our future work, we shall study the generalization of our method of buffer allocations in tiling at two levels and we shall look at the issues introduced by combining for buffer and register allocations.

References

1. Youcef Bouchebaba and Fabien Coelho. Buffered tiling for sequences of loops nests. In *Compilers and Operating Systems for Low Power 2001.*
2. Youcef Bouchebaba and Fabien Coelho. Pavage pour une séquence de nids de boucles. *To appear in Technique et science informatiques*, 2000.
3. F. Cathoor and al. *Custom memory management methodology-Exploration of memory organisation for embedded multimedia system design.* Kluwer Academic Publishers, 1998.
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. Equipe PIPS. Pips (interprocedural parallelizer for scientific programs) http://www.cri.ensmp.fr/pips.
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 *Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.
10. N. Museux. *Aide au placement d'applications de traitement du signal sur machines parallèles multi-spmd*. Phd thesis, École Nationale Supérieure des Mines de Paris, 2001.
11. W. Pugh and E. Rosser. Iteration space slicing for locality. In *LCPC99*, pages 165–184, San Diego, CA, 1999.
12. F. Quilleré, S. Rajopadhye, and D. Wild. Generation of efficient nested loops from polyhedra. *International journal of parallel programming*, 28(5):496–498, 2000.
13. Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.
14. M. Wolf, D. Maydan, and Ding-Kai-Chen. Combining loop transformations considering caches and scheduling. *International Journal of Parallel Programming*, 26(4):479–503, 1998.
15. M. E. Wolf. *Improving locality and parallelism in nested loops*. Phd thesis, University of stanford, 1992.
16. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
17. H. P. Zima and B. M. Chapman. *Supercompilers for parallel and vector computers*, volume 1. Addison-Wesley, 1990.