Buffered tiling for sequences of loop nests

Youcef BOUCHEBABA and Fabien COELHO
CRI - ENSMP 35 rue Saint-Honoré
77305 Fontainebleau Cedex, France
email:{boucheba, coelho}@cri.ensmp.fr

Abstract

Usually tiling is applied to one loop nest at a time. In this paper we apply tiling and fusion simultaneously to a sequence of parallel nested loops in order to minimize data movements and energy consumption and/or to maximize the speed of execution. Each of these nests uses as input a stencil of data computed in a previous nest. After fusion and tiling, we guarantee that data necessary to the execution of an iteration has been already computed by the previous iterations by delaying the computation of consumer nest. We take into account the relation among the various stencils, the added delays and the tiling parameters and we give a solution for a class of tiling. To store only the live data elements, we compute the surface of these data for every array and during the code generation we replace this array by a buffer whose size is equal to the surface of live data. We measured cache misses for the transformed versions of the example program.

Keywords: tiling, loop fusion, program transformations, buffer allocation, energy consumption .

1 Introduction

The objective of this work is to minimize the electrical energy used during the execution of signal processing applications which are a sequence of nested loops. This energy is mostly used to data transfers among various levels of the memory hierarchy [3]. Our transformations aim at improving data locality so as to switch costly transfers from the main memory to cheaper cache or register memory accesses. To minimize these transfers, we transform these programs by using tiling [10, 4, 11, 14, 15, 13, 5], loop fusion [16, 7, 8, 13, 6, 9] and unimodular transformations [2, 1, 15, 12]. A lot of work on loop transformations has been done but most of it is only dedicated to code with a single loop nest. For codes with sequence of nested loops some authors suggest to merge these various nests and then to apply the tiling to the merged nest. Irigoin and Triolet [4] have modeled the tiling by a matrix $H \in Q^{n*n}$ and they gave a sufficient condition for the application of this tiling. Later, Xue [14] has suggested to use the tiling as a loops transformation and given the necessary and sufficient condition to its application. In our approach we combine Xue [14]'s modeling with the various techniques of loops fusion. The input programs are sequences of nested loops. Each of these nests uses a stencil of data elements produced in the previous nest. After tiling, we have to guarantee that all necessary data for the computation of given iteration has already been computed by previous iterations. For this purpose, we add a delay [8] to each nest. Instead of finding the legal condition of tiling according to the dependences of the initial code, we determine the delays which we have to add to the various nests, so our tiling is valid. Our tiling is represented by two matrices: (1) a matrix of hierarchical tiling which gives the various coefficients of tiles and (2) a permutation matrix which allows to exchange several loops and so as to specify the organization of tiles.

To better understand the problem, we consider the code in *Figure* 1, and apply successively a fusion, a fusion with buffer allocation, tiling and tiling with buffer allocation.

```
do (i = 1, N_i)
    do (j = 1, N_j)
        A1(i, j) = ....
    enddo
enddo
do (i = 2, N_i - 1)
    do (j = 2, N_j - 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
```

Figure 1: Sequence of loop nests

1.1 Fusion

To merge the two nests of our example, we must ensure that all elements of array A_1 necessary to the computation of an element $A_2(i,\ j)$ at iteration $(i,\ j)^t$

in the merged nest, have been already computed by previous iterations. To satisfy this condition we shift the computation of each element $A_1(i,\ j)$ with a delay $\vec{d} = (1,\ 0)^t$. The merged nest is:

```
do (i = 0, N_i - 1)
    do (j = 1, N_j)
        A1(i + 1, j) = ....
        if (i ≥ 2) and (2 ≤ j ≤ N_j - 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
```

Figure 2: Nest after loop fusion

1.2 Fusion with buffer allocation

To keep in memory only the live data and to avoid loading several times the same data, we replace the array A_1 by a circular buffer B_1. The size of this buffer will be equal to the surface of live data of the array A_1 which is noted by S. To compute this surface, we consider an iteration $\vec{i}$ of the merged nest. At this iteration, we use 5 data which are produced by $\vec{i_1} = \vec{i} - (2,\ 0)^t$, $\vec{i_2} = \vec{i} - (1,\ 1)^t$, $\vec{i_3} = \vec{i} - (1,\ 0)t$, $\vec{i_4} = \vec{i} - (1,\ -1)^t$ and $\vec{i_5} = \vec{i} - (0,\ 0)^t$. In *section* 3, we show that memory surface is equal to a number of iterations between $\vec{i}$ and the oldest live production ($\vec{i_1}$). Consequently $S = 2*N_j+1$. The code generated with buffer is shown in *figure* 3.

```
do (i = 0, N_i - 1)
 do (j = 1, N_j)
  B1((i * N_j + j - 1) mod S) = ...
  if (i ≥ 2) and (2 ≤ j ≤ N_j - 1)
  A2(i, j) = B1(( (i - 2) * N_j + j - 1) mod S)
           + B1(((i - 1) * N_j + j - 2) mod S)
           + B1(((i - 1) * N_j + j - 1) mod S)
           + B1(((i - 1) * N_j + j) mod S)
           + B1((i * N_j + j - 1) mod S)
 enddo
enddo
```

Figure 3: Nest with buffer allocation

1.3 Tiling

The tiling for *Exemple* 1 is specified by two matrices:

$$\mathcal{A} = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & 0 & 4 & 1 \end{pmatrix} \text{ and } \mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrix $\mathcal{A}$ transforms every point $\vec{i} = (i,\ j)^t$ in a point $\vec{i'} = (i_1,\ i_2,\ j_1,\ j_2)^t \in Z^4$ with $\vec{i} = A\ .\vec{i'}$. The matrix $\mathcal{P}$ transforms every point $\vec{i'} \in Z^4$ in a point $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ with $\vec{l} = \mathcal{P}\ .\ \vec{i'}$. At each iteration $\vec{i}$ of the second nest in the initial code we compute the element $A_2(i,\ j)$ according to the stencil of data from the array A_1 produced in the previous nest. After tiling, we obtain a new single loop nest (see *Figure* 4). The body of this nest is constituted by two statements ($A_1() = ...$ and $A_2() = ...$) and its iteration vector is represented by $\vec{l}$. To apply this tiling we have to assure that all the elements of array A_1 necessary for the computation of an element of the array A_2 at iteration $\vec{l}$ are already computed by the previous iterations. To satisfy this condition we shift the computation of the statement ($A_1(i,j) = ...$) in the initial code with a delay $\vec{d} = (d_1,\ d_2)^t = (1,\ 1)^t$. The tiled code has the following form:

```
do (l1 = 0, N_i/4 - 1)
 do (l2 = 0, N_j/4 - 1)
  do (l3 = 0, 3)
   do (l4 = 0, 3)
    A1(4l1 + l3 + 1, 4l2 + l4 + 1) = ....
    if (4l1 + l3 ≥ 2) and (4l2 + l4 ≥ 2)
    A2(4l1 + l3, 4l2 + l4) = A1(4l1 + l3 - 1, 4l2 + l4)
                           +A1(4l1 + l3, 4l2 + l4 - 1)
                           + A1(4l1 + l3, 4l2 + l4)
                           + A1(4l1 + l3, 4l2 + l4 + 1)
                           + A1(4l1 + l3 + 1, 4l2 + l4)
   enddo
  enddo
 enddo
enddo
```

Figure 4: Tiled code

We assume $N_i \bmod 4 = 0$ and $N_j \bmod 4 = 0$.

1.4 Tiling with buffer allocation

To keep in memory only the live data and to avoid loading several times the same piece of data, we propose two methods to replace the array A_1 by buffers: circular buffer and three buffer.

1.4.1 Circular buffer

We replace array A_1 by a circular buffer B_1. The size of this buffer is equal to the number of iterations between the oldest live production and a given iteration $\vec{l}$ of the tiled nest. As shown on *Figure* 5, the memory surface depends of the position of iteration $\vec{l}$ in the tile.

In the *section* 4.4.1, we show that the surface of live data in this case is a parameterized linear problem.

1.4.2 Three buffers solution

Some elements stored in the circular buffer are not live (see *Figure* 5). We use 3 buffers (see *Figure* 6):

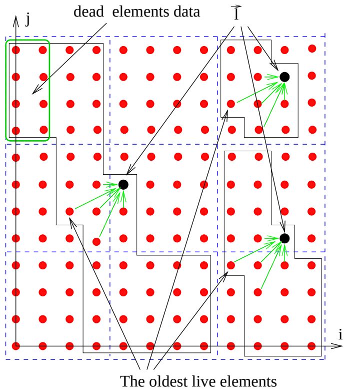

Figure 5: Circular buffer

1. Buffer B_1 contains the live elements in the same tile and it is managed as the circular buffer for the fusion (*Section* 3.1).
2. Buffer B_2 contains the data produced in a tile and used in the following tile.
3. Buffer B_3 contains the data produced by a column of tiles ($l_1 = b$) and which will be consumed by the following column ($l_1 = b + 1$).

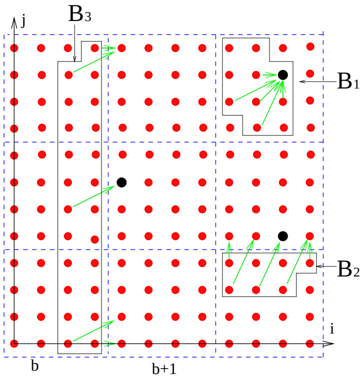

Figure 6: Three buffers

1.5 Overview

In this example we described briefly the various transformations applied to our code. We describe in more detail these transformations in the rest of paper which is organized as follows. In *Section* 2 we describe the input code. *Section* 3 and *Section* 4 present respectively loop fusion and tiling. Experimental results are discussed in *Section* 5. Conclusions are given in the last section.

2 Input code

The input code is assumed to be a sequence of nested loops of depth two. Each of these nests uses a stencil of elements represented by $R^k = \{\vec{v}_1^k,\ \vec{v}_2^k, \ldots\ldots, \vec{v}_{m_k}^k\}$.

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

Figure 7: A sequence of nested loops

Domain D_0 associated to the array A_0 is defined by user. The domains $D_{k=1\ldots n}$, are derived in the following way:
$D_k = \{\vec{i}_k \mid \forall \vec{v} \in R^k :\ \vec{i}_k + \vec{v} \in D_{k-1}\}$.
We suppose that vectors of the various stencils are ordered in the following way:
$\forall\ k : \vec{v}_1^k \preceq \vec{v}_2^k \preceq \ldots.. \preceq \vec{v}_{m_k}^k$

3 Loop fusion

Let Time_k be the translation associated to nest k and defined in the following way:
$\mathrm{Time}_k : D^k \to Z^2$ so that $\vec{i}_k \longmapsto \vec{i} = \vec{i}_k + \vec{d}_k$
The fusion of all nests is legal, if and only if, each translation Time_k meets this condition:
$(\forall \vec{i}_k,\ \forall \vec{i}_{k+1}\ , \exists \vec{v} \in R^{k+1}\ \ : \vec{i}_k = \vec{i}_{k+1} + \vec{v}$
$\Rightarrow \mathrm{Time}_k(\vec{i}_k) \preceq \mathrm{Time}_{k+1}(\vec{i}_{k+1}))$ (1).

$(1) \Leftrightarrow (\forall \vec{i}_k,\ \vec{i}_{k+1}\ , \exists \vec{v} \in R^{k+1}\ \ : \vec{i}_k = \vec{i}_{k+1} + \vec{v}$
$\Rightarrow \vec{v} \preceq -\vec{d}_k + \vec{d}_{k+1})$ (2)

do $\vec{i} \in D_{iter}$
$S_1 : C_1(\vec{i}) A_1(\vec{i} - \vec{d}_1) = A_0(\vec{i} - \vec{d}_1 + \vec{v}_1^1) \otimes \ldots. \otimes A_0(\vec{i} - \vec{d}_1 + \vec{v}_{m_1}^1)$
.
.
$S_k : C_k(\vec{i}) A_k(\vec{i} - \vec{d}_k) = A_{k-1}(\vec{i} - \vec{d}_k + \vec{v}_1^k) \otimes \ldots. \otimes A_{k-1}(\vec{i} - \vec{d}_k + \vec{v}_{m_k}^k)$
.
.
$S_n : C_n(\vec{i}) A_n(\vec{i} - \vec{d}_n) = A_{n-1}(\vec{i} - \vec{d}_n + \vec{v}_1^n) \otimes .. \otimes A_{n-1}(\vec{i} - \vec{d}_n + \vec{v}_{m_n}^n)$
enddo

Figure 8: Merged nest

The domain D_{iter} is the union of the translations of domains $D_{k=1..n}$ by vectors $\vec{d}_{k=1..n}$:
$D_{iter} = \cup_{k=1}^n (D'_k)$, with $D'_k = \{\vec{i} = \vec{i}_k + \vec{d}_k \mid \vec{i}_k \in D_k\}$.
This domain is not necessarily convex. If not, we use its convex hull to generate the code.

As instruction S_k is not executed at each iteration of domain D_{iter}, we guard it by condition

$C_k(\vec{i}) =$ if $(\vec{i} \in D'_k)$.

If we choose the frame of the last nest as an axis of the merged nest ($\vec{d_n} = (0,\ 0)^t$), the condition of legality of the fusion given in (2) will be equivalent to: $-\vec{d_k} \succeq -\vec{d}_{k+1} + \vec{v}^{k+1}_{m_{k+1}}\ :\ (1 \leq k \leq n-1)$

3.1 Fusion with buffer allocation

To save memory space, we replace the arrays A_1, A_2,..and A_{n-1} by circular buffers B_1, B_2,..and B_{n-1}.

3.1.1 Live data

Let be $D'_k = \{\vec{i}\ |\vec{O}_k \preceq \vec{i} \preceq (N_{k,i}+1,\ N_{k,j}+1)^t - \vec{O}_k\}$. The surface memory $M_k(\vec{i})$ corresponding to an iteration $\vec{i} = (i,\ j) \in D'_{k=2,n}$ is the number of elements of the array A_{k-1} which were defined before $\vec{i}$ and which are not yet fully used:

$M_k(\vec{i}) = |E_k(\vec{i})|$ with

$E_k(\vec{i}) = \{\vec{i}_1\ \in D'_{k-1} |\ \exists \vec{v} \in R^k$ and $\exists\ \vec{i}_2 \in D'_k\ :\ \vec{i}_1 - \vec{d}_{k-1} = \vec{i}_2 - \vec{d_k} + \vec{v}$ and $\vec{i}_1 \preceq \vec{i} \preceq \vec{i}_2\}$

At iteration $\vec{i}$, to compute $A_k(\vec{i} - \vec{d_k})$, we use m_k data of array A_{k-1} produced respectively by $\vec{i}_1$, $\vec{i}_2$,.....and $\vec{i}_{m_k}$ such as

$\vec{i}_q = \vec{i} - (\vec{d_k} - \vec{d}_{k-1} - \vec{v}^k_q)\ \ (q = 1, m_k)$

The oldest of these productions is $\vec{i}_1$. Consequently the surface $M_k(\vec{i})$ is bounded by the number of iterations in D'_{k-1} between $\vec{i}_1$ and $\vec{i}$. This upper boundary is given by:

$\mathrm{Sup}_k = (N_{k-1,j},\ 1)^t\ .\ (\vec{d_k} - \vec{d}_{k-1} - \vec{v}^k_1) + 1$

3.1.2 Code generation

Let $\vec{O}_k$ be the origin of the domain D'_k in the frame of the fusioned nest and $B_{k=1,n-1}$ the buffers associated to the various arrays $A_{k=1,n-1}$. Sup_{k+1} represents an upper bound number of live data of array A_k. Consequently the size of the buffer B_k can safely be Sup_{k+1}. For each buffer, an access function F_k is defined:

$F_k\ :\ D'_k \rightarrow N$ so that $\vec{i} \longmapsto F_k(\vec{i})$ with

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

To satisfy these two conditions, it is sufficient to choose $F_k(\vec{i}) = ((\vec{i} - \vec{O}_k).(N_{k,j},\ 1)^t)\ \bmod\ \mathrm{Sup}_{k+1}$.

Let's consider statement S_k of the merged code in *Figure* 8. At iteration $\vec{i}$, we compute the element $A_k(\vec{i} - \vec{d_k})$ according to m_k elements of array A_{k-1} produced respectively by $\vec{i}_1$, $\vec{i}_2$,...$\vec{i}_{m_k}$. The element $A_k(\vec{i} - \vec{d_k})$ is stored in the buffer B_k at position $F_k(\vec{i})$. The elements of the array A_{k-1} are already stored in the buffer B_{k-1} at positions $F_{k-1}(\vec{i}_1)$, $F_{k-1}(\vec{i}_2)$,..,$F_{k-1}(\vec{i}_{m_k})$ ($\vec{i}_q = \vec{i} - (\vec{d_k} - \vec{d}_{k-1} - \vec{v}^k_q)$). Thus the statement S_k will be replaced by:

$C_k(\vec{i}) B_k(F_k(\vec{i})) = B_{k-1}(F_{k-1}(\vec{i}_1)) \otimes .. \otimes B_{k-1}(F_{k-1}(\vec{i}_{m_k})$

4 Tiling

In this paper, we are interested only at the data live at the cache level, thus our tiling is a one level tiling.

4.1 Matrix $\mathcal{A}$

Matrix $\mathcal{A}$ has the following shape

$$\mathcal{A} = \begin{pmatrix} a_{1,1} & 1 & 0 & 0 \\ 0 & 0 & a_{2,3} & 1 \end{pmatrix}$$

This matrix allows us to transform every point $\vec{i} = (i,\ j)^t \in Z^2$ (initial domain) in a point $\vec{i'} = (i_1,\ i_2,\ j_1,\ j_2)^t \in Z^4$, with $\vec{i} = \mathcal{A}.\vec{i'}$.

4.2 Matrix $\mathcal{P}$

Matrix $\mathcal{P}$ is a permutation matrix which allows to transform every point $\vec{i'} = (i_1,\ i_2,\ j_1,\ j_2)^t \in Z^4$, in a point $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t \in Z^4$, with $\vec{l} = \mathcal{P}.\vec{i'}$.

To scan tile elements in a contiguous way, it is necessary that the matrix $\mathcal{P}$ meets two additional conditions:

- $\forall a, b\ : (\mathcal{P}_{a,1} = \mathcal{P}_{b,2} = 1) \Rightarrow a < b \qquad (3)$
- $\forall a, b\ : (\mathcal{P}_{a,3} = \mathcal{P}_{b,4} = 1) \Rightarrow a < b \qquad (4)$

4.3 Tiling as loop transformation

The tiling defined by matrices $\mathcal{A}$ and $\mathcal{P}$ is represented by transformation ω:

ω: $Z^2 \rightarrow Z^4$

$\vec{i} \longmapsto \vec{l} = \mathcal{P}\ (\lfloor \frac{i}{a_{1,1}} \rfloor, i \bmod a_{1,1},\ \lfloor \frac{j}{a_{2,3}} \rfloor,\ j \bmod a_{2,3})^t$.

To preserve the data flow dependences, each domain D_k is translated by a vector $\vec{d_k}$.

Theorem :

The tiling defined by matrices A and P is legal iff:

$\forall k,\ \forall \vec{i},\ \forall m : \omega(\vec{i} + \vec{v}^{k+1}_m - \vec{d}_{k+1} + \vec{d_k}) \preceq \omega(\vec{i})$

If matrix $\mathcal{P}$ satisfies conditions (3) and (4), one legal delay is:

$-\vec{d_k} = -\vec{d}_{k+1} + (\max_l\ v^{k+1}_{l,1},\ \max_l\ v^{k+1}_{l,2})^t$

The choice of this delay make the merged nest fully permutable. We knows if a loop nest is fully permutable, we can apply it any tiling parallel at axis [11]

In the following we note by $D'_k = \{\vec{i} \mid \vec{O}_k \preceq \vec{i} \preceq (N_{k,i}+1,\ N_{k,j}+1)^t - \vec{O}_k\}$ the translation of domain D_k by vector $\vec{d}_k$.

4.4 Tiling with buffer allocation

Let $\vec{i}$, $\vec{i'}$ and $\vec{l}$ be the iterations vectors associated respectively to the initial code, to the code after the application of matrix $\mathcal{A}$ and the code after the application of matrix $\mathcal{A}$ and $\mathcal{P}$. The relations between these various vectors are given by $\vec{i} = \mathcal{A}\ .\ \vec{i'}$, $\vec{l} = \mathcal{P}.\vec{i'}$ and $\vec{i} = \mathcal{A}\mathcal{P}^{-1}.\ \vec{l}$. In the following, we will assume that matrix $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. The method is valid for any permutation matrix $\mathcal{P}$. The access functions and the surface functions (Sup_k) are depend on $\mathcal{P}$.

4.4.1 Circular buffer

To compute element $A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d}_k)$ at iteration $\vec{l}$, we use m_k data of array A_{k-1} produced respectively by iterations $\vec{l}_1$, $\vec{l}_2$,.....and $\vec{l}_{m_k}$ such that:

$$\mathcal{A}\mathcal{P}^{-1}\vec{l}_q - \vec{d}_{k-1} = \mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d}_k + \vec{v}^k_q\ (q = 1, m_k)$$

The set of live elements data $M_k(\vec{l})$ is bounded by the number of iterations in $\omega(D'_{k-1})$ between the oldest of these productions and $\vec{l}$. The oldest of these productions vary according to the position of the iteration $\vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l}$, in the tile. Let $M_{k,q}(\vec{l})$ $(q = 1, m_k)$ be the number of iterations between $\vec{l}_q$ and $\vec{l}$.

$$M_{k,q}(\vec{l}) = (N_{k-1,j}.a_{1,1},\ a_{1,1}.a_{2,3},\ a_{2,3},\ 1)^t.(\vec{l} - \vec{l}_q) + 1$$

In the generated code we replace every array A_k by a buffer B_k. At any iteration $\vec{l}$, buffer B_k should contain all the live elements of array A_k. Consequently the size of this buffer is bounded by:

$$\mathrm{Sup}_{k+1} = \max(\ M_{k+1,q=1,m_{k+1}}(\vec{l}))$$

$$\begin{cases} \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l} \\ \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l}_q + \vec{d}_{k+1} - \vec{d}_k - \vec{v}^{k+1}_q\ (q = 1, m_{k+1}) \\ 0 \preceq l_{q,3} \preceq a_{1,1} - 1\ (q = 1, m_{k+1}) \\ 0 \preceq l_{q,4} \preceq a_{2,3} - 1\ (q = 1, m_{k+1}) \\ \vec{i} \in D'_{k+1} \end{cases}$$

Code generation:

For each of these domains we define an access function F_k in the following way:

$F_k : \omega(D'_k) \to N$ so that $\vec{l} \longmapsto F_k(\vec{l})$ with

1. $F_k(\omega(\vec{O}_k)) = 0$
2. $F_k(succ(\vec{l})) = \begin{cases} F_k(\vec{l}) + 1 & \text{if } (F_k(\vec{l}) \neq \mathrm{Sup}_{k+1} - 1) \\ 0 & \text{sinon} \end{cases}$

To satisfy these two conditions, it is sufficient to choose

$F_k(\vec{l}) = [(N_{k,j}.a_{1,1},\ a_{1,1}.a_{2,3},\ a_{2,3},\ 1)^t.(\vec{l} - \omega(\vec{O}_k))] \bmod \mathrm{Sup}_{k+1}$

4.4.2 Three buffers solution

Every array A_k is replaced by three buffers: $B_{k,1}$, $B_{k,2}$ and $B_{k,3}$.

1. Buffer $B_{k,1}$ contains the alive data in the same tile and it is managed as the circular buffer of the fusion. The size of this buffer is given by :

   $\mathrm{Sup}_{k+1,1} = (a_{2,3},\ 1)^t\ .\ (\vec{d}_{k+1} - \vec{d}_k - \vec{v}^{k+1}_1) + 1$
   and its access function is:

   $F_{k,1}(\vec{l}) = ((l_3,\ l_4)^t.(a_{2,3},\ 1)^t) \bmod \mathrm{Sup}_{k+1,1}$
2. Buffer $B_{k,2}$ contains the data produced in a given tile and used in the next one.
   Let $\vec{m} = \vec{d}_{k+1} - \vec{d}_k - \vec{v}^{k+1}_l \mid \forall\ q :$
   $d_{k+1,2} - d_{k,2} - v^{k+1}_{q,2} < d_{k+1,2} - d_{k,2} - v^{k+1}_{l,2}$.
   The size of this buffer is given by :

   $\mathrm{Sup}_{k+1,2} = \vec{m}\ .\ (0,\ a_{1,1})^t$ and its access function is:

   $F_{k,2}(\vec{l}) = ((l_3,\ l_4)^t.(m_2,\ 1)^t - a_{2,3} + m_2)$.
3. Buffer $B_{k,3}$ contains the data produced by a column of tiles ($l_1 = b$) and which will be consumed by the following one ($l_1 = b + 1$).
   Let $\vec{m} = \vec{d}_{k+1} - \vec{d}_k - \vec{v}^{k+1}_1$. The size of this buffer is given by :

   $\mathrm{Sup}_{k+1,3} = \vec{m}\ .\ (N_{k,j},\ 0)^t$ and its access function is:

   $F_{k,3}(\vec{l}) = (l_2,\ l_3,\ l_4)^t.(a_{2,3}.m_1,\ a_{2,3},\ 1)^t - a_{2,3}.(a_{1,1} - m_1)$.

4.5 Extension

To simplify the presentation, the input code is assumed to be a sequence of nested loops of depth two. Our method is applicable for any nest depth.

5 Experimental Results

We used an UltraSparc10 machine, which has 512 M main memory, 2 M external cache ($L2$) and 16 K data internal cache ($L1$). We measured cache misses for 1) the initial code, 2) merged nest, 3) merged nest with buffer allocation, 4) tiled code and 5) tiled code with buffer allocation. *Figure* 9 shows that the various transformations decrease considerably the number of internal cache misses as compared to the initial code. But these decreases are almost the same for the different transformations.

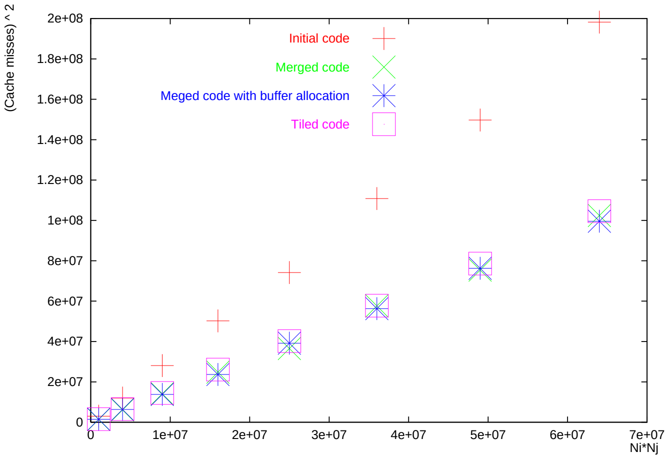

Figure 9: Internal cache misses (L1)

On the other hand, *Figure* 10 shows that fusion with buffer allocation and tiling with buffer allocation give the best result for the external cache misses. We can remark that the tiling and the fusion decrease in the same way this misses on comparison with the initial code.

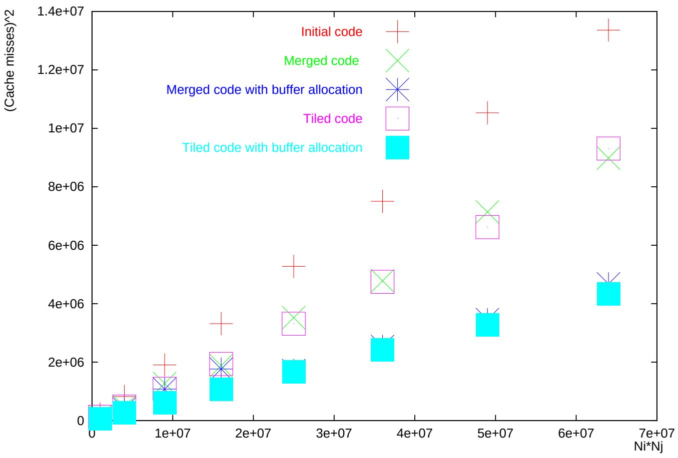

Figure 10: External cache misses (L2)

N_i and N_j in this two figures, are respectively the upper bounds of loop i and loop j of code given in example and we have assumed $N_i = N_j$.

6 Conclusion

There are many works on the application of tiling and fusion to a sequence of nested loops. To our knowledge, the application of these two transformations simultaneously has not been treated. We combine Xue results [14] who modeled tiling for single loop nest by a loop transformation and contributions on fusion [16, 7, 8, 13, 6]. Our approach combines tiling and fusion with a necessary shift of the iterations of the loops nests so as to ensure that used data has been already produced by previous iterations.

We give a system of inequalities which take into account the relation among the added delays, the various stencils, 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 computed the surface of the live data of various arrays for the merged and the tiled codes. We replaced these arrays by a circular buffers whose sizes equal to corresponding surface of live data. Our tests show that the replacement of the various arrays by a circular buffers decrease considerably the number of external cache misses.

In our future work we shall study other techniques for buffer allocations (multidimensional buffer) and the second level of tiling (level for registers).

References

[1] U. Banerjee. Loop transformations for restructuring compilers. *Kluwer Academic Publishers*, 93.

[2] D. Chesney; B. Cheng. Generalising the unimodular approach program code transformation. *Proceedings 1994 International Conference on Parallel and Distributed Systems*, pages 398–404, 1994.

[3] F. Cathoor et al. Custom memory management methodology-exploration of memory organisation for embedded multimedia system design. *Kluwer Academic Publishers,ISBN 3-540-64105-X*, 98.

[4] F. Irigoin and R. Triolet. Super-node partitioning. *In Proc. 15th Annual ACM Symp. Principles of Programming Languages.*, pages 319–329, 1988.

[5] M. Kandemir, A. Choudhary, and J. Ramanujam. I/O-conscious tiling for disk-resident data sets. *Europar*, 1999.

[6] T. SS. Abdelrahman M. Manjikian. Fusion of loops for parallelism and locality. *IEEE trans. on paral*, 97.

[7] K.S. McKinley, S. Carr, and C-W. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, 96.

[8] W. Pugh and E. Rosser. Iteration space slicing for locality. *LCPC99*, pages 165–184, 1999.

[9] V. Sarkar. Automatic selection of high order transformations in the ibm xl fortran compilers. *IBM-Journal-of-Research-and-Development*, 41:233–64, 1997.

[10] M. Wolf, D. Maydan, and Ding-Kai-Chen. Combining loop transformations considering caches and scheduling. *International-Journal-of-Parallel-Programming*, 26:479–503, 1998.

[11] M. E. Wolf. Improving locality and parallelism in nested loops. *PhD thesis, University of stanford*, 1992.

[12] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE trans. on paral*, 2, 91.

[13] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley*, 1996.

[14] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 1997.

[15] J. Xue and ChuaHuang Huang. Reuse-driven tiling for improiving data locality. *Parallel programming*, 98.

[16] H. P. Zima and B. M. Chapman. Supercompilers for parallel and vector computers. *Addison-Wesley*, 1, 1990.