Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests

Philippe Clauss and Benoît Meister
ICPS, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch FRANCE
e-mail: `{clauss, meister}@icps.u-strasbg.fr`

Abstract

One of the most efficient ways to improve program performances onto nowadays computers is to optimize the way cache memories are used. In particular, many scientific applications contain loop nests that operate on large multi-dimensional arrays whose sizes are often parameterized. No special attention is paid to cache memory performance when such loops are written. In this work, we focus on spatial locality optimization such that all the data that are loaded as a block in the cache will be used successively by the program. Our method consists in providing a new array reference evaluation function to the compiler, such that the data layout corresponds exactly to the utilization order of these data. The computation of this function concerns the field of parameterized polyhedra and Ehrhart polynomials.

Keywords Cache memory, spatial locality, loop nests, program performance optimization, optimizing compiler, parameterized polyhedron, Ehrhart polynomials.

1 Introduction

As the disparity between processor cycle times and main memory access grows, performances of programs are greatly dependent on the way the cache memory is effectively used. The unit of data transfer between main memory and cache is a block. If the same data in the cache is reused, this is called *temporal locality*. Since the unit of data in cache is a block, once the block is brought into the cache, any access to the data elements in the block will be a cache hit. This is called *spatial locality*. Typically, a cache hit takes one cycle, while a cache miss takes 8-32 cycles. It is ideal for all memory references to hit in the cache, however, since cache size is much smaller than main memory, once new data needs to be brought in from main memory, some data in the cache has to be replaced. There will be a cache miss when the replaced data is accessed in the future. Therefore, high performance requires programs to possess *cache locality*, i.e., having data reuse of the data in the cache before it has been replaced.

Most of the scientific and engineering applications contain loop nests that operate on large multi-dimensional arrays. Unfortunately, most of these loops are written without special attention paid to cache memory performance. It has already been observed that compiler techniques are useful for optimizing locality. Programmers and compiler writers often attempt to modify the access patterns of a program so that the majority of accesses are made to the nearby memory. Several efforts have been aimed at iteration space transformations and scheduling techniques to improve locality [8, 10, 12, 11] .

Unlike traditional compiler techniques which are based on re-ordering computations, we focus directly on the data space, and transform data layouts to obtain better locality. Our approach is similar to the one proposed by Kandemir *et al.* in [6, 7], except that our result consists in a mapping function from the data space to memory, instead of a layout matrix. Such information can then allow the compiler to physically organize the data to optimize spatial locality. Such an approach has several advantages. First, some programs are not amenable to loop transformations. The data dependences in a loop nest may not allow a loop transformation to improve locality. On the other hand, data space transformations are not affected by and do not place any restrictions on the data dependences. Thus they have a wider applicability for a given loop nest. Secondly, for some programs, even though an iteration space transformation is legal, there may be a data space transformation which results in better locality. In addition, unlike loop transformations, data transformations do not affect all the arrays accessed in a given loop nest. Finally, imperfectly nested loops are in general more difficult to optimize using loop transformations whereas in many cases data transformations can be successfully applied to the arrays referenced in them.

Although it is based on this worthwhile approach, the technique proposed by Kandemir *et al.* deals only with locality optimization at the innermost loop level, unlike our goal which is to consider all the iteration levels. Their technique, whose answer is given on the form of a layout matrix, attempt to distribute all the data used during the execution of the innermost loop, for given values of the enclosing loops indices, in the same array dimension, i.e., in the same column for a matrix. This technique can yield bad results in the case of a small innermost loop, since no attention is given in the utilization sequence between the different dimensions of the array, or the different columns in the case of a matrix. Moreover, examples given in [6, 7], just consist in inverting lines and columns of some matrices. No details are given in the case of a more complicated use of the data, as diagonal access or any other direction. In addition, it is not checked if the loaded data are effectively used: if the considered computations only use a small subset of a large array, then the cache will often be loaded with unuseful data, leading to bad performance.

In addition, our method allows to consider parameterized nested loops, accessing parameterized arrays: the size of the accessed arrays and the loop bounds can contain parameters, since such situations often occur in scientific and engineering applications. This is done due to our parametric and geometrical tools presented in [2, 3, 4], and implemented in the Polyhedral Library *Polylib* (http://icps.u-strasbg.fr/PolyLib). Hence we are able to compute a parameterized mapping function of the data to memory when analyzing a nested loop containing non-instanciated parameters.

To sum up, our method consists in re-ordering the storage of the used data in the order given by the computations with the following contributions:

- optimization is made for all the iteration levels,

- any reference defined by affine functions on the loop indices, and resulting in any kind of utilization sequence of the data, is considered,

- only the effectively used data are considered, avoiding unuseful load in the cache,
- the method can deal with parameterized loops and parameterized arrays.

The main consequence is that any block that is brought into the cache will be used entirely until it is replaced. Hence, our method focuses only on spatial locality, since it does not ensure that a block will never be accessed again after it has been replaced. But temporal locality can only be improved through program transformations, by bringing closer instructions referencing the same data. Therefore, our method should be associated with some program transformations for a complete optimization framework. Moreover, our method does not affect the temporal locality of the program, since it does not change the execution order of the instructions, but only the storage order of the data. So the best framework would consist in first optimizing temporal locality through some program transformations and then optimizing spatial locality by reordering the storage of data through our method.

The paper is organized as follows. In section 2, the background and motivations are presented. Section 3 is devoted to present our method for spatial locality optimization: subsection 3.1 shows how are characterized the iterations referencing a given datum and the set of effectively used data, in terms of parameterized polyhedra ; in subsection 3.2, it is shown how the array reference evaluation function of the data is computed, allowing an optimal spatial locality. Conclusions are given in section 4.

2 Background

The iteration space of a loop nest of depth n is an n-dimensional convex polyhedron P where each point is denoted by an n-vector $\overrightarrow{I} = (i_1, i_2, \ldots, i_n)$ where each i_k denotes a loop index with i_1 as the outermost loop and i_n the innermost. We will use $(i_1, i_2, \ldots, i_n)$ to denote an iteration as well as a point in the iteration space.

We denote by $l_m(i_1, i_2, \ldots, i_{m-1})$ (respectively $h_m(i_1, i_2, \ldots, i_{m-1})$) the lower (resp. the upper) limit for the loop of depth $m, m \leq n$. Such limits are defined by parametric affine functions of the enclosing loop indices. They are of the form :

$$a_1 i_1 + a_2 i_2 + \ldots + a_{m-1} i_{m-1} + b_1 p_1 + b_2 p_2 + \ldots + b_q p_q + c$$

where $a_1, a_2, \ldots, a_{m-1}, b_1, b_2, \ldots, b_q, c$ are rational constants and $p_1, p_2, \ldots, p_q$ are positive integer parameters. Thus, the polyhedron corresponding to the iteration space is bounded by parameterized linear inequalities imposed by the loop bounds.

We assume that the loops are normalized such that the step size is one. A reference to an array element is represented by the pair $(A, \overrightarrow{o})$ where A is the access matrix and $\overrightarrow{o}$ is the offset vector. Such a reference is an affine mapping $f(\overrightarrow{I}) = A \cdot \overrightarrow{I} + \overrightarrow{o}$. Hence, it can be seen as an affine transformation of the iteration space, as described in [3]. For a reference to an m-dimensional array inside an n-dimensional loop nest, the access matrix is $m \times n$ and the offset vector is of size m.

Example 1 The presentation will be illustrated with the loop nest represented on figure 1 where p is a positive integer parameter. Let us focus on array Y. For the reference $Y(i_1 + i_2 - 1, i_1 + i_3 + 2, i_2 + i_3)$, we have:

```
for i1 = 1 to p
  for i2 = i1 to 2 * i1 - 1
    for i3 = i1 - i2 to i1 + i2
      X(i1,i2,i3) := Y(i1 + i2 - 1, i1 + i3 + 2, i2 + i3) + 10
```

Figure 1: the loop nest example

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \text{ and } \vec{a} = \begin{pmatrix} -1 \\ 2 \\ 0 \end{pmatrix}$$

□

Considering any reference to array elements $A \cdot \vec{I} + \vec{a}$, we define *utilization vectors* $\overrightarrow{U_m}$ characterizing the utilization sequence order of the array elements. One vector $\overrightarrow{U_m}$ is associated with each loop level m.

Definition 1 *The utilization vector of loop level m, $\overrightarrow{U_m}$, is equal to the m^{th} column of the access matrix A.*

Example 2 *(continued)* The utilization vectors are

$$\overrightarrow{U_1} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \overrightarrow{U_2} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \text{ and } \overrightarrow{U_3} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Let us consider any array element $Y(i,j,k)$ referenced by the loop nest at a given iteration (i_1, i_2, i_3). If $i_3 < i_1 + i_2$, the next used element will be $Y((i,j,k) + \overrightarrow{U_3}) = Y(i, j+1, k+1)$ at iteration $(i_1, i_2, i_3 + 1)$. More generally, for any $1 \leq q \leq i_1 + i_2 - i_3$, the element $Y((i,j,k) + q \times \overrightarrow{U_3}) = Y(i, j+q, k+q)$ will be referenced at iteration $(i_1, i_2, i_3 + q)$.

Then, when $q = i_1 + i_2 - i_3$, the element referenced after $Y(i, j+q, k+q)$ will be $Y((i, j+q, k+q) + \overrightarrow{U_2}) = Y(i+1, j+q, k+q+1)$. The next references can again be determined using the vector $\overrightarrow{U_3}$ and so on.

□

These vectors allow to characterize the different kinds of reuse occurring in the loop nest. If at least one vector $\overrightarrow{U_m} = 0$, the array has temporal reuse in the enclosed loops $i_{m+1}, \ldots, i_n$, i.e., the same utilization sequence is repeated in the enclosed loops. If two or several vectors are linearly dependent, i.e., A is singular, some array elements may have temporal reuse.

When an array is referenced several times by the same statement, it is difficult to conciliate spatial locality optimizations for all these references. However, when all these references are characterized by the same access matrix, they are said to belong to a *uniformly generated reference* [5], and spatial locality optimization can be achieved. In the following, our method is restricted to loop nests with uniformly generated references as in [6, 7]. However, some dedicated approaches can be proposed consisting in compromise data layouts or in duplications of some data. Some future works will concern these approaches.

3 Spatial locality optimization

3.1 The effectively used data

Let $Y(\overrightarrow{I_0}) = Y(i0_1, i0_2, \ldots, i0_n)$ be any element of an array Y referenced by a considered loop nest. The set of iterations referencing this array element $Y(\overrightarrow{I_0})$ is defined by

$$P(\overrightarrow{I_0}) = \{\overrightarrow{I} \in P | A \cdot \overrightarrow{I} + \overrightarrow{a} = \overrightarrow{I_0}\}$$

If $P(\overrightarrow{I_0})$ is empty, then $Y(\overrightarrow{I_0})$ is never referenced by the loop nest and does never have to be loaded in the cache. Depending on how the data memorization process is implemented in the compiler, it can be useful, for any given array element $Y(\overrightarrow{I_0})$, to know if it is effectively used and how many times. By computing the Ehrhart polynomial $EP(\overrightarrow{I_0})$ of $P(\overrightarrow{I_0})$ [2, 4], we can provide to the compiler a function giving this information. An Ehrhart polynomial is a parametric expression of the exact number of integer points in a parameterized polyhedron and can be computed using our program available at http://icps.u-strasbg.fr/Ehrhart/program/. Hence, if $EP(\overrightarrow{I_0}) = 0$, then $Y(\overrightarrow{I_0})$ is never referenced by the loop nest.

If $P(\overrightarrow{I_0})$ is reduced to a single iteration, then array Y has no temporal reuse. The best locality will then be obtained by indexing the datum with the position of the iteration that references it, relatively to the execution order: if the q^{th} iteration references $Y(\overrightarrow{I_0})$, then $Y(\overrightarrow{I_0})$ will be mapped at address $b + q \times w$ in main memory, where b is the base address of array Y, and w is the size in bytes of an array element.

If $P(\overrightarrow{I_0})$ contains several iterations, we choose to consider the first iteration referencing $Y(\overrightarrow{I_0})$. This iteration is the lexicographic minimum of $P(\overrightarrow{I_0})$. Hence this minimum has to be computed first. It is done using our geometrical tools as described in [4]. Since in the case of temporal reuse, the same utilization sequence will occur several times, this choice ensures spatial locality each time that such a sequence occurs. Moreover, if this utilization sequence is small enough to be loaded entirely in the cache, then temporal locality will also be achieved. If temporal locality has already been optimized, our method will not affect this previous optimization, but will eventually improve a previous spatial locality optimization.

Example 3 *(continued)* The set of iterations referencing any array element $Y(i0_1, i0_2, i0_3)$ is defined by

$$\begin{aligned} P(i0_1, i0_2, i0_3) &= \{(i_1, i_2, i_3) \in P | i_1 + i_2 - 1 = i0_1, i_1 + i_3 + 2 = i0_2, i_2 + i_3 = i0_3\} \\ &= \{(i_1, i_2, i_3) \in P | i_1 = \tfrac{i0_1 + i0_2 - i0_3 - 1}{2}, i_2 = \tfrac{i0_1 - i0_2 + i0_3 + 3}{2}, i_3 = \tfrac{-i0_1 + i0_2 + i0_3 - 3}{2}\} \end{aligned}$$

Observe that for a given datum $Y(i0_1, i0_2, i0_3)$, $P(i0_1, i0_2, i0_3)$ is either empty, when the resulting values of $\frac{i0_1 + i0_2 - i0_3 - 1}{2}$, $\frac{i0_1 - i0_2 + i0_3 + 3}{2}$ or $\frac{-i0_1 + i0_2 + i0_3 - 3}{2}$ are not integer or do not belong to P, or contains a single iteration. The computation of the Ehrhart polynomial of $P(i0_1, i0_2, i0_3)$ gives the following answer:

$$EP(i0_1, i0_2, i0_3) = \begin{cases} 0 & if\ (i0_1 + i0_2 + i0_3)\ mod\ 2 = 0 \\ 1 & otherwise \end{cases}$$

For example, since $EP(1, 3, 2) = 0$, $Y(1, 3, 2)$ is never referenced by the loop nest.

□

The set of effectively used data is defined by

$$D_Y = \{\overrightarrow{I_0} | \overrightarrow{I_0} = A \cdot \overrightarrow{I} + \overrightarrow{a}, \overrightarrow{I} \in P\}$$

These data are associated with the points resulting from the affine transformation $(A, \overrightarrow{a})$ of the iteration space. We have presented in [3] a method allowing to determine the exact number of these effectively used data. This set is not generally a convex polyhedron containing a regular lattice of points since some "holes" do occur irregularly [3].

In this work, an exact determination of D_Y is not necessary. The convex hull of D_Y, $Conv(D_Y)$, is sufficient in order to reduce the answer of our method to the strictly useful informations. For this purpose, we use our parametric vertices finding program presented in [9, 4]. This program computes the parametric coordinates of the vertices of a convex parameterized polyhedron defined by some linear parameterized inequalities. The program answer is given as a set of convex adjacent domains of the parameters associated with some vertices. In order to compute $Conv(D_Y)$, we compute the parametric vertices of the parameterized polyhedron $P(\overrightarrow{I_0})$. Hence, the domains of the parameters computed by the program will define the convex hull of the values taken by $\overrightarrow{I_0}$, i.e., $Conv(D_Y)$.

Example 4 *(continued)* The set of effectively used data is defined by $D_Y = \{(i0_1, i0_2, i0_3) | i0_1 = i_1 + i_2 - 1, i0_2 = i_1 + i_3 + 2, i0_3 = i_2 + i_3, 1 \le i_1 \le p, i_1 \le i_2 \le 2i_1 - 1, i_1 - i_2 \le i_3 \le i_1 + i_2\}$. The convex hull of D_Y, $Conv(D_Y)$, is determined by computing the parametric vertices of $P(i0_1, i0_2, i0_3)$. The program gives the following answer:

$$\begin{aligned} Conv(D_Y) = \{(i0_1, i0_2, i0_3) | \quad & i0_1 + i0_2 \le i0_3 + 2p + 1, 3i0_3 + 7 \le i0_1 + 3i0_2, \\ & i0_2 \le i0_3 + 2, i0_1 + i0_2 \le 3i0_3 + 1, \\ & i0_2 + i0_3 \le 3i0_1 + 5\} \end{aligned}$$

3.2 Mapping array elements to memory

The lexicographic minimum of the set $P(\overrightarrow{I_0})$ defines the first iteration referencing any array element $Y(\overrightarrow{I_0})$. If $P(\overrightarrow{I_0})$ contains only one point, this point defines the unique iteration referencing $Y(\overrightarrow{I_0})$. Let $\overrightarrow{I_{min}}$ be this point whose coordinates are affine functions of $\overrightarrow{I_0}$. The position of this iteration, relatively to the execution order, gives the position index in main memory of the referenced datum $Y(\overrightarrow{I_0})$, relatively to the base address b of Y. This will ensure an optimal spatial locality in the sense that the datum referenced just before or just after $Y(\overrightarrow{I_0})$ will be contiguous in memory with $Y(\overrightarrow{I_0})$.

The position of iteration $\overrightarrow{I_{min}}$ is determined by computing the number of iterations that occur before $\overrightarrow{I_{min}}$. The set of iterations occurring before $\overrightarrow{I_{min}}$ (included) is defined by:

$$P(\overrightarrow{I_{min}}) = \{\overrightarrow{I} \in P | \overrightarrow{I} \lhd \overrightarrow{I_{min}}\}$$

where $\lhd$ denotes the lexicographic order. In order to compute the number of iterations in $P(\overrightarrow{I_{min}})$, the lexicographic inequality has first to be transformed into linear inequalities. This transformation will result in a decomposition of $P(\overrightarrow{I_{min}})$ as a union of disjoint convex polyhedra in the following way:

$$P(\overrightarrow{I_{min}}) = P_1(\overrightarrow{I_{min}}) \cup P_2(\overrightarrow{I_{min}}) \cup P_3(\overrightarrow{I_{min}}) \cup \ldots \cup P_n(\overrightarrow{I_{min}})$$

where

$$\begin{array}{ll}
P_1(\overrightarrow{I_{min}}) & = \{\overrightarrow{I} \in P | i_1 < i_{1,min}\} \\
P_2(\overrightarrow{I_{min}}) & = \{\overrightarrow{I} \in P | i_1 = i_{1,min}, i_2 < i_{2,min}\} \\
P_3(\overrightarrow{I_{min}}) & = \{\overrightarrow{I} \in P | i_1 = i_{1,min}, i_2 = i_{2,min}, i_3 < i_{3,min}\} \\
\ldots & \\
P_n(\overrightarrow{I_{min}}) & = \{\overrightarrow{I} \in P | i_1 = i_{1,min}, i_2 = i_{2,min}, \ldots, i_{n-1} = i_{n-1,min}, i_n \leq i_{n,min}\}
\end{array}$$

Since $\overrightarrow{I_{min}}$ is defined as an affine function of $\overrightarrow{I_0}$, $P(\overrightarrow{I_{min}})$ is a union of polyhedra parameterized by $\overrightarrow{I_0}$. By computing the Ehrhart polynomials $EP_q(\overrightarrow{I_0})$ of each of these polyhedra, and by summing the results, we obtain the number of iterations defined by $P(\overrightarrow{I_{min}})$ and parameterized by $\overrightarrow{I_0}$. This final result provides a mapping function of any array element $Y(\overrightarrow{I_0})$ to main memory, by giving its position index relatively to the base address b of array Y. It is expressed as the Ehrhart polynomial $EP(\overrightarrow{I_0})$ of the union of polyhedra $P(\overrightarrow{I_{min}})$.

Finally, the obtained mapping function may be used by the compiler during the array reference evaluation process: instead of classically evaluating any reference to an array element [1], the process should be replaced by the evaluation of the previously computed Ehrhart polynomial.

The array reference function that we provide seems to represent very expensive calculations for a compiler. Moreover, if this function is parameterized by some size parameters whose values are not known at compilation time, it has to be evaluated during the execution of the program. But the implementation of this process should use the resulting perfect link between the storage order of the data and the reference order of these data: since the data are referenced in the same order than their storage order, only the adress of the first referenced data has to be evaluated. Then the next adress is simply obtained by incrementing this initial adress by the storage size of one data and so on.

This compiler transformation can be seen as the production of the following execution code:

```
$R = EP\left(A \cdot (\lceil l_1 \rceil, \lceil l_2(\lceil l_1 \rceil) \rceil, \lceil l_3(\lceil l_1 \rceil, \lceil l_2(\lceil l_1 \rceil) \rceil) \rceil, \ldots) + (o_1, o_2, o_3, \ldots)\right)$
for $i_1 = l_1$ to $h_1$
    for $i_2 = l_2(i_1)$ to $h_2(i_1)$
        ...
            for $i_n = l_n(i_1, i_2, ..., i_{n-1})$ to $h_n(i_1, i_2, ..., i_{n-1})$
                $\ldots Y(R) \ldots$
                $R = R + 1$
```

where the referenced array Y is now seen as a one-dimensional array.

Example 5 *(continued)* The set $P(i0_1, i0_2, i0_3)$ contains at most one point defining a unique iteration referencing any array element $Y(i0_1, i0_2, i0_3)$. This iteration is defined by $\overrightarrow{I_{min}} = (\frac{i0_1+i0_2-i0_3-1}{2}, \frac{i0_1-i0_2+i0_3+3}{2}, \frac{-i0_1+i0_2+i0_3-3}{2})$. Hence, the number of iterations occurring before is determined by computing the Ehrhart polynomial of the following union of polyhedra:

$$\begin{aligned}
&P(\tfrac{i0_1+i0_2-i0_3-1}{2}, \tfrac{i0_1-i0_2+i0_3+3}{2}, \tfrac{-i0_1+i0_2+i0_3-3}{2})\\
&= \{(i_1,i_2,i_3) \in P | i_1 < \tfrac{i0_1+i0_2-i0_3-1}{2}, (i0_1,i0_2,i0_3) \in Conv(D_Y)\}\\
&\cup \{(i_1,i_2,i_3) \in P | i_1 = \tfrac{i0_1+i0_2-i0_3-1}{2}, i_2 < \tfrac{i0_1-i0_2+i0_3+3}{2}, (i0_1,i0_2,i0_3) \in Conv(D_Y)\}\\
&\cup \{(i_1,i_2,i_3) \in P | i_1 = \tfrac{i0_1+i0_2-i0_3-1}{2}, i_2 = \tfrac{i0_1-i0_2+i0_3+3}{2}, i_3 \le \tfrac{-i0_1+i0_2+i0_3-3}{2}, (i0_1,i0_2,i0_3) \in Conv(D_Y)\}
\end{aligned}$$

On the first set, our program gives the following answer:

$$\begin{aligned}
&EP_1(i0_1,i0_2,i0_3) = -\tfrac{1}{8}i0_3^3 + \tfrac{3}{8}i0_2 i0_3^2 + \tfrac{3}{8}i0_1 i0_3^2 - \tfrac{3}{4}i0_3^2 - \tfrac{3}{8}i0_2^2 i0_3 - \tfrac{3}{4}i0_1 i0_2 i0_3 + \tfrac{3}{2}i0_2 i0_3\\
&-\tfrac{3}{8}i0_1^2 i0_3 + \tfrac{3}{2}i0_1 i0_3 - \tfrac{11}{8}i0_3 + \tfrac{1}{8}i0_2^3 + \tfrac{3}{8}i0_1 i0_2^2 - \tfrac{3}{4}i0_2^2 + \tfrac{3}{8}i0_1^2 i0_2 - \tfrac{3}{2}i0_1 i0_2 + \tfrac{11}{8}i0_2 + \tfrac{1}{8}i0_1^3\\
&-\tfrac{3}{4}i0_1^2 + \tfrac{11}{8}i0_1 - \tfrac{3}{4}
\end{aligned}$$

on the second, the answer is:

$$EP_2(i0_1,i0_2,i0_3) = i0_1 i0_3 - i0_1 i0_2 + i0_3 - i0_2 + 2i0_1 + 2$$

and on the third:

$$EP_3(i0_1,i0_2,i0_3) = \frac{3}{2}i0_3 - \frac{1}{2}i0_2 - \frac{1}{2}i0_1 + \frac{3}{2}$$

Finally, the memory address of any array element $Y(i0_1, i0_2, i0_3)$, resulting in an optimal spatial locality, is given by:

$$EP(i0_1,i0_2,i0_3) \times w + b = (EP_1(i0_1,i0_2,i0_3) + EP_2(i0_1,i0_2,i0_3) + EP_3(i0_1,i0_2,i0_3)) \times w + b$$

where b denotes the base address of array Y.

For example, let us consider the 3 successive iterations $(3,5,8)$, $(4,4,0)$ and $(4,4,1)$. The referenced array elements are respectively $Y(7,13,13)$, $Y(7,6,4)$ and $Y(7,7,5)$. We now verify that the array reference evaluation function we just determined, results in contiguous memory addresses for these array elements:

$$\begin{array}{lll}
EP(7,13,13) & = 15+16+11 & = 42\\
EP(7,6,4) & = 42+0+1 & = 43\\
EP(7,7,5) & = 42+0+2 & = 44
\end{array}$$

Hence, $Y(7,13,13)$, $Y(7,6,4)$ and $Y(7,7,5)$ will respectively be mapped at memory addresses $42 \times w + b$, $43 \times w + b$ and $44 \times w + b$, leading to optimal spatial locality.

4 Conclusion

We have presented a method dedicated to replace the array reference evaluation process of the compiler with a new one resulting in an optimal spatial locality. At this time, our method is restricted to loop nests containing at most one uniformly generated reference (UGR) for each referenced array. In future works, some strategies will be proposed in order to conciliate optimizations for non-UGRs, as array duplication for example. Anyway, many scientific and engineering applications are concerned with UGR and their performance can be greatly improved by our technique.

Our method can also be used to maximize instruction level parallelism in processors, since spatial locality optimization in loops favours the use of the SIMD units and the simultaneous use of all the fonctionnal units.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1987.

[2] Ph. Clauss. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs. *10th ACM Int. Conf. on Supercomputing*, Philadelphia, May 1996. Also available as Research Report ICPS 96-03. `http://icps.u-strasbg.fr/pub-96/pub-96-03.ps.gz`

[3] Ph. Clauss. Handling Memory Cache Policy with Integer Points Countings. LNCS 1300, pp 285-293, *Euro-Par'97*, Passau, Germany, August 1997.

[4] Ph. Clauss and V. Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, Vol. 19, No. 2, Kluwer Academic Pub., July 1998.

[5] D. Gannon, W. Jalby and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587-616, 1988.

[6] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Optimizing Spatial Locality in Loop Nests using Linear Algebra. *Proc. 7th Int. Workshop on Compilers for Parallel Computers*, Sweden, June 1998.

[7] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. *Proc. 11th ACM Int. Conf. on Supercomputing*, Melbourne, Australia, July 1998.

[8] W. Li. *Compiling for NUMA parallel machines.* Ph.D. dissertation, Dept. Computer Science, Cornell University, Ithaca, NY, 1993.

[9] V. Loechner and D.K. Wilde. Parameterized polyhedra and their vertices. *Int. Journal of Parallel Programming*, Vol. 25, No. 6, 1997.

[10] K. McKinley, S. Carr and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[11] M. Wolf and M. Lam. A data locality optimizing algorithm. *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30-44, Toronto, Ont., June 1991.

[12] M. Wolf. *Improving locality and parallelism in nested loops.* Ph.D. dissertation, Dept. Computer Science, Stanford University, CSL-TR-92-538, 1992.